# 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"

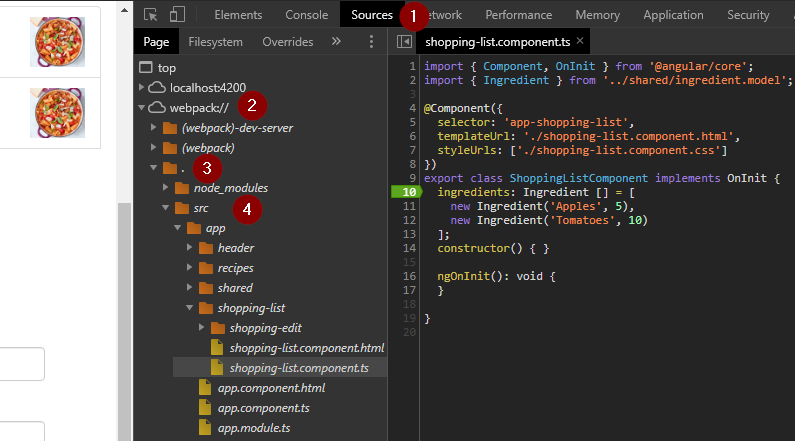
    [ngStyle]="{'background-color': getColor()}"

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

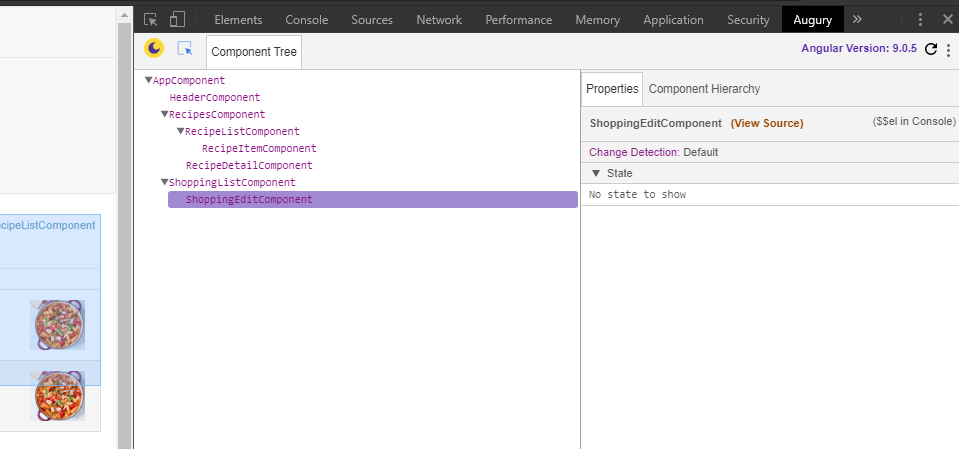
</div>

## Session 4

### How debug in Chrome

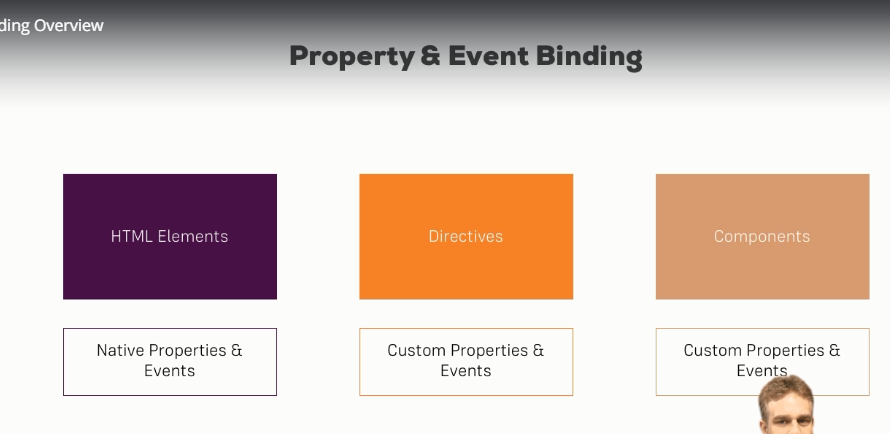


### Using Augury to Dive into Angular Apps



## 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>

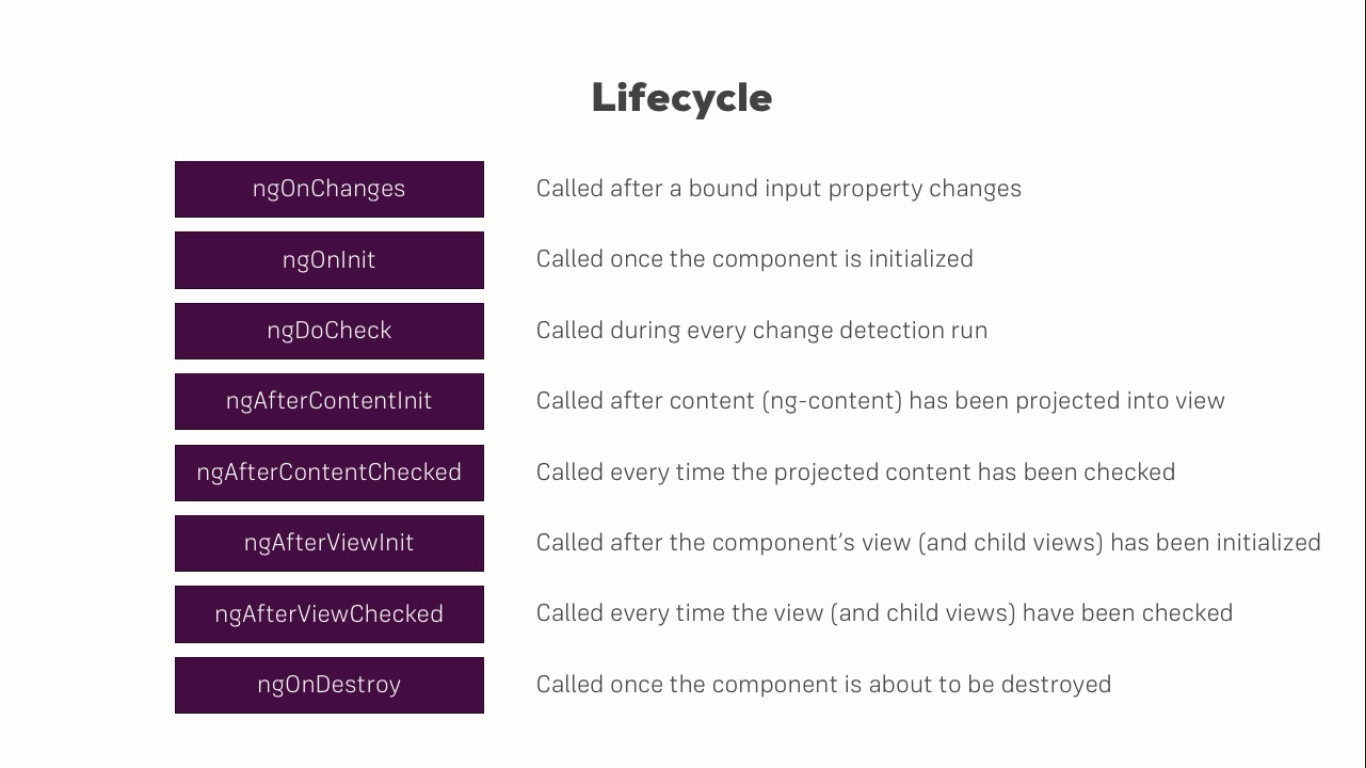
</p>

</app-server-element>

</div>

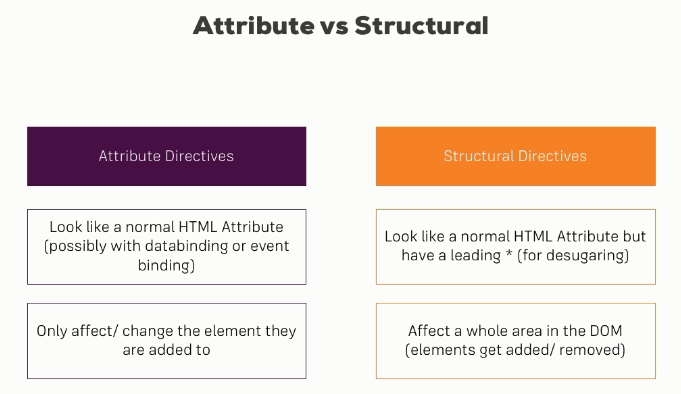
### Angular lifecycle

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, 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);

}

## 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)

);

}

}

# 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());

}

# 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/>