# Notas Angular

## How Are They Different?

**1. Angular está orientado a móviles y tiene mejor rendimiento**

Angular 1.x no se hizo para soportar móviles, mientras que Angular está orientado desde el principio a dar buen rendimiento y funcionar bien en dispositivos móviles.

AngularJS se diseñó para crear apps Web [de tipo SPA](https://www.campusmvp.es/recursos/post/Video-que-son-las-Single-Page-Applications.aspx), con enlazado a datos bidireccional. No había soporte para móviles, aunque sí existen otras bibliotecas que hacen que Angular 1.x se ejecute en móviles.

**2. Angular ofrece más opciones a la hora de elegir lenguajes**

Angular ofrece la posibilidad de elegir entre varios lenguajes a la hora de programar. Para escribir código Angular puedes usar cualquiera de los lenguajes: ECMAScript 5 de toda la vida, ES6, TypeScript o incluso Dart (de Google). Mientras que con Angular 1.x puedes usar ES5, ES6 y Dart.

Poder usar **TypeScript** es un gran avance ya que [**es una forma genial de escribir JavaScript**](https://www.campusmvp.es/recursos/post/por-que-aprender-typescript.aspx). TypeScript es el lenguaje por defecto para desarrollar en Angular, y el que sin duda vas a querer utilizar. La mayor parte la documentación que encontrarás por ahí estará creada con TypeScript, así que debes aprenderlo.

**3. Los controladores y el $scope de Angular 1.x han desaparecido**

Podemos decir que en Angular los controladores se substituyen por **componentes**. Angular se basa en componentes web, con las ventajas que ello supone al adoptar un estándar de futuro, que cuando esté bien soportado por todos los navegadores ofrecerá más rendimiento todavía.

**4. La sintaxis de las directivas estructurales ha cambiado**

En Angular, la sintaxis de las directivas estructurales ha cambiado, ng-repeat se sustituye por \*ngFor, por ejemplo.

**Directivas estructurales Angular 1.x:**

<ul>

<li ng-repeat="prod in productos">

{{prod.name}}

</li>

</ul>

**Directivas estructurales Angular:**

<ul>

<li \*ngFor="prod of productos">

{{prod.name}}

</li>

</ul>

El signo Asterico(\*) se usa como prefijo para directivas estructurales, in se sustituye por of y se usa la sintaxis *camelCase*. Hay muchos más detalles de esta nueva sintaxis, pero de momento quédate con esto.

**5. Angular usa directamente las propiedades de los elementos y los eventos estándar del DOM**

Uno de los mayores cambios en Angular es, que usa directamente las propiedades de los elementos y los eventos estándar del DOM.

Por ello, muchas de las directivas integradas disponibles en Angular 1.x ya no se necesitan, como por ejemplo: ng-href, ng-src, ng-show o ng-hide. Angular usa directamente href, src y propiedades hidden para obtener el mismo resultado.

Y lo mismo se puede decir de las directivas basadas en eventos como ng-click o ng-blur.

En AngularJS:

<button ng-click="doSomething()">

En Angular simplemente tomas el evento estándar y lo envuelves entre paréntesis:

<button (click)="doSomething()">

Nuevamente aquí hay muchos otros detalles a tener en cuenta, pero quédate con esta idea principal.

**6. La directiva de datos enlazados en una sola dirección (*one-way data binding*) se sustituye por [property]**

En Angular 1.x, ng-bind se usa para enlazar datos en una sola dirección (*one-way data binding*), lo que quiere decir que sólo se modifica el enlace desde el código hacia la vista, pero no al revés, lo que permite un mayor control de flujo dentro de la aplicación.

Con Angular esto se reemplaza por [property], siendo "property" una propiedad válida del elemento del DOM sobre el que actuamos.

Por ejemplo, en Angular 1.x escribíamos:

<input ng-bind="prod.name"></input>

En Angular se utilizan simplemente corchetes sobre la propiedad estándar:

<input [value]="prod.name"></input>

Aunque existen otras variantes para lograr lo mismo.

**7. Enlaces de datos de doble dirección (*two-way data binding*): ng-model se sustituye por [(ngModel)]**

Este es el enlazado que todo el mundo conoce y usa en AngularJS. En Angular se retira esta sintaxis para lograr mayor seguridad, control y mejora del rendimiento.

En Angular 1.x hacíamos esto para tener un enlazado en dos direcciones:

<input ng-model="prod.name"></input>

En Angular la sintaxis equivalente sería:

<input [(ngModel)]="prod.name"></input>

Este *doble-binding* ofrece ventajas relevantes en la gestión avanzada de formularios.

**8. Ha cambiado la sintaxis de la inyección de dependencias**

Una de las grandes ventajas de Angular es la inyección de dependencias. Con Angular hay una manera distinta de llevar a cabo esto. Como en Angular todo son "clases", la inyección de dependencias se consigue mediante constructores.

Link: <https://www.campusmvp.es/recursos/post/las-10-principales-diferencias-entre-angularjs-y-angular.aspx>

**Performance**

AngularJS was originally developed for designers, not developers. Although there were a few evolutionary improvements in its design, they were not enough to fulfill developer requirements. The later versions, Angular 2 and Angular 4, have been upgraded to provide an overall improvement in performance, especially in speed and dependency injection.

**1. Speed**

By providing features like 2-way binding, AngularJS reduced the development effort and time. However, by creating more processing on the client side, page load was taking considerable time. Angular2 provides a better structure to more easily create and maintain big applications and a better change detection mechanism. Angular 4 is the fastest version yet.

**2. Dependency injection**

Angular implements unidirectional tree-based change detection and uses Hierarchical Dependency Injection system. This significantly boosts performance for the framework.

# Angular's template syntax

Common features of Angular's template syntax:

* \*[ngFor](https://angular.io/api/common/NgForOf) Structural directives are responsible for HTML layout. They shape or reshape the DOM's *structure*, typically by adding, removing, or manipulating elements. Structural directives are easy to recognize. An asterisk (**\***) precedes the directive attribute name as in this example.
* \*[ngIf](https://angular.io/api/common/NgIf)
* Interpolation {{ }} Para acceder a información del controlador y mostrarla como texto.
* Property binding [ ] Acceder a una propiedad del HTML y añadirle información del controlador.
* Event binding ( ) Para a un evento añadirle la función a llamar en la controladora

<https://angular.io/start>

# Instalación

<https://openwebinars.net/blog/instalacion-angular-8-requisitos/>

<https://www.agiratech.com/best-angular-code-editors-ide/>

**Instalacion angular y mi primera app**

https://www.c-sharpcorner.com/article/three-steps-to-install-angular-and-create-first-hello-world-angular-app/

## Comandos CMD para angular

node -v -> Para obtener la verison actual de node

npm -vdi-r-pena -> para obtener la version actual de npm

npm install -g typescript -> para instalar typescript

npm install -g @angular/cli -> instalar angular

npm install --save [bootstrap@3](mailto:bootstrap@3) → let you install boostrap version 3 to your project

**Angular commands:**

|  |  |
| --- | --- |
| Commands | Explenation |
| **ng --version** | To know the version |
| **ng new my-firt-app** | for create a new project |
| **ng serve** | For compile and execute the application. Before that you should be located in the directory of the app files are. Any change over the app will be refresh automatically. |
| **ng generate component name** | Command to generate a component automatically inside on the app folder. Also can be used the abbreviation **ng g c name** |
| **ng g d directive-name** | Command to generate a directive. Its means generate directive. |
| **ng g service service\_name** | Command to generate a service. It create a file called: service\_name.service.ts and a class inside with name Service\_NameService |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Errors in Angular installation**

https://stackoverflow.com/questions/46623571/angular-ng-command-not-found/46623602

## Visual Studio Code

**Extensions to add**

This are good extension to add to the visual studio code IDE when you are working with angular.

<https://medium.com/frontend-coach/7-must-have-visual-studio-code-extensions-for-angular-af9c476147fd>

TODO: take a look, and test

# Course Angular

## Session 1

**Forum of the course**

As a student of this course you also get free access to our “Academind Community” on Discord: <https://discord.gg/gxvEWGU>

There, you can find like-minded people, discuss issues, help each other, share progress, successes and ideas and simply have a good time!

Note: my logging it the [leandrod.pty@gmail.com](mailto:leandrod.pty@gmail.com) account and password.

General Notes

* Angular its is SPA.
* Angular have a new version over 6 month, but all of they are compatible.
* If you want to dive deeper into the CLI and learn more about its usage, have a look at its official documentation: <https://github.com/angular/angular-cli/wiki>
* **app.module.t**s: In this file is where we tell to Angular the modules or package that going to be used by the app.
* **node\_modules**: in this directory inside every project the nmp dependencies installed will be. For example the version of typescroopt, bootstrap, etc. All that we install with NPM.
* **angular.json**: in this file is where all the configuration of the project will be. For example in this file we add the the specific version of bootstrap we going to use. The name of the project, etc.
* Start flow.

1. Angular execution start from the main.ts file where its passed as parameter the AppModule to platformBrowserDynamic().bootstrapModule(AppModule).

2. The file app.module.ts is imported in main.ts and paseed. In app.module.ts we inject the component AppComponent to Angular.

3. Then angular analyze it the component AppComponent and all its files: .html, .css, .ts.

4. Now the index.html file how is the main html can draw the component AppComponent.

<body>

  <app-root></app-root>

</body>

AppComponent have in its app.component.html file son html to be embedded in the index.html.

* Angular in the end is a Js framework changing your DOM(HTML) at run time.

## Session 2

### Components

* Angular is based on component with its own html, css if needed and ts file with the logic.
* Component .ts is not more than a typescript class that should have the name export. The export name let use this component, and be injected.
* After you create your component in its folder with its file, angular don't scant the component. You have to associate, register this component into a module. An angular module is used to encapsulate, group together different pieces like for example components into packages. For the most apps the app module will be enough.
* **ng generate component name** → command to generate a component automatically inside on the app folder. Also can be used the abbreviation **ng g c name**
* If you want in a component you can put instead of a **templateUrl** to a file, put the word **template,** and you are creating an inline template. Also if you want to put a lot of lines you have to use instead of quotes (‘’) the symbols (``).
* Just like template, for css we can do the same, we can use styleUrls for add in the line styles of css, instead of use styleUrls for adding the styles files.
* Selector. The selector function like a css styles. If you put the selector inside [] your directive will be used as a propertiy of an html tag. Also if a dot is used before the name of the directive, then the directive will be used as an html class.

### Databinding

its the communication between the typescript code (business logic) and Template (html) code.

**String interpolation:**

- When you put a variable or function front the logic (ts) between {{}} in the html template. Also expression could be written between the {{}}.

- At the end this element will be converted to string, angular will do it for you if the value can be converted to string.

Example:

<h4 class="list-group-item-heading">{{recipe.name}}</h4>

<img [src]="recipe.imagePath"

                     alt="{{recipe.imagePath}}"

                     class="img-responsive" style="max-height: 50px;">

**Property binding**

- When we put an html property between [] we are telling that this property will be binding and controlled by angular. Then we assign the value of a variable inside double quotes “”. Example:

<button

    class="btn btn-primary"

    [disabled]="allowNewServer">Add server</button>

**Even Biding**

- We have to put the even between () and also write the even clear, without prepositions. Then the assignation and double quotes “”. Example:

<button

    class="btn btn-primary"

    [disabled]="!allowNewServer"

    (click)="onServerCreation()">Add server</button>

**Bindable Properties and Events**

How do you know to which Properties or Events of HTML Elements you may bind? You can basically bind to all Properties and Events - a good idea is to console.log() the element you're interested in to see which properties and events it offers.

Important: For events, you don't bind to onclick but only to click (=> (click)).

The MDN (Mozilla Developer Network) offers nice lists of all properties and events of the element you're interested in. Googling for YOUR\_ELEMENT properties or YOUR\_ELEMENT events should yield nice results.

**Passing and Using Data with Event Binding**

How to get the even data of an HTML element ? If to an event of an input, for example we hava (clic)= "onServerFunction($event)"

If we pass the variable $event, we are sending to the TS the data and event contained in the object, in this case an input.

**32. Two-Way-Databinding**

Using two ways of data binding the value of an input for example associated to a variable in the TS (serverName in this example), any change in the TS is reflected in the template and vice versa.

Example:

<input type=”text” [(ngModel)]=”serverName”>

**Important: FormsModule is Required for Two-Way-Binding!**

Important: For Two-Way-Binding (covered in the next lecture) to work, you need to enable the ngModel  directive. This is done by adding the FormsModule  to the imports[]  array in the AppModule.

You then also need to add the import from @angular/forms  in the app.module.ts file:

import { FormsModule } from '@angular/forms';

### Understanding Directives

**Using ngIf to Output Data Conditionally**

Ngif is and structural directive, so should be used with asterisk before, this is because structural directives modify the DOM. If the value its true the element is draw in this example, else it not draw.

Example:

<p \*ngIf="serverCreated">Server was created, name is {{ serverName }}</p>

**Enhancing ngIf with an Else Condition**

After the \*ngIf you can put ; else, and do something in the other case.

For example:

<p \*ngIf="serverCreated ; else noServer">Server was created, server name is {{ serverName }}</p>

<ng-template #noServer>

    <p>No server was created!</p>

</ng-template>

<https://ultimatecourses.com/blog/angular-ngif-else-then>

**Styling Elements Dynamically with ngStyle**

The directive that are not of type structural directive are called attribute directive. They don’t add or remove elements, they only change the element they were placed on.

ngStyle let you change the style of an object dynamically.

Example:

<p [ngStyle]="{'background-color': getColor(1), 'color': getColor(2)}">{{numClick}}</p>

**Applying CSS Classes Dynamically with ngClass**

On the other hand ngClass let you add or remove dynamically a CSS class to an object if a certain condition is true. When we call it we have to pass a key:value, the key it’s the class name we going to add and the value is the condition determining whatever this class should be showed or not.

Example:

<p [ngClass]="{hidding : displayDetailsParagraph == true}">'Secret password = tuna'</p>

**Outputting Lists with ngFor**

\*ngFor it’s a structural directive that let you add elements to the DOM dynamically, just like a FOR of any language.

Example:

<div \*ngFor="let numClick of numClicks">

    <p>{{numClick}}</p>

</div>

if we want access to the index, current index in the for we can declare a variable an assing the reserved word in this context “index”:

<div \*ngFor="let numClick of numClicks; let i = index"

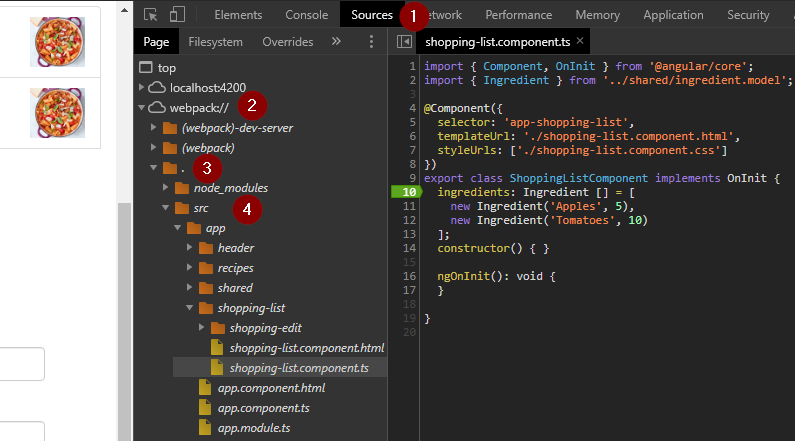
    [ngStyle]="{'background-color': getColor()}"

    [ngClass]="{'letters-color': i > 5}">{{numClick}}

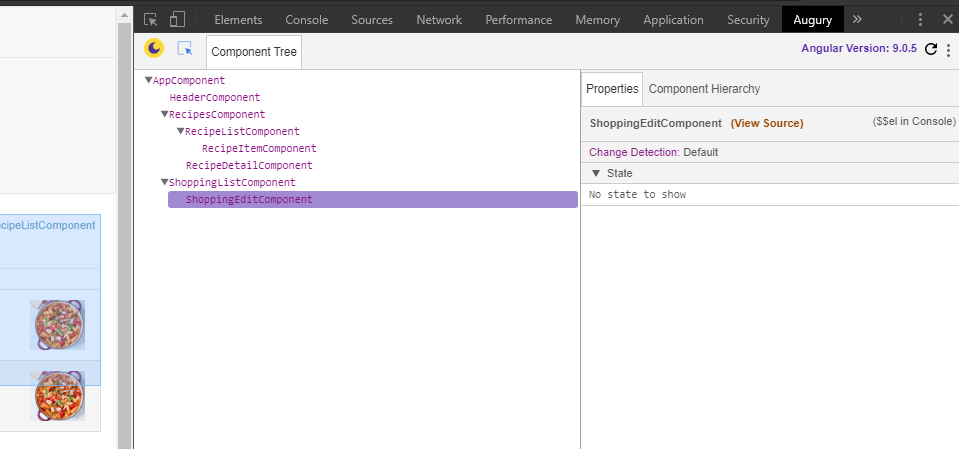
</div>

## Session 4

### How debug in Chrome

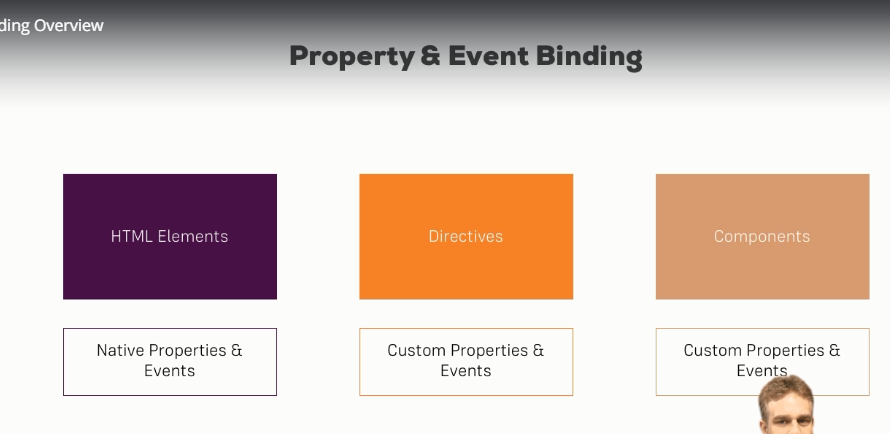


### Using Augury to Dive into Angular Apps



## Section 5: Components & Databinding Deep Dive

In Angular you can use property even binding in HTML, in directives like (ngclass or ngstyles) but also we can bind our own custom properties (variables) or events of our component.



### @Input

By default all properties (variables) of a component are only accessible inside this component. If you want one propertie be accessible for outside, you have to be explicit using the @Input decorator. This is the way we pass information from a component to a dawn component (hacia abajo), example:

**export** **class** ServerElementComponent **implements** OnInit {

@Input() element: {type: string, name: string, content: string};

**constructor**() { }

ngOnInit(): void {

}

}

Note: input need to be execute, it’s like a function so at the end it have to be execute, so we have to put the parenthesis.

Now any parent component that house our component it now able to bind and use our property. Example in parent temple component

<div class="col-xs-12">

<app-server-element

\*ngFor="let serverElement of serverElements"

[element]="serverElement">

</app-server-element>

</div>

**Assigning an Alias to Custom Properties**

If you don’t want that your custom property have the same name than in the backend, you can put a parameter to @Input decorator and that will be the name of the custom property. Example:

In the child element:

**export** **class** ServerElementComponent **implements** OnInit {

@Input('SvrElement') element: {type: string, name: string, content: string};

**constructor**() { }

ngOnInit(): void {

}

}

In the template of the component that use our element:

<div class="col-xs-12">

<app-server-element

\*ngFor="let serverElement of serverElements"

[SvrElement]="serverElement">

</app-server-element>

</div>

### @Output

With @Output we pass information at inverse direction, from a component to the parent, it there is a change in the child component I want send information to the parent component (hacia arriba). The parent component it is using in its template the child component.

In the child we have to emit the event and use @Output. Example:

**export** **class** CockpitComponent **implements** OnInit {

@Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();

@Output('bpCreated') blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();

newServerName = '';

newServerContent = '';

onAddServer() {

**this**.serverCreated.emit({

serverName: **this**.newServerName,

serverContent: **this**.newServerContent

});

}

}

In the parent to be used in the template:

<div class="container">

<app-cockpit

(serverCreated)="onServerAdded($event)"

(bpCreated)="onBlueprintAdded($event)"

></app-cockpit>

</div>

In the parent in the TS:

**export** **class** AppComponent {

onServerAdded(serverData: {serverName: string, serverContent: string}) {

}

Also just like @Imput, here we can create and use alias, has can we watch in the child component the alias is bpCreated for blueprintCreated output.

Note: It seems that Output only can send events to his father component, the component that house our component ?

**Understanding View Encapsulation**

Angular force to only apply our css styles to our component, encapsulate the styles for the component, this is not the default behavior of the DOM. If you put a style in a component parent that style doesn’t apply for child components.   
Angular do a king of emulation of the Shadow DOM. the Shadow DOM it is a technology not supported for all the browsers.

However, there is an option that let your component to not be encapsulated and apply CSS changes to all application, this can be done in the component definition using the reserved word encapsulation with none view encapsulation. Example:

@Component({

selector: 'app-server-element',

templateUrl: './server-element.component.html',

styleUrls: ['./server-element.component.css'],

encapsulation: ViewEncapsulation.None

})

Now, any change done in the CSS of this component will be applied to the all application.

### **Local References in Templates**

In some occasions we don’t want a double binding (template with the TS), we only want get the value of an HTML element and send it to the TC code, this can be done with **#reference\_name**. Example:

In template:

<input

type="text"

class="form-control"

#serverNameInput>

<button

class="btn btn-primary"

(click)="onAddServer(serverNameInput)">Add Server</button>

In TS code:

onAddServer(nameInput: HTMLInputElement) {

**this**.serverCreated.emit({

serverName: nameInput.value,

serverContent: **this**.newServerContent

});

}

* A local reference can be associated to any HTML element.
* A local reference hold the value of the all html element with all its properties.
* Can be used anywhere in the template, but only in the template, not in the TC code.

### @ViewChild

In some occasions we want access to an element in the DOM before give click on a button. We can in the TC access to an HTML element using @ViewChild.

The @ViewChild needs a parameter that it is a selector of the HTML element, passing here a local reference to the element will be enough.

Example in the template:

<input

type="text"

class="form-control"

#serverContentInput>

Example in the TC code:

@ViewChild('serverContentInput') serverContentInput: ElementRef;

onAddServer(nameInput: HTMLInputElement) {

**this**.serverCreated.emit({

serverName: nameInput.value,

serverContent: **this**.serverContentInput.nativeElement.value

});

}

Note: the @ViewChild decorator it is not recommendable used to modify the DOM, only to access the DOM an retrieve information. For modify the DOM we have the Angular property binding or string expressions.

### Ng-content

This is another way to pass information between components. In this case we put code inside a component in the template. Sometimes you have complex angular code that you want to pass into another component.

Template of component app-server-element:

<div class="panel panel-default">

<div class="panel-heading">{{ element.name }}</div>

<div class="panel-body">

<ng-content></ng-content>

</div>

</div>

Here we put the directive <ng-content> whish serve has a hook, to mark where the code to pass to this directive between its tags should be added.

Template of component app-component, where the app-server-element directive is used:

<div class="col-xs-12">

<app-server-element

\*ngFor="let serverElement of serverElements"

[SvrElement]="serverElement">

<p>

<strong \*ngIf="serverElement.type === 'server'" style="color: red">{{ serverElement.content }}</strong>

<em \*ngIf="serverElement.type === 'blueprint'">{{ serverElement.content }}</em>

</p>

</app-server-element>

</div>

Now, the code beetween the <app-server-element> directive it’s projected into the app-server-element, exactly where the <ng-content> is.

### @ContentChild

If we want access to an HTML element of a template that it is passed throughout and ng-content, how the view doesn’t have the html element, we can’t not use @ViewChild, but instead, we could use @ContentChild, because this HTML element don’t belong to the view but it’s part of the content. E.g.

TS file of the app-server-element:

@ContentChild('contentParagraph') paragraph: ElementRef;

Template of component app-server-element:

<div class="panel panel-default">

<div class="panel-heading">{{ element.name }}</div>

<div class="panel-body">

<ng-content></ng-content>

</div>

</div>

Template of component app-component, where the app-server-element directive is used:

<div class="col-xs-12">

<app-server-element

\*ngFor="let serverElement of serverElements"

[SvrElement]="serverElement">

<p #contentParagraph>

<strong \*ngIf="serverElement.type === 'server'" style="color: red">{{ serverElement.content }}</strong>

<em \*ngIf="serverElement.type === 'blueprint'">{{ serverElement.content }}</em>

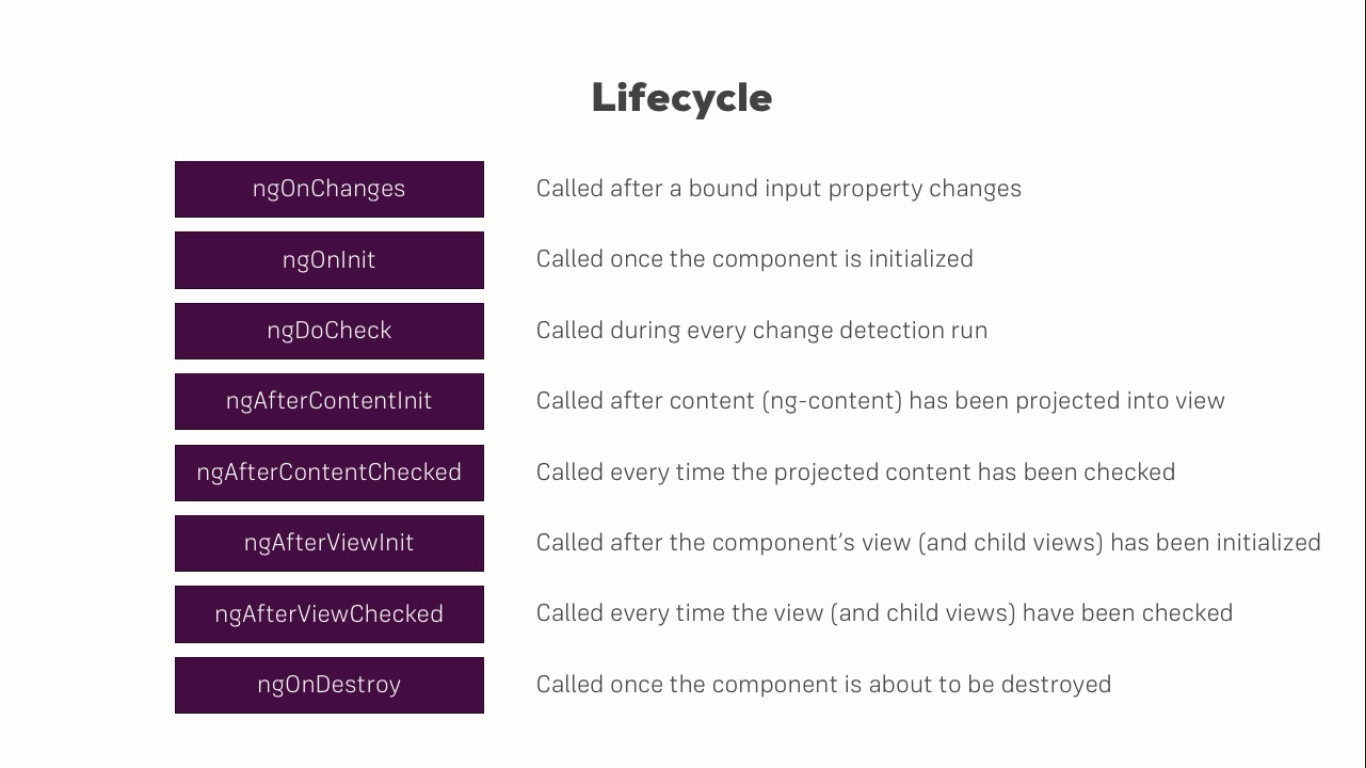
</p>

</app-server-element>

</div>

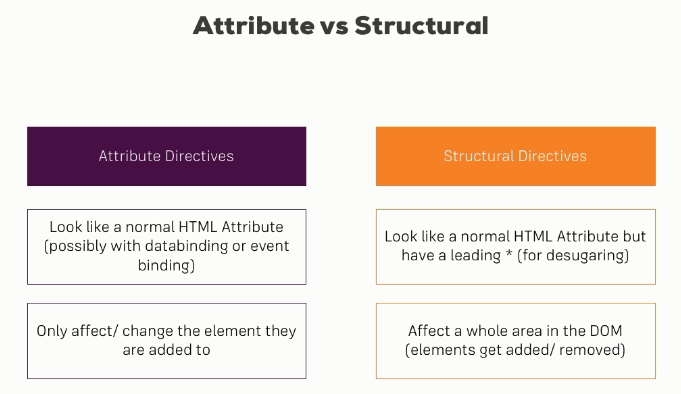
### Angular lifecycle

This is the phases, the order of events in an angular component.



* **Constructor**: it is the first that it’s called.
* **ngOnChanges(changes: SimpleChanges)**: it is called jus after constructor and every time the properties inside the component, or at list @input properties are changed. It’s the only event of the cycle that have parameter.
* **ngOnInit**: every time a component e.g. <app-server-element> its draw in the HTML the methods Contructor() and ngOnInit() are called, in this order. The ngOnInit it’s called before the component it is draw. Initializations of class variables should be done here and not in the constructor.
* **ngDoCheck**: it’s called by angular anytime it have reload de component or on a promise, event, etc. How this is called a lot of time, its’ not recommendable put here a big amount of code.
* **ngAfterContentInit**: it’s called after the component its draw in the view.
* **ngAfterContentChecked**: called always after **ngDoCheck** method because it’s when the content has been changed.
* **ngAfterViewInit**: called every time the component’s view (and child views) have been initialized.
* **ngAfterVieChecked**: called every time the view (and child views) have been cheched.
* **ngOnDestroy**: called when the element it’s deleted from the DOM.

# Section 7: Directives Deep Dive



## Creating a Basic Attribute Directive

In Angular we can create our own directives. For that we have to:

* Execute **ng g d directive-name** to create directive. This create a TS file to store the directive and mark the file as a directive with the @Directive decorator.
* In the constructor pass an **ElementRef** to the object.
* In the constructor also pass a **Render2** object, it’s a good idea use it to access to the DOM, not do it directly.
* In an ngOnInit () method do what you need.

Directive:

**import** { Directive, OnInit, ElementRef, Renderer2 } **from** '@angular/core';

@Directive({

selector: '[appBetterHightlight]'

})

**export** **class** BetterHightlightDirective **implements** OnInit {

**constructor**(**private** elRef: ElementRef, **private** renderer: Renderer2) { }

ngOnInit() {

**this**.renderer.setStyle(**this**.elRef.nativeElement, 'background-color', 'blue');

}

}

Usage in other component:

<div class="container">

<p appBetterHightlight>Style me with the better directive created!!!</p>

</div>

Note: The [] in the @Directive decorator are telling to Angular that his directive going to behave as attribute of an HTML element, that no scare its required when it will used.

Something important its that the directive created should be imported in our declarations section for app.module.ts file. E.g

@NgModule({

declarations: [

AppComponent,

BetterHightlightDirective

]

**More about the Renderer**

In the last lecture, we used the Angular Renderer class to change the style of a HTML element. As explained in that lecture, you should use the Renderer for any DOM manipulations.

Of course, you can do more than simply change the styling of an element via setStyle(). Learn more about the available Renderer methods [here](https://angular.io/api/core/Renderer2).

## @HostListener

The decorator @HostListener is just a convenient way of listening to events on that element. With that event listener for our directive we can change the styles on events. E.g. here we put a background blue when the mouse is over the element an transparent again when the mouse leave.

**import** { Directive, OnInit, ElementRef, Renderer2, HostListener } **from** '@angular/core';

@Directive({

selector: '[appBetterHightlight]'

})

**export** **class** BetterHightlightDirective **implements** OnInit {

**constructor**(**private** elRef: ElementRef, **private** renderer: Renderer2) { }

@HostListener('mouseenter') mouseenter (eventData: Event) {

**this**.renderer.setStyle(**this**.elRef.nativeElement, 'background-color', 'blue');

}

@HostListener('mouseleave') mouseleave (eventData: Event) {

**this**.renderer.setStyle(**this**.elRef.nativeElement, 'background-color', 'transparent');

}

## @HostBinding

This is another decorator that let us do the same that Render2, this is another way to do it. So here we access the style property and then there a sub-property, the backgroundColor and we set this equal to whatever background color we want. Camel case is important here because we're accessing the DOM property which doesn't know dashes.

**import** { Directive, OnInit, ElementRef, Renderer2, HostListener, HostBinding } **from** '@angular/core';

@Directive({

selector: '[appBetterHightlight]'

})

**export** **class** BetterHightlightDirective **implements** OnInit {

@HostBinding('style.backgroundColor') backgroundColor: string = 'transparent';

**constructor**(**private** elRef: ElementRef, **private** renderer: Renderer2) { }

@HostListener('mouseenter') mouseenter (eventData: Event) {

**this**.backgroundColor = 'blue';

}

@HostListener('mouseleave') mouseleave(eventData: Event) {

**this**.backgroundColor = 'transparent';

}

}

## Passing parameters to a directive (Binding to Directive Properties)

Using @Input() we can pass parameters to our directive, this help us to have a more configurable directive, that where it is used we can send the color we want, for example. E.g:

Our directive:

@Directive({

selector: '[appBetterHightlight]'

})

**export** **class** BetterHightlightDirective **implements** OnInit {

@Input() defaultColor: string = 'transparent';

@Input() highlightColor: string = 'blue';

@HostBinding('style.backgroundColor') backgroundColor: string;

@HostListener('mouseenter') mouseenter (eventData: Event) {

**this**.backgroundColor = **this**.highlightColor;

}

@HostListener('mouseleave') mouseleave(eventData: Event) {

**this**.backgroundColor = **this**.defaultColor;

}

}

How it used in another component:

<div class="container">

<p appBetterHightlight [defaultColor]="'yellow'" [highlightColor]="'red'">Style me with the better directive created!!!</p>

</div>

Note: if we pass after our directive some properties binding angular infers that the binding it relative with the directive and used them like parameters.

Other note: for property binding in general we use to put the binding property between [], and the element we pass between ‘’ if it is an string, but we have an exception just for string, we could use:

highlightColor="red"

## Building a Structural Directive

We have to create an @input element but also use the set reserved word to create a method associated to this property, a method that will be executed every time the property change, or when the parameter we pass to the directive change. Also the name of the @input should be the same that the directive name.  
The parameter passed to the constructor mark the place where angular going to build the directive.

e.g. In this example I have created a property that that it is the opposite of \*ngIf, it something its false show somenting.

Directive:

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

@Directive({

selector: '[appUnless]'

})

**export** **class** UnlessDirective {

@Input() **set** appUnless(condition: boolean) {

**if** (!condition) { // what we want

**this**.vcRef.createEmbeddedView(**this**.templateRef);

} **else** { // dont show nothing

**this**.vcRef.clear();

}

}

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

}

Usage in other component (onlyOdd it’s a boolean value):

<div \*appUnless="onlyOdd">

<p>Only odd</p>

</div>

## ngSwitch

This angular directive function just like a switch of any language. E.g.

In tS:

value = 2;

in template:

<div [ngSwitch]="value">

<p \*ngSwitchCase="5"> Value is 5</p>

<p \*ngSwitchCase="10"> Value is 10</p>

<p \*ngSwitchCase="100"> Value is 100</p>

<p \*ngSwitchDefault> Value is Default</p>

</div>

# Section 9: Using Services & Dependency Injection

Angular dependency injection work similar than in Spring framework, Angular its responsible for inject and create for us a new object we need, in this case a service. We have to fallow some steps to inject a service in our component:

1. Import our service class into our component.
2. Provide a decorator called **providers** inside our @Component creation to inject in an array our services.
3. In our constructor declare a variable of the service type

E.g. service:

**export** **class** LoggingService {

logStatusChange(status: string) {

console.log('A server status changed, new status: ' + status);

}

}

E.g. of usage:

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

**import** { LoggingService } **from** '../logging.service';

@Component({

selector: 'app-account',

templateUrl: './account.component.html',

styleUrls: ['./account.component.css'],

providers: [LoggingService]

})

**export** **class** AccountComponent {

**constructor**(**private** loggingService: LoggingService) { }

onSetTo(status: string) {

**this**.loggingService.logStatusChange(status);

}

}

**How to create a service**

A service can be created just like a class and there is no need to be included in the app.module.ts, however, there is commando to do that: ng g service service\_name

## Angular dependency injector

The Angular dependency injector actually is a hierarchical injector, that means that if we provide a service in some place of our app, let's say on one component, the Angular framework knows how to create an instance of that service for this component and important, all its child components and actually this component and all its child components and the child components of the child components will receive the same instance of the service.



The propagation will always be for downstairs and not upstairs.

If we inject the service in the appModule the service will be available for all components and services with the same service name.

If we add to the appComponent the service will be available for all components but not services with the same service name.

There is no need to add to a child the service we add to the parent (no need to add it in the providers decorator, but you have to put in the constructor). Anyway it you need use another instance of the service in the child components you can do id but you will be overriding the parent service and creating a new instance of the service.

## Injecting Services into Services

To do that we need to inject our both services in the module where you going to use them, after that we need to add to the service where we going to inject the other service the @Injectable metadata decorator. In Angular 9 it is recommendable always in services use @Injectable decorator, and also when you created with commands he do it for you.

App.module.ts:

@NgModule({

declarations: [

AppComponent,

AccountComponent,

NewAccountComponent

],

imports: [

BrowserModule,

FormsModule,

],

providers: [AccountService, LoggingService],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

Service LoggingService injected in AccountService:

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

**import** { LoggingService } **from** './logging.service';

@Injectable({

providedIn: 'root'

})

**export** **class** AccountService {

accounts = [

{

…

];

**constructor**(**private** loggingService: LoggingService) { }

addAccount(name: string, status: string) {

**this**.accounts.push({name: name, status: status});

**this**.loggingService.logStatusChange(status);

}

Note: If we include @Injectable({providedIn: 'root'}) decorator and include the provided in root this is an alternative instead of simply add it to providers in the app module. Here is also fine, this is a shortcut, a shorter and easier way.

## Services communication with EventEmmiter

We can establish communication between component using the services and the EventEmmiter core class. It seems there is a more efficient way but for now this tell the course.

For example I have a service used by two components, in the service we create an EventEmmiter object, and in one component we emit the event and in the other we receive the event(suscription):

Service:

@Injectable({

providedIn: 'root'

})

**export** **class** AccountService {

statusUpdate = new EventEmitter<string>();

One component on the event of an account status changed:

@Component({

selector: 'app-account',

templateUrl: './account.component.html',

styleUrls: ['./account.component.css'],

})

**export** **class** AccountComponent {

@Input() account: { name: string, status: string };

@Input() id: number;

**constructor**(**private** loggingService: LoggingService,

**private** accountService: AccountService) { }

onSetTo(status: string) {

**this**.accountService.onStatusChanged(**this**.id, status);

// this is the even emitting

**this**.accountService.statusUpdate.emit(status);

}

}

Reception of the event in other component:

@Component({

selector: 'app-new-account',

templateUrl: './new-account.component.html',

styleUrls: ['./new-account.component.css'],

})

**export** **class** NewAccountComponent {

**constructor**(**private** loggingService: LoggingService,

**private** accountService: AccountService) {

**this**.accountService.statusUpdate.subscribe(

(status: string) **=>** alert('New status: ' + status)

);

}

}

## Services communication with Subject (Observable)

You could use EventEmmiter to communicate between components, but there is a better approach, a more recommended one and that new approach, the better approach uses a **Subject**. Now a subject is something we import from RxJS because it’s a king of observable. Now instead of event emitter, you now create a subject here. It's pretty similar though, it's a generic type where you define which data will eventually be emitted, in this case a boolean.

user.service.ts where exists the subject.

**import** { Injectable, EventEmitter } **from** '@angular/core';

**import** { Subject } **from** 'rxjs';

@Injectable({providedIn: 'root'})

**export** **class** UserService {

// activatedEmitter = new EventEmitter<boolean>();

activatedEmitter = new Subject<boolean>();

}

The way to emit the event is also very similar, in the user component we don't call emit, else you call ***next*** because a Subject is a special kind of Observable you could say.

user.component.ts (where we emit the event)

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

**import** { ActivatedRoute, Params } **from** '@angular/router';

**import** { UserService } **from** '../user.service';

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css']

})

**export** **class** UserComponent **implements** OnInit {

id: number;

**constructor**(**private** userService: UserService) {

}

onActivated() {

// this.userService.activatedEmitter.emit(true);

**this**.userService.activatedEmitter.next(**true**);

}

}

app.component.ts (where we receive the event)

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

**import** { UserService } **from** './user.service';

**import** { Subscription } **from** 'rxjs';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

**export** **class** AppComponent **implements** OnInit, OnDestroy {

userActivates = **false**;

**private** activatedSub: Subscription;

**constructor**(**private** userService: UserService) { }

ngOnInit() {

**this**.activatedSub = **this**.userService.activatedEmitter.subscribe(

(didActivated: boolean) **=>** {

**this**.userActivates = didActivated;

}

);

}

ngOnDestroy() {

**this**.activatedSub.unsubscribe();

}

}

So, don't use event emitter, use subjects they are in the end a bit more efficient behind the scenes, you can also now use all these cool operators because a subject in the end also is kind of an observable. One important note, just as with your own observables, you should unsubscribe to your subjects though whenever you don't need them.

Now one important note about subjects as a replacement for event emitters, this only counts if you're using them as cross component event emitters, where you manually call next or previously emit.

You don't use subjects instead of event emitter when you're using @output.

So in a component if you're using @output here with your own event, you still use the Angular event emitter, you're not using subject there because the subject is not suitable for that, there you need the Angular event emitter, you only use subjects to communicate across components, through services so through a mechanism where you in the end subscribe to somewhere, like here in the app component.

If you're not subscribing to an event emitter, then it probably is an output, if you do plan to subscribe manually, then it is a subject.

More about this in: Subjects vs EvenEmmiter.

# Section 11: Changing Pages with Routing

Setting up and Loading Routes

The place to configure our routing it is: app.module.ts file. For configure in our app the routing we have to fallow this steps:

1. Add a constant variable of the type: Routes that will be an array. This array will have javascript objects with the specific nomenclature:

path: the name of the path in the url.

component: the component that should be loaded when the path be selected.

1. Register our Routes in the app in the import session with the forRoot method of the class RouterModule. E.g. RouterModule.forRoot(appRoutes)
2. Finally we have to inform Angular where the router will be draw. Where the component associated to every url will be draw. This is done writing the angular directive: <router-outlet></router-outlet>
3. We have to use in the navigation menu of the app the directive routerLink="/servers" that let us change our page to the indicated without refresh the pages of our app. This directive can be used in two ways how we can see in the example.

E.g. of app.module.ts:

**import** { BrowserModule } **from** '@angular/platform-browser';

**import** { NgModule } **from** '@angular/core';

**import** { FormsModule } **from** '@angular/forms';

**import** { Routes, RouterModule } **from** '@angular/router';

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent},

{ path: 'users', component: UsersComponent},

{ path: 'servers', component: ServersComponent},

];

@NgModule({

declarations: [

AppComponent,

HomeComponent,

UsersComponent

],

imports: [

BrowserModule,

FormsModule,

RouterModule.forRoot(appRoutes)

],

providers: [ServersService],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

E.g. of app.component.html:

<div class="container">

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<ul class="nav nav-tabs">

<li role="presentation" class="active"><a routerLink="/">Home</a></li>

<li role="presentation"><a routerLink="/servers">Servers</a></li>

<li role="presentation"><a [routerLink]="['/users']">Users</a></li>

</ul>

</div>

</div>

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<router-outlet></router-outlet>

</div>

</div>

Note: the value of routeLink could appear with / or without it. If we use slash we are using a full path url, on the other hand if we don’t used it, we are using a relative path.

## Styling Active Router Links

Exists and angular directive called **routerLinkActive** that let us put a CSS class for the element selected, this directive can be put in the link or in the element that wrap our link (bootstrap only work if we put it on the wrap). Also exists the directive **routerLinkActiveOptions** that in this case say that if the route it is exactly equal (not partially) equal so the class its attached.

E.g.

<ul class="nav nav-tabs">

<li role="presentation" routerLinkActive="active" [routerLinkActiveOptions]="{exact: **true**}"   
 ><a routerLink="/">Home</a></li>

<li role="presentation" routerLinkActive="active">  
 <a routerLink="/servers">Servers</a></li>

<li role="presentation" routerLinkActive="active">  
 <a [routerLink]="['/users']">Users</a></li>

</ul>

Note: in this example the class active is the bootstrap class that give a different color to the selected item.

## Navigating Programmatically

We can also from our code after some event execute the navigation if we need it. To doid you just have to declare in the constructor a variable of type Route.

E.g. of click event over a button:

**export** **class** HomeComponent **implements** OnInit {

**constructor**(**private** router: Router) { }

onLoadServers() {

**this**.router.navigate(['/servers']);

}

}

Also the navigate method have a second argument that could be pass to the function, that is the relative argument, where you can say navigate to this route from this relative path.

If our URL also have query parameters, to avoid louse them, we can use the queryParamsHandling parameter. This parameter could have two possible values: merge or conserve. E.g.

onEdit() {

**this**.router.navigate(['edit'], {relativeTo: **this**.route, queryParamsHandling: 'preserve'});

}

## Passing Parameters to Routes

You can pass paramether to a url route using the colon (:) and some name for the parameter. E.g

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent},

{ path: 'users', component: UsersComponent},

{ path: 'users/:id', component: UserComponent}, //example

{ path: 'servers', component: ServersComponent},

];

Also we can retrieve the parameter passed using the class **ActivatedRoute** and accessing to the **snapshot.params** method. E.g.

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

**import** { ActivatedRoute } **from** '@angular/router';

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css']

})

**export** **class** UserComponent **implements** OnInit {

user: {id: number, name: string};

**constructor**( **private** route: ActivatedRoute) { }

ngOnInit() {

**this**.user = {

id: **this**.route.snapshot.params['id'],

name: **‘test’**

}

}

}

## Fetching Route Parameters Reactively

The way we retrieve our parameter in last session is ok, but if we from our page try to call our own page using the routeLink approach, we don’t going to see our data depending of the parameter be updated. A better approach to do that is use an Observable. Observables are asynchronous methods that will be executed when an asynchronous event occurred. In this case when the paramethers of the route change. E.g. in user.component.ts

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

**import** { ActivatedRoute, Params } **from** '@angular/router';

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css']

})

**export** **class** UserComponent **implements** OnInit {

user: {id: number, name: string};

**constructor**( **private** route: ActivatedRoute) { }

ngOnInit() {

**this**.user = {

id: **this**.route.snapshot.params['id'],

name: **this**.route.snapshot.params['name'],

}

**this**.route.params.subscribe(

(params: Params) **=>** {

**this**.user.id = params['id'];

**this**.user.name = params['name'];

}

)

}

}

E.g. in user.component.html

<p>User with ID: {{user.id}}</p>

<p>User name is: {{user.name}}</p>

<hr>

<a [routerLink]="['/users', 10, 'Anna']">Load Anna (10)</a>

In this case the observable won’t be execute whit ngOnInit, he just going to subscribe and after that any time the parameters change he going to change it in the template.

## Passing Query Parameters and Fragments

Querry paramethers are the ones that: ?key=value&key=value. And fragment are the ones that start with #fragmen\_name and point to a specific place in the page.  
We have two ways of do it.

**On a template**, the way to do it is for example:

app.module.ts:

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent},

{ path: 'users', component: UsersComponent},

{ path: 'users/:id/:name', component: UserComponent},

{ path: 'servers', component: ServersComponent},

{ path: 'servers/:id/edit', component: EditServerComponent}

];

Template of a component:

<a

[routerLink]="['/servers', 5, 'edit']"

[queryParams]="{allowEdit: '1'}"

[fragment]="'loading'"

href="#"

class="list-group-item"

\*ngFor="let server of servers">

{{ server.name }}

</a>

Finally: <http://localhost:4200/servers/5/edit?allowEdit=1#loading>

Here to the routerLink parameter we add another property of it that its queryParams. QueryParams and that's important is not a new directive, it's just another bindable property of the routerLink directive.

**On a TS file**, the way to do it is for example:

onLoadServer(id: number) {

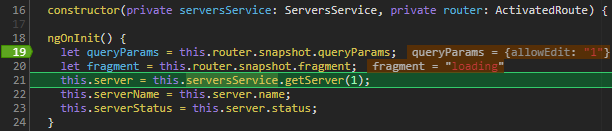
**this**.router.navigate(['/servers', id, 'edit'], {queryParams: {allowEdit: '1'}, fragment: 'loading'});

}

Finally: <http://localhost:4200/servers/1/edit?allowEdit=1#loading>

**Retrieving query parameters and fragments**

There are also two ways of retrieving it. For the first approach, we can simply access the snapshot of our route and access query params here or our fragment, just like that.



Now this might bring the same problem as with the params, this is only run or updated at the time this component is created. So if your query params are changed from the page you're currently on, you might not want to use this approach because it won't be reactive, it won't display or allow you to react to any changes which happen after this component has been loaded.

The alternative of course is to use the route and just like we had params as observable, we also have query params as an observable we can subscribe to, like this and of course we also have fragment as an observable we can subscribe to. So just like before, this will now allow you to react to changed query parameters.

ngOnInit() {

**let** queryParams = **this**.router.snapshot.queryParams;

**let** fragment = **this**.router.snapshot.fragment;

**this**.router.queryParams.subscribe(); //better way

**this**.router.fragment.subscribe(); //better way

**this**.server = **this**.serversService.getServer(1);

**this**.serverName = **this**.server.name;

**this**.serverStatus = **this**.server.status;

}

## Setting up Child (Nested) Routes

By default when we click all the component where we are change by the clicked, but we don’t want that. Sometimes we need to show our element when we click on a rote next to the element where we are. For that we need use nested routing.

1. We have to in out app.module.ts add the **children** parameter to the parent component, and inside him put its childs, now the urls of the childs.

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent},

{ path: 'servers', component: ServersComponent, children: [

{ path: ':id', component: ServerComponent},

{ path: ':id/edit', component: EditServerComponent}

]},

];

2. Now ServerComponent and EditServerComponent will be show inside of the ServersComponent. But for that we need to tell the childrens where they will be draw. For that we have to put the directive: <router-outlet></router-outlet> in the template of ServersComponent component.

## Redirecting and Wildcard Routes

We probably want than if something not existing in our routes its entered in the URL we redirect to default Page not found page.

A convenient way to catch all routes which are not covered by your app is to use the double asterisk route here, this is the wildcard route which means catch all paths you don't know. The order is super important here. Make sure that this very generic route is the last one in your array of routes because your routes get parsed from top to bottom here.

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent},

{ path: 'users', component: UsersComponent, children: [

{ path: ':id/:name', component: UserComponent}

]},

{ path: 'not-found', component: PageNotFoundComponent},

{ path: '\*\*', redirectTo: 'not-found', pathMatch: 'full' }

];

We have a component in our app for paged not found and we always redirect to him if something not existing in the routes are input.

## Module for Router

Sometimes if we going to have a lot of things in our Route, it’s a good practice to have our own module just for route. We simply want to use this app routing module to outsource our routes.

Therefore, we need to add our AppRoutingModule back to our main module and for this, we need to add the exports array here. Exports simply tells Angular, hey from this module, if I were to add this module to the imports of another module, what should be accessible to this module, which imports this module?

And the one thing we want to make accessible is our router module.

E.g. app-routing.module.ts

**import** { NgModule } **from** '@angular/core';

**import** { Routes, RouterModule } **from** '@angular/router';

**import** { HomeComponent } **from** './home/home.component';

**import** { UsersComponent } **from** './users/users.component';

**import** { ServersComponent } **from** './servers/servers.component';

**import** { UserComponent } **from** './users/user/user.component';

**import** { EditServerComponent } **from** './servers/edit-server/edit-server.component';

**import** { ServerComponent } **from** './servers/server/server.component';

**import** { PageNotFoundComponent } **from** './page-not-found/page-not-found.component';

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent},

{ path: 'users', component: UsersComponent, children: [

{ path: ':id/:name', component: UserComponent}

]},

{ path: 'servers', component: ServersComponent, children: [

{ path: ':id', component: ServerComponent},

{ path: ':id/edit', component: EditServerComponent}

]},

{ path: 'not-found', component: PageNotFoundComponent},

{ path: '\*\*', redirectTo: 'not-found', pathMatch: 'full'}

];

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

],

exports: [RouterModule]

})

**export** **class** AppRoutingModule {}

E.g. app.module.ts where we import the AppRoutingModule module

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

AppRoutingModule

],

providers: [ServersService],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

## Protecting Routes with **canActivate**

Sometimes we need to execute some code before a route is showed, or after, for example may be you need to know if a user have access to certain route, or if the user it is logged.

For that in our appRoutes we have to add the canActivate property to a route and pass to this property the service how going to return true or false about accessing the page. This service need to implement the CanActivate interface, that forces you to implement canActivate method.

app-routing.module.ts

**import** { AuthGuard } **from** './auth-guard.service';

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent, children: [

{ path: ':id/:name', component: UserComponent }

] },

{

path: 'servers',

canActivate: [AuthGuard], component: ServersComponent, children: [

{ path: ':id', component: ServerComponent },

{ path: ':id/edit', component: EditServerComponent }

] },

// { path: 'not-found', component: PageNotFoundComponent },

{ path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },

{ path: '\*\*', redirectTo: '/not-found' }

];

auth-guard.service.ts

**import** { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router, CanActivateChild } **from** '@angular/router';

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

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

**import** { AuthService } **from** './auth.service';

@Injectable()

**export** **class** AuthGuard **implements** CanActivate {

**constructor**(**private** authService: AuthService, **private** router: Router) {}

canActivate(route: ActivatedRouteSnapshot,

state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

**return** **this**.authService.isAuthenticated()

.then(

(authenticated: boolean) **=>** {

**if** (authenticated) {

**return** **true**;

} **else** {

**this**.router.navigate(['/']);

}

}

);

}

}

How we can see in this service we also call another service that return if the user its authenticated. Also, if not authenticated we send the navigation to home page (‘/’).

## Protecting Child (Nested) Routes with canActivateChild

The last canActivate example was working for our whole servers path here. Now we could grab it from here and add it to our child to make sure that only the child are protected, the children and not our root path but that is not the easiest way because if we add more child items, we have to add canActivate to each of them.

There is another guard we can use, it's pretty similar to canActivate, it's called CanActivateChild.

app-routing.module.ts

**import** { AuthGuard } **from** './auth-guard.service';

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent, children: [

{ path: ':id/:name', component: UserComponent }

] },

{

path: 'servers',

// canActivate: [AuthGuard],

canActivateChild: [AuthGuard],

component: ServersComponent,

children: [

{ path: ':id', component: ServerComponent },

{ path: ':id/edit', component: EditServerComponent }

] },

// { path: 'not-found', component: PageNotFoundComponent },

{ path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },

{ path: '\*\*', redirectTo: '/not-found' }

];

auth-guard.service.ts

**import** { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router, CanActivateChild } **from** '@angular/router';

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

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

**import** { AuthService } **from** './auth.service';

@Injectable()

**export** **class** AuthGuard **implements** CanActivate, CanActivateChild {

**constructor**(**private** authService: AuthService, **private** router: Router) {}

canActivate(route: ActivatedRouteSnapshot,

state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

**return** **this**.authService.isAuthenticated()

.then(

(authenticated: boolean) **=>** {

**if** (authenticated) {

**return** **true**;

} **else** {

**this**.router.navigate(['/']);

}

}

);

}

canActivateChild(route: ActivatedRouteSnapshot,

state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

**return** **this**.canActivate(route, state);

}

}

## Controlling Navigation with canDeactivate

We can execute certain logic before leave our route, this can be done with canDeactivate property. We have to fallow some steps:

1. Create a service that implement the Route class CanDeactivate, also here create an interface to be passed to the class. The service name in this case is: can-deactivate-guard.service.ts

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

**import** { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } **from** '@angular/router';

**export** **interface** CanComponentDeactivate {

canDeactivate: () **=>** Observable<boolean> | Promise<boolean> | boolean;

}

**export** **class** CanDeactivateGuard **implements** CanDeactivate<CanComponentDeactivate> {

canDeactivate(component: CanComponentDeactivate,

currentRoute: ActivatedRouteSnapshot,

currentState: RouterStateSnapshot,

nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

**return** component.canDeactivate();

}

}

2. In our app.routing.module.ts add the parameter canDeactivate to the route path where we want the execution. We have to pass to this parameter the CanDeactivateGuard class created.

**import** { CanDeactivateGuard } **from** './servers/edit-server/can-deactivate-guard.service';

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent, children: [

{ path: ':id/:name', component: UserComponent }

] },

{

path: 'servers', canActivateChild: [AuthGuard], component: ServersComponent,

children: [

{ path: ':id', component: ServerComponent, resolve: {server: ServerResolver} },

{ path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }

] },

{ path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },

{ path: '\*\*', redirectTo: '/not-found' }

];

3. in the TS on the module showed by the route path, in this case the component EditServerComponent , we implement in the class the interface (CanComponentDeactivate) created in the service. This interface forces you to develop the method canDeactivate . Where finally we going to put the code we cant to execute when before the user leave the path. In this case we check if there is any change in the page and show a message.

**export** **class** EditServerComponent **implements** OnInit, CanComponentDeactivate {

server: {id: number, name: string, status: string};

serverName = '';

serverStatus = '';

allowEdit = **false**;

changesSaved = **false**;

**constructor**(**private** serversService: ServersService,

**private** route: ActivatedRoute,

**private** router: Router) {

}

canDeactivate(): Observable<boolean> | Promise<boolean> | boolean {

**if** (!**this**.allowEdit) {

**return** **true**;

}

**if** ((**this**.serverName !== **this**.server.name || **this**.serverStatus !== **this**.server.status) && !**this**.changesSaved) {

**return** confirm('Do you want to discard the changes?');

} **else** {

**return** **true**;

}

}

}

## Passing Static Data to a Route

In our appRoutes exists a parameter or property that let us send static data to the page, this is **data** property. The data property allows us to pass an object here and in this object, we can define any key-value pairs. Any properties we want, like for example our message which is Page Not Found.

In our app.routing.module.ts

**import** { CanDeactivateGuard } **from** './servers/edit-server/can-deactivate-guard.service';

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent, children: [

{ path: ':id/:name', component: UserComponent }

] },

{ path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },

{ path: '\*\*', redirectTo: '/not-found' }

];

In the TS file of ErrorPageComponent component we can use snapshot to retrieve the data or just like with params or query params, if this could possibly change while you still are on this page, you want to use your route and here, the data observable to which you subscribe and which will give you this new data object here of type Data. Both way are correct, just depend on if the parameter going to change.

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

**import** { ActivatedRoute, Data } **from** '@angular/router';

@Component({

selector: 'app-error-page',

templateUrl: './error-page.component.html',

styleUrls: ['./error-page.component.css']

})

**export** **class** ErrorPageComponent **implements** OnInit {

errorMessage: string;

**constructor**(**private** route: ActivatedRoute) { }

ngOnInit() {

**this**.errorMessage = **this**.route.snapshot.data['message'];

**this**.route.data.subscribe(

(data: Data) **=>** {

**this**.errorMessage = data['message'];

}

);

}

}

## Resolving Dynamic Data with the resolve Guard

The resolver is a service that will always render the component in the end but it will do some pre-loading, you could say, it will fetch some data the component will then need later on. The resolver will technically be a service.

The resolve service should implement the angular interface Resolve, and to Resolve interface we have to pass a type, the type of the data we going to resolve at the end, in this case it’s a Server with its id, name and status.

server-resolver.service.ts

**import** { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot } **from** '@angular/router';

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

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

**import** { ServersService } **from** '../servers.service';

**interface** Server {

id: number;

name: string;

status: string;

}

@Injectable()

**export** **class** ServerResolver **implements** Resolve<Server> {

**constructor**(**private** serversService: ServersService) {}

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Server> | Promise<Server> | Server {

**return** **this**.serversService.getServer(+route.params['id']);

}

}

Note: as we can see our resolve method going to retrieve and observable with a server, a promise with a server or just a server. In this class an interface have been locally created only to be used as a type in this class. Also this class going to use the service ServersService who going to permit retrieve the server we want to show, because this component receive a parameter with the id of a server (route.params['id']) and them the component show details of his server.

app-routing.module.ts

In this file we now cand add the property resolve: {server: ServerResolver}, resolve going to have key-value pairs of the resolvers we want to use, the key value “server” is totally up to you.

**const** appRoutes: Routes = [

{ path: '', component: HomeComponent },

{

path: 'servers', component: ServersComponent, children: [

{ path: ':id', component: ServerComponent, resolve: {server: ServerResolver} },

{ path: ':id/edit', component: EditServerComponent}

] },

{ path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },

{ path: '\*\*', redirectTo: '/not-found' }

];

server.component.ts

Now, in our ServerComponent in the ngOnInit() method, before the component be loaded , we going to subscribe to the **data** observable in the route, because it’s there were the result of the ServerResolver going to travel. And them copy to this.server the selected server to be showed by the component.

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

**import** { ActivatedRoute, Params, Router, Data } **from** '@angular/router';

**import** { ServersService } **from** '../servers.service';

@Component({

selector: 'app-server',

templateUrl: './server.component.html',

styleUrls: ['./server.component.css']

})

**export** **class** ServerComponent **implements** OnInit {

server: {id: number, name: string, status: string};

**constructor**(**private** serversService: ServersService,

**private** route: ActivatedRoute,

**private** router: Router) {

}

ngOnInit() {

**this**.route.data

.subscribe(

(data: Data) **=>** {

**this**.server = data['server'];

}

);

}

## Understanding Location Strategies (url)

The pretty URL we have with router may will not be underhanded when we have our real production server, we now use a special server for development. in the web, this might not work out of the box because there, routes, the URL is always (this is how the web works) parsed handled by the server first, so by that server which hosts your application. To solve this the server hosting your Angular single page application has to be configured such that in a case of a 404 error, **it returns the index.html** file, so the file starting and containing your Angular app. Why? Because as I mentioned, all your URLs are parsed by the server first, so not by Angular, by the server.

The about is the best solution, however exists another solution using the #, used more for older browsers. Let's see how we enable it in Angular:

In the file app-routing.module.ts. You can enable it in your app-routing.module where you register your routes, here with the *forRoot* method. You can pass a second argument, a Javascript object to this method configuring the set up of the routes.

@NgModule({

imports: [

RouterModule.forRoot(appRoutes, {useHash: **true**})

],

exports: [RouterModule]

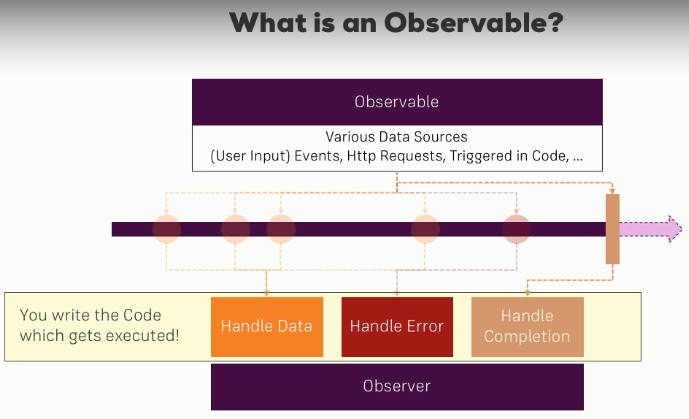
})

Now our URL look like: <http://localhost:4200/#/servers> . What this hashtag will do is, it informs your web server, hey only care about the part in this URL before this hashtag, so all the parts thereafter will be ignored by your web server.

Therefore this will run even on servers which don't return the index.html file in case of 404 errors because they will only care about the part in front of the hashtag.

That's how it works by default and the part after the hashtag can now be parsed by your client, by Angular.

# Section 13: Observable



An observable basically can be thought of as a data source. Now in our Angular project, an observable basically just is an object we import from a third-party package, RxJS (<https://es.wikipedia.org/wiki/Redux_(JavaScript)>). The observable here is implemented in a way that it follows the observable pattern, so we have an observable and we have an observer. In between, we have a stream, a timeline and on this timeline, we can have multiple events emitted by the observable or data packages you could say emitted by the observable, depending on the data source of that observable of course.

So the observable could emit data because you trigger it to do so, you can do that programmatically, it could be connected to a button and therefore whenever the button is clicked, an event in a data package is emitted automatically or as the Angular HTTP service does it, it's connected to a HTTP request.

So when the response returns, the response is emitted as a data package and there are dozens of other data sources too, we will have a look on where to find more soon.

So as I said, the other part is the observer, this actually is your code you could say. It's the subscribe function you saw earlier. There, you have three ways of handling data packages - you can handle the normal data, you can handle errors or you can handle the completion of the observable because these are the three types of data packages you can receive and in these hooks, in these boxes you could say, your code gets executed.

So you can determine what should happen if I receive a new data package, what should happen if I receive an error, what should happen when the observable eventually completes?

Side note, an observable doesn't have to complete, there are observables, for example hooked up to a normal button, which never complete because how would you know when it completes?

A user could click the button how often the user wants right. Other observables, like the HTTP request on the other hand, will have a clear end and will complete eventually because once the response is there, what else should happen, it's done.

So this is how the observable pattern generally works and of course you use it **to handle asynchronous tasks** because all these data sources here, user events triggered in your code or a HTTP request are asynchronous tasks, you don't know when they will happen and you don't know how long they will take. So if you execute your normal application code, you don't want to wait for these events or you don't want to wait for the completion of such a HTTP request because that would block your program, would block your logic.

Therefore, we need methods of handling such asynchronous tasks and historically you might have used callbacks or promises and it's not necessarily bad to use them, observables is just a different approach of handling that, a different alternative and Angular embraces observables which is why I chose to explain these because Angular uses them a lot and actually, observables have one major advantage, their operators, which I will show later in the section too.

## Creating an Oservable

1. First we can simply import observable, the type itself from RxJS.
2. Them we create a variable (customIntervalObservable) an assign *Oservable.crete()* or*new Oservable()*. It creates a new observable, an as a parameter takes a function and I'll pass in an anonymous arrow function here, like this which will get an argument automatically, RxJS will pass in that argument for us and that argument is a so-called **observer**.

The observer parameter is the part that is interested in being informed about new data, about errors or about the observable being completed. Now our job here is to tell the observer about new data, about an error or about the observable being completed. Here, we're not responsible for listening because the observer is the listener.

To inform we use the next function to emit a new value and that is important. The observer has a couple of important methods, next is one of them, error is another of them, and complete. In this example we a pass a counter to *next*.

1. Also we have to subscribe to our customIntervalObservable observable created, and when we receive the data we going to do the logic we need. In this case just print the data received.
2. It is very important that we have unsubscribe to our observable to destroy him, on the other hand we could have trobles of memory links. For that to the class que implemente tht interface OnDestroy, how when the component be destroyed we unsubscribe for the observable with unsubscribe(); method. When we use an observable created by Angular, we don’t need to unsubscribe because Angular do it for us.

E.g.

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

**import** { interval, Subscription, Observable } **from** 'rxjs';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.css']

})

**export** **class** HomeComponent **implements** OnInit, OnDestroy {

**private** firstObsSubscription: Subscription;

ngOnInit() {

**const** customIntervalObservable = new Observable(

observer **=>** {

**let** count = 0;

setInterval(() **=>** {

observer.next(count);

count++;

}, 1000);

});

**this**.firstObsSubscription = customIntervalObservable.subscribe(

data **=>** {

console.log(data);

}

}

ngOnDestroy(): void {

**this**.firstObsSubscription.unsubscribe();

}

}

In this example when we clic on a link menu we start to count on show numbers in log console, and when we give click on another link them the count stop.

## Errors & Completion

Emitting new data is arguably the most important thing observables do and in I'd say 99% of all cases, when you subscribe, you'll pass that first argument where you are interested in the data you are getting or maybe it's 90% because there is one other important use case too, especially when we think about things like HTTP requests and that is error handling.

In our example we going to fake an error, we could for example check if count is greater than three, so if that occurs, we'll also use the error method to throw a new error here. So here, I'll create a new error object with observer.error(new Error('Count is greater 3!'));

In our subscribe method we also can pass a new argument called error, now we can react or do something when an error occur.

Whenever an observable throws an error, it cancels, it's done, it will not emit any other values, it dies so to say and therefore in that case, you also don't need to unsubscribe. E. g.

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

**import** { interval, Subscription, Observable } **from** 'rxjs';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.css']

})

**export** **class** HomeComponent **implements** OnInit, OnDestroy {

**private** firstObsSubscription: Subscription;

ngOnInit() {

**const** customIntervalObservable = new Observable (observer **=>** {

**let** count = 0;

setInterval(() **=>** {

observer.next(count);

**if** (count > 3) {

observer.error(new Error('Count is greater 3!'));

}

count++;

}, 1000);

});

**this**.firstObsSubscription = customIntervalObservable.subscribe(

data **=>** {

console.log(data);

}, error **=>** {

console.log(error);

alert(error.message);

});

}

For completion we also going to add a logic that when the counter rich 5 we throws a completion event, for that we use the instruction: observer.complete();

Because we complete the observable, and that is important to understand and to keep in mind, whenever an observable completes, it really is done, there are no other values emitted thereafter, which kind of makes sense because it completed.

Now if you want to react to that completion, you can add a third arguments to the subscribe method and that is your completion handler function. It's a function that gets no arguments because completing doesn't pass any arguments and it's simply a function where you can do some cleanup work or whatever you need to do. E.g.

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

**import** { interval, Subscription, Observable } **from** 'rxjs';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.css']

})

**export** **class** HomeComponent **implements** OnInit, OnDestroy {

**private** firstObsSubscription: Subscription;

ngOnInit() {

**const** customIntervalObservable = new Observable (observer **=>** {

**let** count = 0;

setInterval(() **=>** {

observer.next(count);

**if** (count === 5) {

observer.complete();

}

**if** (count > 3) {

observer.error(new Error('Count is greater 3!'));

}

count++;

}, 1000);

});

**this**.firstObsSubscription = customIntervalObservable.subscribe(

data **=>** {

console.log(data);

}, error **=>** {

console.log(error);

alert(error.message);

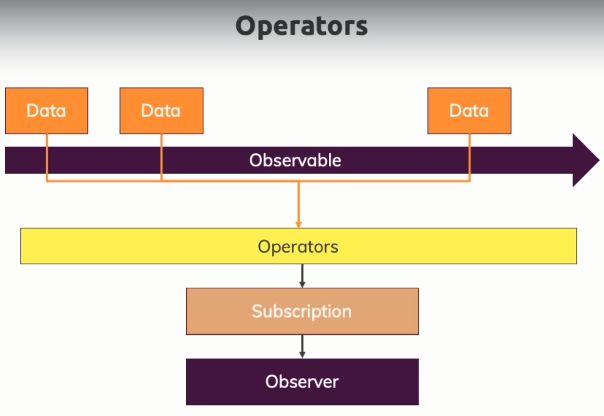
}, () **=>** {

console.log('Completed!');

});

}

## Operators



Operators are the magic feature of the RxJS library and they are the thing that really turn observable into awesome constructs.

If we have an observable and an observer, we of course get data and we listen to that with a subscription, that is what you learned. Now however, sometimes you don't need the raw data, you might want to transform it or filter out certain data points and of course, you could do all of that inside of your subscription or in the function you pass to your subscription but there's a more elegant way. Instead of setting up that subscription like this, you can use built-in operators in between.

That means that the data points first of all reach these operators that do something to the data and that something can be anything, there are tons of built-in operators and then you subscribe to the result of these operators.

**To create and operator:**

Operators can use on any observable, like our custom interval observable, by calling a method called *pipe*. E.g.

customIntervalObservable.pipe( map( (data: number) **=>** {

**return** 'Upper Round: ' + (data + 1);

}))

Every observable has a pipe method, and pipe method will receive an operator or all the operators you want. The pipe method is built into RxJS. We have to import the package: **import** { map } **from** 'rxjs/operators';

Here we use the map operator, but there are tons of built-in operators, like the map operator which is one of the operators you'll use more often. There are many, you can take a look at: <https://www.learnrxjs.io/learn-rxjs/operators>

E.g.

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

**import** { interval, Subscription, Observable } **from** 'rxjs';

**import** { map } **from** 'rxjs/operators';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.css']

})

**export** **class** HomeComponent **implements** OnInit, OnDestroy {

**private** firstObsSubscription: Subscription;

ngOnInit() {

**const** customIntervalObservable = new Observable (observer **=>** {

**let** count = 0;

setInterval(() **=>** {

observer.next(count);

**if** (count === 5) {

observer.complete();

}

**if** (count > 3) {

observer.error(new Error('Count is greater 3!'));

}

count++;

}, 1000);

});

**this**.firstObsSubscription = customIntervalObservable.pipe( map( (data: number) **=>** {

**return** 'Upper Round: ' + (data + 1);

})).subscribe(

data **=>** {

console.log(data);

}, error **=>** {

console.log(error);

alert(error.message);

}, () **=>** {

console.log('Completed!');

});

}

ngOnDestroy(): void {

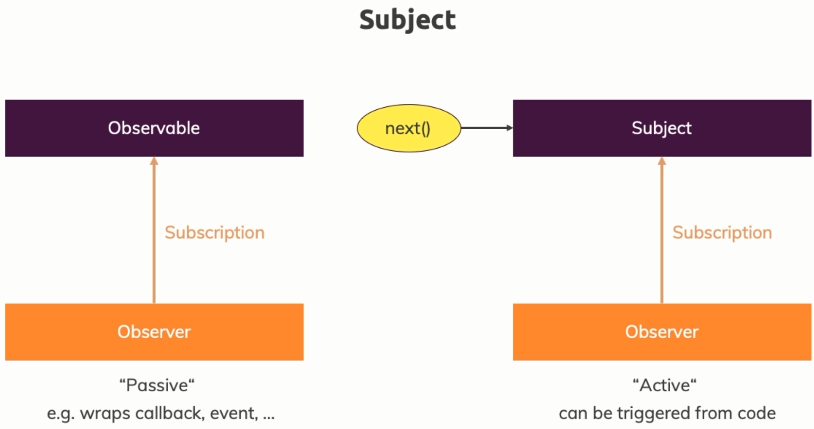
**this**.firstObsSubscription.unsubscribe();

}

}

Note: The map operator take a function as an argument, so an anonymous function here if you want. The argument that receive the function is the data send by the Observable object, and the data send by the operator is what we receive in our subscribed callback.

## Subjects vs EvenEmmiter



We know observables, we can subscribe to them as you learned but they're rather passive. You'll learn how to create your own observable but the core idea always is that you wrap a callback or an event or something like that. A **subject** is different, a subject also is an object you can subscribe to but it's more active because you can actively call next on it from outside.

Remember in the observable, we also called next but that was from inside the observable when we created it. So that is a more active observable that is perfect when we want to use it as an event emitter, so if we don't have a passive event source, like an HTTP request or DOM events but if we have something that actively needs to be triggered by us in our application and that's exactly the use case we have here. E.g.

See an example in: Services communication with Subject (Observable).

**Useful general Resources:**

* Official Docs: <https://rxjs-dev.firebaseapp.com/>
* RxJS Series: <https://academind.com/learn/javascript/understanding-rxjs/>
* Updating to RxJS 6: <https://academind.com/learn/javascript/rxjs-6-what-changed/>

# TypeScript

**What is TypeScript?**

TypeScript is a superset to JavaScript, which means that, it compiles into pure JavaScript in the end. Why do we use it then?

First, it provides ‘strong typing’ (that’s where the name comes from). This means that we can (and should) assign types to our variables and class members. These types won’t compile to JavaScript (as JS does not know types) but we will get compilation errors if we assign wrong types or make any other type-related errors. This is a HUGE help in the daily development work and should not be underestimated!

Second, TypeScript introduces some nice features, JS does not have out of the box (at least in the ES5 specification). This includes classes (‘class’ keyword), interfaces, generics and modules. Being able to use these constructs makes our code cleaner, easier to read and helps us avoid nasty errors. Especially in combination with the strong typing we are really able to write high quality code and track down errors quickly.

**Where can I learn all the TypeScript fundamentals?**

There are a lot of great resources out there which will get you started very quickly.

The official documentation is not too bad to be honest, so you may give it a try: <http://www.typescriptlang.org/Handbook>

There’s also a course here on Udemy, though I have not tested it! <https://www.udemy.com/typescript/>

**Can we mix TypeScript and JavaScript?**

Yes, we can. No one is preventing us from not setting types, using ‘var’ instead of ‘let’ or using pure JavaScript libraries (i.e. libraries which don’t offer a TypeScript version/ implementation).

**Can’t I use ‘normal’ JavaScript to write Angular 2 applications?**

You can absolutely do that. But currently finding good documentation and examples on Angular 2 using plain JavaScript is extremely hard. And to be honest: TypeScript will be the standard ‘language’ to be used when developing Angular 2 applications. So I definitely recommend using TypeScript.

## Installation an compilation

How install it:

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>

To use TS in command line you have to install it from npm:

npm install -g typescript

To compile a TS file with cd in the console put in the file to compile an use:

**tsc file\_name**

After that will be generated a file in javascritp e.g. file\_name.js

## Variable creation

// TypeScripts - well - strong typing allows us to define types for our variables and class members

// The compiler is going to yell at us if we assign a value of a wrong type to such a variable or member

// Declaring a variable with a type

// Using the 'let' keyword to create a variable ('const' would define an immutable constant)

**let** myString: string;

myString = 'This is a string';

// Try to assign a number to a string => Error

//myString = 4;

// TypeScript can also infer types

**let** anotherString = 'This is a string without :string'; // => Type 'string' was inferred from the assigned value

// This will still resolve in a compilation error

// anotherString = 4;

// TypeScript may only infer values when those values are assigned at the declaration

// This does not work:

**let** yetAnotherString;

yetAnotherString = 'This is a string';

// TypeScript does not know the type, therefore we don't get an error ... but no we're also ignoring TypeScripts strength: Typing

yetAnotherString = 5;

// Other basic types

**let** aString: string;

**let** aNumber: number;

**let** aBoolean: boolean;

**let** anArray: Array<string>; // This is a generic type => May only hold 'strings' in this case

**let** anything: any; // Any can be used if we don't know the actual type => Use it rarely!

// We also got void (=> nothing) and enums (a set of numeric values)

## Classes

They are preatty similar like in Java.

- You have cosntrucctor,

- Also you can create static methods.

- You have visibility for variables: private, public

Example:

// Classes allow us to create 'blueprints' for objects

// In Angular 2 we use classes a lot. For example to create Components, Services, Directives, Pipes, ...

// How to create a class

class Car {

engineName: string;

gears: number;

private speed: number;

constructor(speed: number) {

this.speed = speed || 0;

}

accelerate(): void {

this.speed++;

}

getSpeed():void {

console.log(this.speed);

}

static numberOfWheels(): number {

return 4;

}

}

// Instantiate (create) an object from a class

let car = new Car(5);

car.accelerate();

car.getSpeed();

console.log(Car.numberOfWheels());

**Constructor**

In TS the traditional way of declare a class and its constructor it is:

export class Ingrediente{

    public name: string;

    public amount: number;

    constructor(name: string, amount: number) {

        this.name = name;

        this.amount = amount;

    }

}

However there is a short way of declare the class with the attributes and it is:

export class Ingrediente{

    constructor(public name: string, public amount: number) {}

}

At the end TS compiler going to convert this second way to the first one.

## Interfaces

In the interface if we declare variables inside an interface, when we create an object of type interface, if we don't assig that values, we have an error. We can declare a variable inside of an interface also how optional whit the symbol ?

Example:

// Interfaces allow us to create contracts other classes/ objects have to implement

// We can use them to define custom types without creating classes

// Interfaces ARE NOT compiled to JavaScript! It's just for checking/ validation done by our TypeScript compiler

// Example interface

interface User {

username: string;

password: string;

confirmPassword?: string; // Optional property => Does not have to be implemented

}

let user:User;

// This value does not satisfy the interface => Compilation error

// user = { anything: 'anything', anynumber: 5};

// This value does satisfy the interface

user = {username: 'max', password: 'supersecret'};

// Interfaces can also contain functions (without the function body - as it only is a blueprint/ requirement)

interface CanDrive {

accelerate(speed:number): void;

}

let car:CanDrive = {

accelerate: function (speed:number) {

// ...

}

};

## Generics

// Generics are types which can hold/ use several types. We're only touching the very basics here - you can go MUCH more into detail

// Consider the Array object

**let** numberArray: Array<number>; // This array will only accept numbers

// Try to initialize it with strings

// numberArray = ['test']; // => Error

numberArray = [1,2,3];

## Modules

TypeScript is modular, we can divide our code up over several files. In Angular 2 we then use   
"import {} from ''" to access the code in these files

We export a class, interface, variable, ... by adding **'export'** keyword in front of it. That it’s the way of this class be available outside of this file for other files.

**export** **class** ExportedClass {

// This class is exported

}

# JavaScript

### Arrays

The arrays in javaScript function just like in Java, it’s consider and object and not a native type. So, if we assing an X variable that contain an array to a Y variable, X and Y are reference that point to the same object, son any change don to Y will be reflected in X variable.

### Array slice() Method

**Definition and Usage**

The slice() method returns the selected elements in an array, **as a new array object**. The slice() method selects the elements starting at the given start argument, and ends at, but does not include, the given end argument.

**Syntax**

*array*.slice(*start*, *end*)

**Parameter Values**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| *start* | Optional. An integer that specifies where to start the selection (The first element has an index of 0). Use negative numbers to select from the end of an array. If omitted, it acts like "0" |
| *end* | Optional. An integer that specifies where to end the selection. If omitted, all elements from the start position and to the end of the array will be selected. Use negative numbers to select from the end of an array |

<https://www.w3schools.com/jsref/jsref_slice_array.asp>

### Spread Operator (…)

**Spread operator** allows an iterable to expand in places where 0+ arguments are expected. It is mostly used in variable array where there is more than 1 values are expected.It allows us the privilege to obtain a list of parameters from an array. Syntax of Spread operator is same as [Rest parameter](https://www.geeksforgeeks.org/javascript-rest-operator/) but it works completely opposite of it.

**Syntax:**

var variablename1 = [**...**value];

E.g. In this method we are coping and array of Ingredient objects into another array of the same type. How push method can have various parameters to be pushed (*array*.push(*item1*, *item2*, ..., *itemX*)), so the spread operator convert the array to various objects to be pushed.

addIngredients(ingredientsP: Ingredient[]) {

**this**.ingredients.push(...ingredientsP);

**this**.ingredientChanged.emit(**this**.ingredients.slice());

}

### Cast to Number using the Unary (+) Operator

The Unary plus is the fastest way of converting something into a number, because it does not perform any other operations on the number. However there I think prefer use [Number()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) or other.

So, to convert a string or a Boolean, or a Numeric String, you can just use **+**

+false // 0  
+‘123’ // 123  
+0xBABE // 47806 (Hexadecimal)  
+null // 0  
+function(val) {return val } // NaN

The Unary (+) can convert string representations of **integers** and **floats**, as well as the non-string values **true**, **false**, and **null**. Integers in both **decimal** and **hexadecimal** formats are supported. **Negative** numbers are supported (though not for hex). If it cannot parse a particular value, it will evaluate to [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN).

However, bear in mind that the Unary (+) *does not* perform well in certain cases. For example, it doesn’t work as expected on empty strings, alphanumeric strings, empty objects etc.

+'' // NaN  
+'123a' // NaN  
+{} // NaN

There are of course other ways to cast to Number in Javascript, including but not limited to:

* [Number()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number)
* [parseInt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt)
* [parseFloat()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/parseFloat)
* [~~](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_NOT)
* [>>>0](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Unsigned_right_shift)

<https://medium.com/@nikjohn/cast-to-number-in-javascript-using-the-unary-operator-f4ca67c792ce>

# Various

## Sobre REACT

npm install → will get everything set up, and running

npm start → will let you run the application live.

React and Angular together

1. In this two pages are explained how integrate two applications of different frameworks through the NX framework.

<https://blog.nrwl.io/building-angular-and-react-applications-together-with-nx-78b5578de598>

<https://medium.com/angular-in-depth/how-to-talk-with-web-components-in-react-and-angular-8deb7d2fb92a>

2. In this other example, that seems to be what we need, the integration is done in a SPA using Micro Frontends integration.

<https://ivanjov.com/micro-frontends-how-i-built-a-spa-with-angular-and-react/>

Important notes:

package.json In this file are all dependencies need for React, Angular and other dependencies used to easy the integration and the communication.

root-application.js In this js, we're importing index files from react and angular directories and we initialize them on the page when the router hash starts with.

event-bus/index.js The communication between React and Angular apps can be tricky. It is done with Eev event bus librery. It's small, fast and zero-dependency event emitter that will help us to exchange information between React and Angular app.

react/index.js This is where the export of the React child app is done. In this code the React module single-spa-react is used to tell single-spa how to bootstrap, mount and unmount the React app.

angular/index.js In the folder angular it is where a component angular is created. In the file index.js , same as React, this code will tell single-spa how to bootstrap, mount and unmount the Angular app. The angular module single-spa-angular2 is used for that.

For execution: fto start the project first execute: npm install and then npm start. The url its: http://localhost:9090/#/

## Possible improvement for Angular 9

<https://profile.es/blog/angular9/>