# 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 |
| **ng g p pipe\_name** | Command to generate a pipe in our project. It create a file called: pipe\_name.pipe.ts and a class inside with all we need to create our pipe. Also add the required import in the app.module.ts file. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**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.

De momento me he instalado estas dos, el primero es un paquete de varias, el segundo es unpluggin para usar las atajos de teclado de eclipse en VSC.

<https://johnpapa.net/rec-ng-extensions/>

<https://marketplace.visualstudio.com/items?itemName=alphabotsec.vscode-eclipse-keybindings>

TODO: take a look, and test

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

# 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 an angular variable inside double quotes “”. Example:

<button

    class="btn btn-primary"

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

Note: When we are binding a simple string to the property we have to put the single quotation marks. E.g.

[pattern]="'^[1-9]+[0-9]\*$'"

On

                    <input type="number" id="amount" class="form-control"

                    ngModel

                    name="amount"

                    required

                    pattern="^[1-9]+[0-9]\*$">

However since it's a string, we can also omit the single quotation marks and omit the square brackets to use this shortcut (atajo) of using property binding when binding to a string where we don't have to use the square brackets. E.g.

pattern="^[1-9]+[0-9]\*$"

**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"

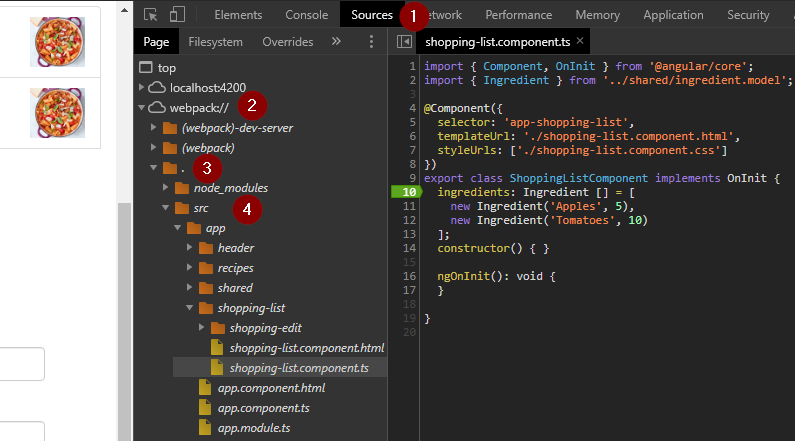
    [ngStyle]="{'background-color': getColor()}"

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

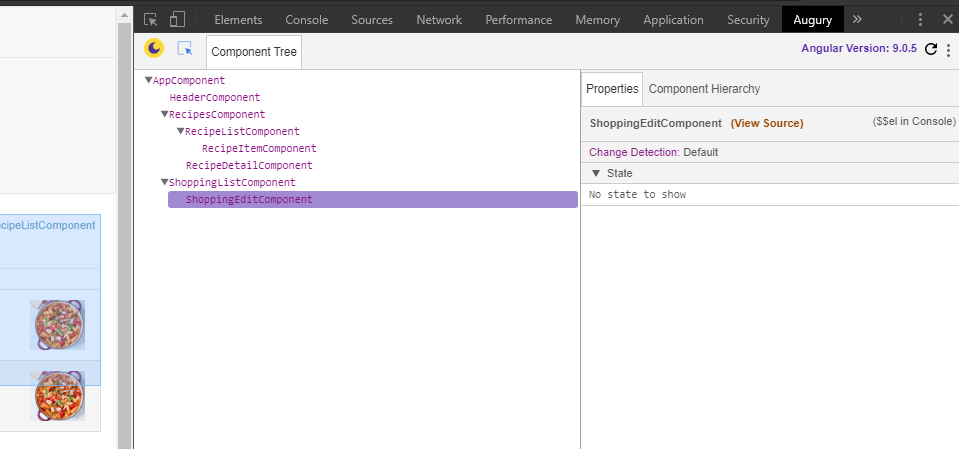
</div>

## Session 4

### How debug in Chrome

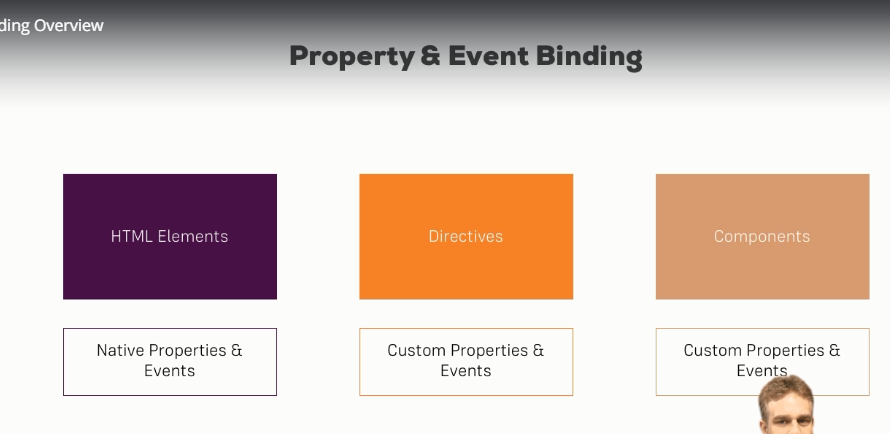


### Using Augury to Dive into Angular Apps



## 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>

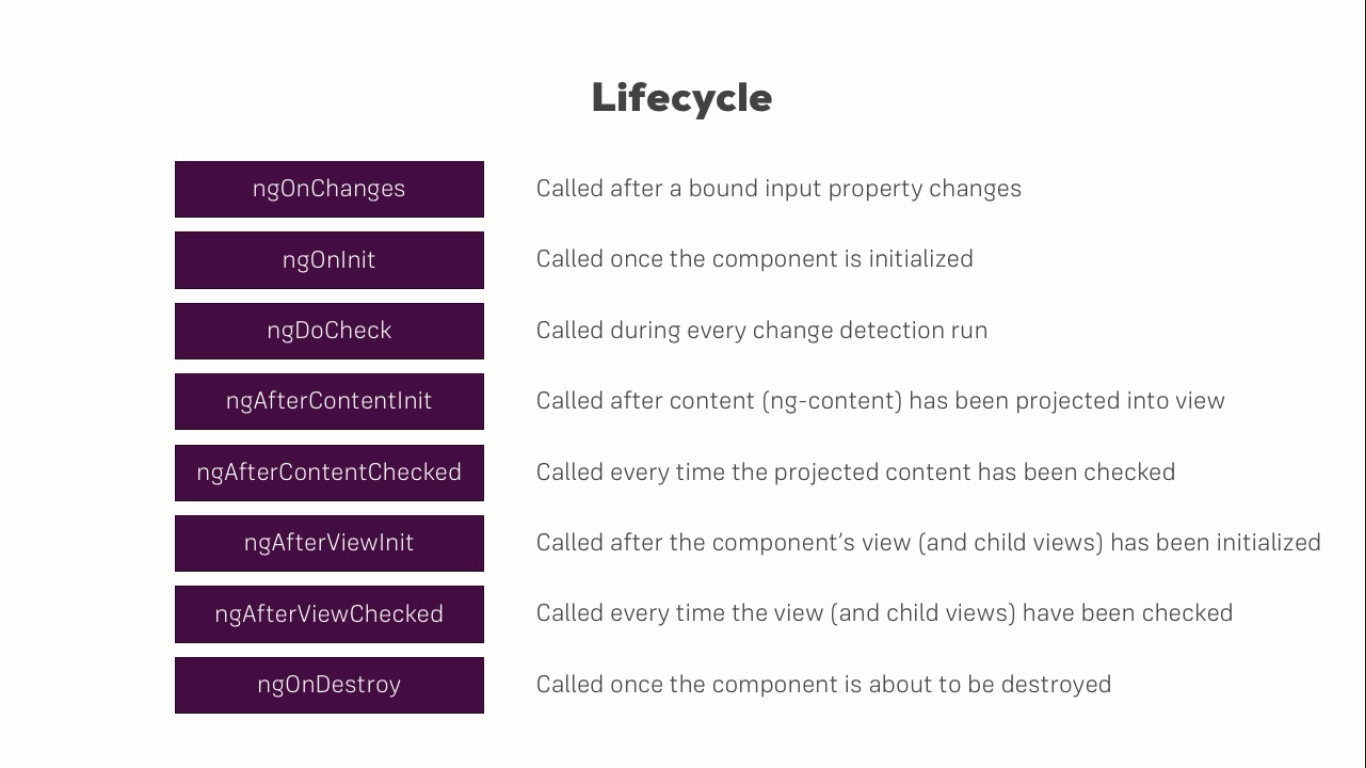
</p>

</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.

### One-way and Two-way Data Binding in Angular

One-way and two-way data binding are two of the important ways by which we can exchange data from component to DOM and vice-versa. Data exchange between the component and the view will help us to build dynamic and interactive web applications.

One-way Data Binding

One-way data binding will bind the data from the component to the view (DOM) or from view to the component. One-way data binding is unidirectional. You can only bind the data from component to the view or from view to the component.

From Component to View

There are different techniques of data binding which use one-way data binding to bind data from component to view. Below are some of the techniques, which uses one-way data binding.

Interpolation Binding: Interpolation refers to binding expressions into marked up language.

Property Binding: Property binding is used to set a property of a view element. The binding sets the property to the value of a template expression.

Attribute Binding: Attribute binding is used to set a attribute property of a view element.

Class Binding: Class binding is used to set a class property of a view element.

Style Binding: Style binding is used to set a style of a view element.

Let's consider an example using the interpolation technique where we are binding two values, firstName and the lastName, to the view, enclosed in double curly braces: {{property Name}}.

In this example, the data binding is done from component to the view. Any changes to the values in the component will be reflected in the view not vice-versa.

File Name: app.component.ts

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

@Component({

selector: 'app-example',

template: `

<div>

<strong>{{firstName}}</strong>

<strong>{{lastName}}</strong>

</div>

`

})

export class AppComponent {

firstName: string = "Yallaling";

lastName:string = "Goudar";

}

Let's consider another example using property binding. In this example, we are binding one value, firstName, to the innerHTML property of the span tag. It will bind the value of firstName to the span element.

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

@Component({

selector: 'app-example',

template: `

<div>

<span [innerHTML]='firstName'></span>

</div>

`

})

export class AppComponent {

firstName: string = "Yallaling";

}

Let's consider one more example of style binding. In this example, we are binding a color style to the 'h1' element. It will display the text within the h1 tags in a blue color.

<h1 [style.color]="blue">This is a Blue Heading</h1>

From View to Component

One-way data binding from view to the component can be achieved by using the event binding technique.

Let's consider an example where within parentheses on the left of the equal sign we have the target event like "click" and on the right side we may have the template statements such as component properties and methods(myFunction() in this case) bind to the event.

<button (click)="myFunction()">Show alert</button>

In the above code, the myFunction() method in the component will be called when user clicks on the button.

Filename app.component.ts

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

@Component({

selector: 'app-example',

template: `<button (click)='myFunction()' >Show alert</button>`

})

export class AppComponent {

myFunction(): void {

alert('Show alert!');

}

}

Once you run the above code, you will see a button with text "Show alert". When you click that button, it will call the myFunction() method in the component, which will, in turn, execute the alert() method, showing an alert box with the text "Show an alert".

**Two-way Data Binding in Angular**

Two-way data binding in Angular will help users to exchange data from the component to view and from view to the component. It will help users to establish communication bi-directionally.

Two-way data binding can be achieved using a ngModel directive in Angular. The below syntax shows the data binding using (ngModel), which is basically the combination of both the square brackets of property binding and parentheses of the event binding.

<input type="text" [(ngModel)] = 'val' />

Before using ngModel to achieve two-way data binding, it’s very important to import the FormsModule from @angular/forms in app.module.ts file as shown below. FormsModule will contain the ngModule directive.

Filename app.module.ts

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

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

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

import { AppComponent } from './app.component';

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

@NgModule({

imports: [BrowserModule, FormsModule],

declarations: [ AppComponent],

bootstrap: [AppComponent]

})

export class AppModule { }

If you do not import the FormsModule, then you will get Template parse errors and it will result in this error:

"Can't bind to 'ngModel' since it is not a known property of 'input'".

After importing the FormsModule, you can go ahead and bind data using (ngModel) as shown below.

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

@Component({

selector: 'app-example',

template: `

Enter the value : <input [(ngModel)] ='val'>

<br>

Entered value is: {{val}}

`

})

export class AppComponent {

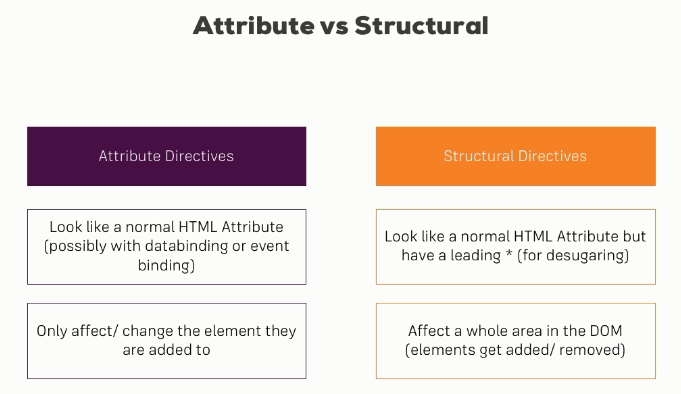
val: string = '';

}

Once we run the above code, we will see an input box asking us to enter a value in the view. Any value entered in that input box will be bound with the text below. Let's assume a user entered the text "John", then the text will be "Entered value is: John".

<https://www.pluralsight.com/guides/one-and-two-way-data-binding-angular>

# 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) {

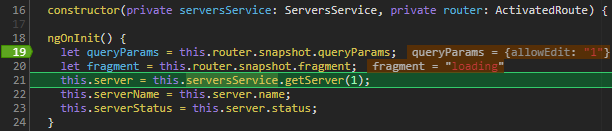
**this**.router.navigate(['/servers', id, 'edit'], {queryParams: {allowEdit: '1'}, fragment: 'loading'});

}

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]

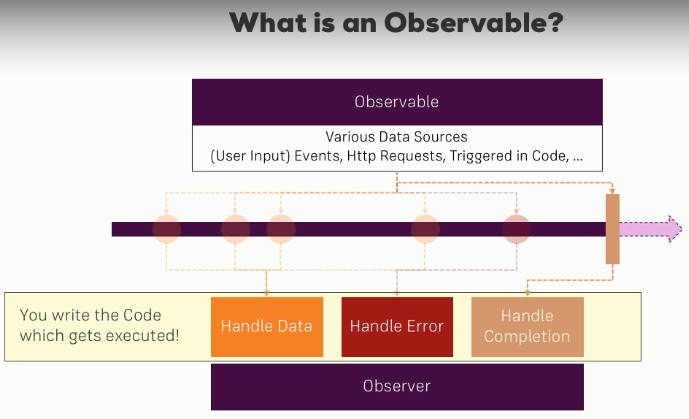
})

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);

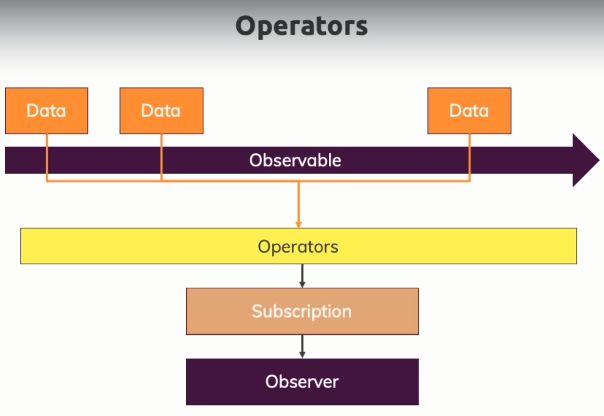
}, () **=>** {

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 {

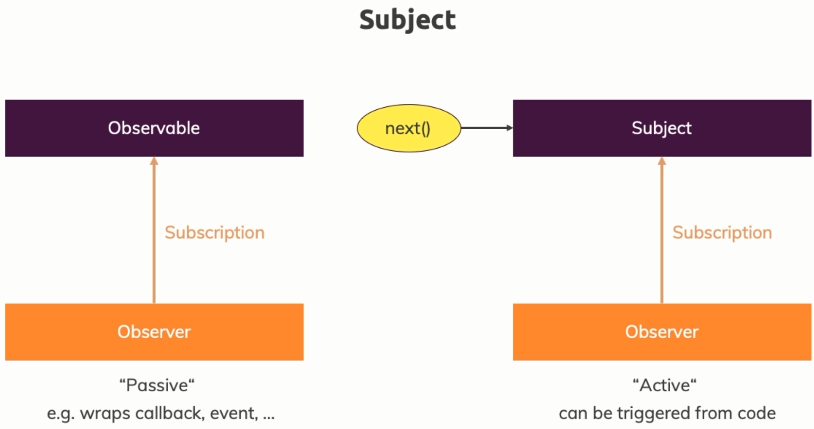
**this**.firstObsSubscription.unsubscribe();

}

}

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.

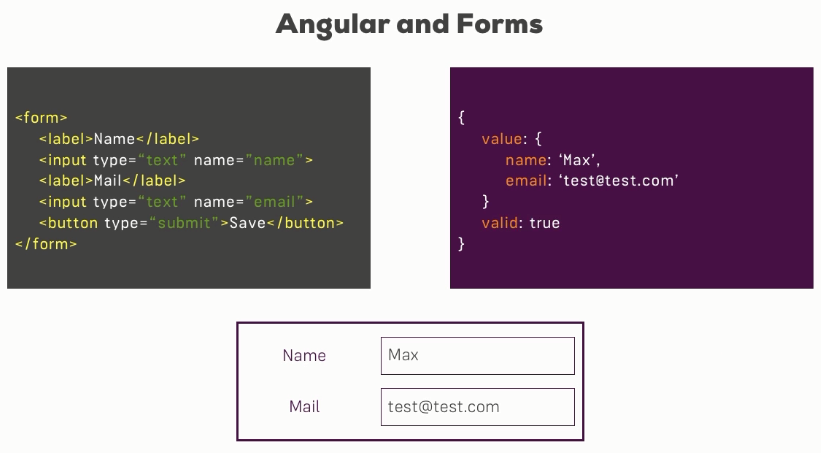
See an example in: Services communication with Subject (Observable).

**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/>

# Section 15: Handling Forms in Angular Apps

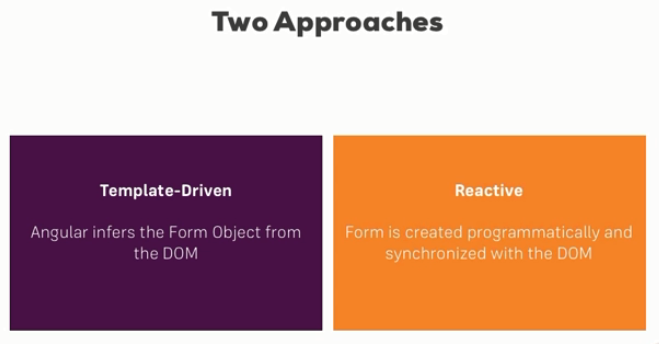
Angular help us to work with forms. This is what Angular does for you, it gives you actually a Javascript object representation of your form, making it simple for you to retrieve user values and to see the state of the form and to work with it.



It's super important to understand that Angular actually offers two approaches when it comes to handling forms.

It offers the template driven approach, which is called like this because there, you simply set up your form in the template, in HTML code and Angular will automatically infer the structure of your form, will infer which controls your forms has, which inputs and makes it easy for you to get started quickly.

It also has a more complex approach, the reactive approach. There, you actually define the structure of the form in TypeScript code, you also set up the HTML code and then you manually connect it which might sound more complicated than it is in the end and therefore, it gives you greater control over it, you can fine tune every little piece about your form.



# Creating the Form and Registering the Controls

If we have a look at this HTML code, you will quickly realize that on the form tag here, I don't have the action attribute pointing to some route. I'm also not specifying the method attribute which typically would be post. The reason for this is that this form should not get submitted to a server, I don't want a HTTP request to be the result of me clicking the submit button, instead, Angular should handle this form and therefore, I don't have an action on it.

Make sure that in your app module, you actually import the forms module, add it here to your imports array and have the import at the top of this file. As the name implies, this built-in module shipping with Angular includes a lot of forms related functionalities and it's actually needed to get this template driven approach to work, to get this form creation by Angular to work. Now by default in a CLI project, this should already be imported so it should work fine.

Steps to user Forms:

1. Import in your app.module.ts file the module FormsModule. With this imported, Angular will actually automatically create a form for you, so a Javascript object representations of the form when it detects a form element in HTML code.)
2. Angular will not automatically detect your inputs or others elements in this form. So you need to tell Angular hey within that form element, what should be an actual control of my javascript object. For that we have to put ngModel in the element and also give the HTML name of the element.
3. We have to add to the form in the HTML the event ngSubmit that going to call a function we should create in the ts file. Also to this function we have to pass a parameter with a reference to the form, a reference of type ngForm. <form (ngSubmit)="onSubmit(f)" #f="ngForm">
4. In the TS we going to receive in the function a parameter of type ngForm and inside of this object we going to have the value property, where finally, we have a key-value of or html elements.

app.module.ts  
**import** { FormsModule } **from** '@angular/forms';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

],

providers: [],

bootstrap: [AppComponent]

app.component.html

<form (ngSubmit)="onSubmit(f)" #f="ngForm">

<div id="user-data">

<div class="form-group">

<label for="username">Username</label>

<input type="text" id="username" class="form-control"

ngModel

name="username">

</div>

<button class="btn btn-default" type="button">Suggest an Username</button>

<div class="form-group">

<label for="email">Mail</label>

<input type="email" id="email" class="form-control"

ngModel

name="email">

</div>

</div>

<button class="btn btn-primary" type="submit">Submit</button>

</form>

app.component.ts

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

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

@Component({

selector: 'app-root',

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

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

})

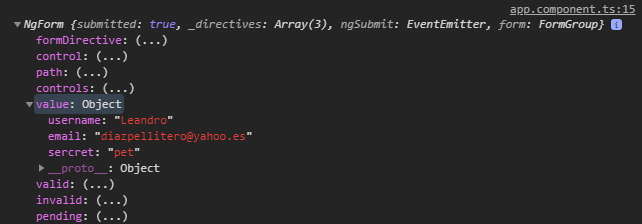
**export** **class** AppComponent {

onSubmit(form: NgForm) {

console.log(form);

}

}



## Understanding Form State

In the ngForm object that we receive in the TS we have a lot of properties that help us to know the state of the form, e.g.

* We can see which controls we registered here on the controls object, each control is of type FormControl.
* Dirty for example is true because we changed something in the elements about that form. If I reload the page and submit it now, you will see that dirty is false because I didn't type into any input.
* Disabled would be true if the form was disabled for some reason, invalid is false because we haven't added any validators.
* You do have the valid property down here too.
* Touched for example to see if we click into some of the fields, the difference to dirty would be that for dirty, we have to change the field, have to change the value of a field, for touch we simply.

## Accessing the Form with @ViewChild

The last approach is absolutely fine and probably the approach you will use in many use cases.

I just want to bring some other approach to your attention. We can also use @ViewChild in TS to retrieve the form. So this gives us access to the very same form without passing it to onSubmit function, this is especially useful if you need to access the form, not just at the point of time when you submit it, but also earlier. E. g.

app.component.html

<form (ngSubmit)="onSubmit()" #f="ngForm">

</form>

app.component.ts

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

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

@Component({

selector: 'app-root',

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

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

})

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

@ViewChild('f', {static: **false**}) signupForm: NgForm;

// onSubmit(form: NgForm) {

// console.log(form);

// }

onSubmit() {

console.log(**this**.signupForm);

}

}

## Adding Validation to check User Input

Whilst you should still validate input on the server as the front-end can always be tricked, you can greatly enhance user experience by also validating the input in the template.

Here we can for example add **required** to our username input. Now required is a default HTML attribute you can add to an input, here however Angular will detect it. And on the e-mail, we can for example therefore also add required and there also is an **email directive** you can add. Now e-mail is not a built-in HTML attribute, it is an Angular directive. Now but if we have a look at it and check the valid attributes, you see it is false when we don’t enter values to the inputs. Also it's filled but the e-mail address is not a valid email, you see that still valid is false. E.g.

<div class="form-group">

<label for="username">Username</label>

<input type="text" id="username" class="form-control"

ngModel

name="username"

required>

</div>

<button class="btn btn-default" type="button">Suggest an Username</button>

<div class="form-group">

<label for="email">Mail</label>

<input type="email" id="email" class="form-control"

ngModel

name="email"

required email>

</div>

Note:

**Built-in Validators & Using HTML5 Validation**

Which Validators do ship with Angular?

Check out the Validators class: <https://angular.io/api/forms/Validators> - these are all built-in validators, though that are the methods which actually get executed (and which you later can add when using the reactive approach).

For the template-driven approach, you need the directives. You can find out their names, by searching for "validator" in the official docs: <https://angular.io/api?type=directive> - everything marked with "D" is a directive and can be added to your template.

Additionally, you might also want to enable HTML5 validation (by default, Angular disables it). You can do so by adding the ngNativeValidate to a control in your template.

## Using the Form State

In the last lecture we found out that Angular tracks the state of each control of the form, whether it's

valid and so on and conditionally also **adds these CSS classes**. E. g.

Classes of an input at start the page: class="form-control ng-untouched ng-pristine ng-invalid"

Classes after write something valid: class="form-control ng-dirty ng-valid ng-touched"

With this information we can for example, draw a red border to the input control if it not valid, also let's disable the submit button if the form is not valid. E.g.

app.component.html

<form (ngSubmit)="onSubmit()" #f="ngForm">

<div id="user-data">

<div class="form-group">

<label for="username">Username</label>

<input type="text" id="username" class="form-control"

ngModel

name="username"

required>

</div>

<button class="btn btn-default" type="button">Suggest an Username</button>

<div class="form-group">

<label for="email">Mail</label>

<input type="email" id="email" class="form-control"

ngModel

name="email"

required email>

</div>

</div>

<button class="btn btn-primary" type="submit"

[disabled]="!f.valid">Submit</button>

</form>

app.component.css

input.ng-invalid.ng-touched {

border: 1px solid red;

}

**Outputting Validation Error Messages**

Well, a quick and easy way of getting access to the control created by Angular automatically as we added ngModel to the input is by adding a local reference to this input element here, for example e-mail, any name you like and associating this now not with ngForm as we did for the overall form but with **ngModel**.

So just like the form directive automatically added by Angular when it detects a form element, the

ngModel directive here also kind of exposes some additional information about the control it creates for us on the overarching form by accessing ngModel.

So with this, we could simply check or say that we want to attach this span here if e-mail is not valid, so add an exclamation mark at the beginning.

<div class="form-group">

<label for="email">Mail</label>

<input type="email" id="email" class="form-control"

ngModel

name="email"

required email

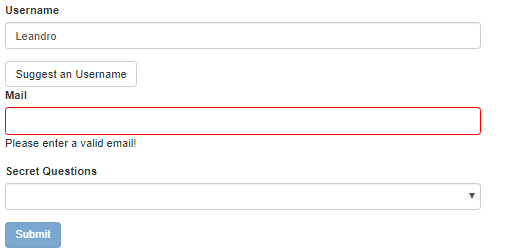
#email="ngModel">

<span class="hekp-block" \*ngIf="!email.valid && email.touched">

Please enter a valid email!

</span>

</div>



## Set Default Values with ngModel Property Binding

You could point to a default string in the form, pre-populating some field with a default value.

Very useful and something you should definitely keep in mind, that you are not limited to using

ngModel without any bindings, you can use one-way binding, property binding to set a default value. E.g.

app.component.ts

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

@ViewChild('f', {static: **false**}) signupForm: NgForm;

defaultQuestion = 'pet';

}

app.component.html

<div class="form-group">

<label for="secret">Secret Questions</label>

<select id="secret" class="form-control"

[ngModel]="defaultQuestion"

name="sercret">

<option value="pet">Your first Pet?</option>

<option value="teacher">Your first teacher?</option>

</select>

</div>

## Using ngModel with Two-Way-Binding

Sometimes you not only want to pre-populate a default the user entered but you also want to instantly react to any changes.

Right now, everything about this form here only updates once I click submit, then I get this form object where I can retrieve the value.

We don't use two-way binding, ngModel is either used without any binding or with one-way binding but sometimes you instantly want to check something or simply repeat whatever the user entered, let's show an example. We going to have a text area and when we write, we going to reply the same text in a paragraph.

app.component.ts

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

@ViewChild('f', {static: **false**}) signupForm: NgForm;

defaultQuestion = 'pet';

answer = '';

}

app.component.html

<div class="form-group">

<textarea name="questionAnswear" rows="3" class="form-control"

[(ngModel)]="answer"></textarea>

</div>

<p>Your reply: {{answer}}</p>



So you can still use two-way binding and actually if I submit the whole form and open up the developer tools, you'll see in the value object, this is still part of this value object.

So two-way binding is still possible, you can still use ngModel with two-way binding and with that, you saw all three forms. No binding to just tell Angular that an input is a control, one-way binding to give that control a default value and two-way binding to instantly output it or do whatever you want to do with that value.

## Grouping Form Controls

In our javascript form object, we can group elements of the html form, this could be helpful when we have a Form with a lot of components. We could also validate our group of elements.

In our template , we put in the div that contain our html elements to group the folloging code: <div id="user-data" ngModelGroup="userData">, where userData will be the key to be showed in the javascript form object . E.g.

app.component.html

<div id="user-data" ngModelGroup="userData" #userDataRef="ngModelGroup">

<div class="form-group">

<label for="username">Username</label>

<input type="text" id="username" class="form-control"

ngModel

name="username"

required>

</div>

<button class="btn btn-default" type="button">Suggest an Username</button>

<div class="form-group">

<label for="email">Mail</label>

<input type="email" id="email" class="form-control"

ngModel

name="email"

required email

#email="ngModel">

<span class="hekp-block" \*ngIf="!email.valid && email.touched">

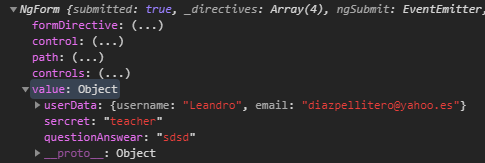
Please enter a valid email!

</span>

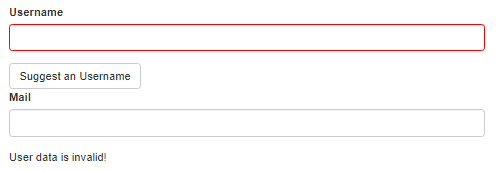
</div>

</div>

<p \*ngIf="!userDataRef.valid && userDataRef.touched">User data is invalid!</p>



Also if we put a reference in the div #userDataRef="ngModelGroup">, we can check if elements of the group are valid.



## Handling Radio Buttons

So I just want to highlight this because radio buttons can look like a very special case, they aren't, they are used like any other input when using the template driven approach in your Angular app. E.g

app.component.ts

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

@ViewChild('f', {static: **false**}) signupForm: NgForm;

defaultQuestion = 'pet';

answer = '';

genders = ['male', 'female'];

}

app.componet.html

<div class="radio" \*ngFor="let gender of genders">

<label>

<input type="radio"

name="gender"

ngModel

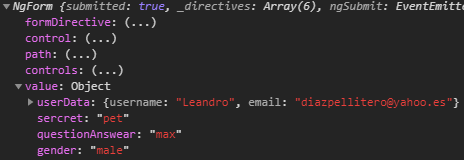
[value]="gender"

required>

{{gender}}

</label>

</div>



## Setting and Patching Form Values

Sopporting in our @ViewChild variable of type NgForm in our TS we can access to the elements in the HTML form and modify them.

We have two commands that are very helpful, *setValue* to set your whole form and *patchValue* to overwrite parts of the form. E.g.

app.component.ts

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

@ViewChild('f', { static: **false** }) signupForm: NgForm;

suggestUserName() {

**const** suggestedName = 'Superuser';

// this.signupForm.setValue({

// userData: {

// username: suggestedName,

// email: ''

// },

// secret: 'pet',

// questionAnswer: '',

// gender: 'male'

// });

**this**.signupForm.form.patchValue({

userData: {

username: suggestedName

}

});

}

app.componet.html

<form (ngSubmit)="onSubmit()" #f="ngForm">

<div id="user-data" ngModelGroup="userData" #userDataRef="ngModelGroup">

<div class="form-group">

<label for="username">Username</label>

<input type="text" id="username" class="form-control"

ngModel

name="username"

required>

</div>

<button class="btn btn-default" type="button" (click)="suggestUserName()">Suggest an Username</button>

<div class="form-group">

<label for="email">Mail</label>

<input type="email" id="email" class="form-control"

ngModel

name="email"

required email

#email="ngModel">

<span class="hekp-block" \*ngIf="!email.valid && email.touched">

Please enter a valid email!

</span>

</div>

</div>

<p \*ngIf="!userDataRef.valid && userDataRef.touched">User data is invalid!</p>

<div class="form-group">

<label for="secret">Secret Questions</label>

<select id="secret" class="form-control"

[ngModel]="defaultQuestion"

name="sercret">

<option value="pet">Your first Pet?</option>

<option value="teacher">Your first teacher?</option>

</select>

</div>

<div class="form-group">

<textarea name="questionAnswear" rows="3" class="form-control"

[(ngModel)]="answer"></textarea>

</div>

<p>Your reply: {{answer}}</p>

<div class="radio" \*ngFor="let gender of genders">

<label>

<input type="radio"

name="gender"

ngModel

[value]="gender"

required>

{{gender}}

</label>

</div>

<button class="btn btn-primary" type="submit"

[disabled]="!f.valid">Submit</button>

</form>

## Using Form Data

Now that we have access to al information in the form, we can store that information in a variable and them show that information in the HTML template. This is the usual way to do it:

app.component.ts

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

@ViewChild('f', { static: **false** }) signupForm: NgForm;

defaultQuestion = 'teacher';

answer = '';

genders = ['male', 'female'];

user = {

username: '',

email: '',

secretQuestion: '',

answer: '',

gender: ''

};

submitted = **false**;

onSubmit() {

**this**.submitted = **true**;

**this**.user.username = **this**.signupForm.value.userData.username;

**this**.user.email = **this**.signupForm.value.userData.email;

**this**.user.secretQuestion = **this**.signupForm.value.secret;

**this**.user.answer = **this**.signupForm.value.questionAnswer;

**this**.user.gender = **this**.signupForm.value.gender;

**this**.signupForm.reset();

}

}

app.componet.html

<hr>

<div class="row" \*ngIf="submitted">

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

<h3>Your Data</h3>

<p>Username: {{ user.username }}</p>

<p>Mail: {{ user.email }}</p>

<p>Secret Question: Your first {{ user.secretQuestion }}</p>

<p>Answer: {{ user.answer }}</p>

<p>Gender: {{ user.gender }}</p>

</div>

</div>

## Resetting Forms

Now let's say we did extract all the data and now we want to reset the form, well turns out this is super simple. On our form, signupForm in this case which again is just fetched directly from our template with @ViewChild, on this form, we can call reset like this: **this**.signupForm.reset();

And this will reset the form and what this means is it will not only empty all the inputs, it will also reset the state like the valid, the touched and so on, these properties.

## Reactive Forms

### Setup

Now as I said, in the reactive approach, the form is created programmatically, so in TypeScript code, so this is where we should start working on it. Angular gives us a property called **FormGroup** with which we can work which will hold our form in the end.

We also need to add to the app.module.ts file the module **ReactiveFormsModule**. You don't need the FormsModule, this is required for the template driven approach, and instead you need the reactive forms module.

app.module.ts

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

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    ReactiveFormsModule

  ],

app.component.ts

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  genders = ['male', 'female'];

  sigupForm: FormGroup;

}

### Syncing HTML and Form

Now we going to initialize our FormGroup in TS File. We going to put inside of it all the elements that we have in the HTML template, also each of the element going be of a generic type called by angular FormControl, to the FormCOntrol we pass the html component initialization.

App.component.ts  
export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  sigupForm: FormGroup;

  ngOnInit(): void {

    this.sigupForm = new FormGroup({

      'username': new FormControl(null),

      'email': new FormControl(null),

      'gender': new FormControl('male')

    });

  }

}

In our template now we going to use the **formGroup** directive. Now this simply tells Angular, hey please take my formGroup, don't infer one, don't create a form for me, use my formGroup and we need to set up property binding here because we need to pass our form as an argument to the directive ([formGroup]="sigupForm"). So here we should reference our signupForm, the property we created here which stores our form. We're passing this via property binding to the formGroup and now this form is actually synchronized.

Also to relate every element in the html form with elements in the TS object we have to use the **formControlName** directive (**formControlName**="name\_in\_TS" ). E.g.

<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">

      <form [formGroup]="sigupForm">

        <div class="form-group">

          <label for="username">Username</label>

          <input

            type="text"

            id="username"

            formControlName="username"

            class="form-control">

        </div>

        <div class="form-group">

          <label for="email">email</label>

          <input

            type="text"

            id="email"

            formControlName="email"

            class="form-control">

        </div>

        <div class="radio" \*ngFor="let gender of genders">

          <label>

            <input

              type="radio"

              formControlName="gender"

              [value]="gender">{{ gender }}

          </label>

        </div>

        <button class="btn btn-primary" type="submit">Submit</button>

      </form>

    </div>

  </div>

</div>

### Submitting the Form

In the template driven approach, we used ngSubmit, the ngSubmit directive here on this form element. Well we still do the same here because we still want to react to this default submit event which is fired by HTML, by Javascript.

app.component.html

<form [formGroup]="sigupForm" (ngSubmit)="onSubmit()">

How we have our object in the TC, we don’t have to put a reference to the form, in the TC we just use it.

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  sigupForm: FormGroup;

  ngOnInit(): void {

    this.sigupForm = new FormGroup({

      'username': new FormControl(null),

      'email': new FormControl(null),

      'gender': new FormControl('male'),

    });

  }

  onSubmit() {

    console.log(this.sigupForm);

  }

}





### Adding Validation

In the template driven approach, we would simply add required here for example to make this field required. It doesn't work like this in the reactive approach because keep in mind and that's key, you're not configuring the form in the template, you're only synchronizing it with the directives, formControlName and formGroup but you're not configuring it here, the TypeScript code, that's where you configure it.

That is why FormControl takes more than one argument, this constructor here.

You cannot only set the default value, the second argument allows you to specify some validators. So you can either only pass one validator or you can also add multiple validators by simply passing an array of validators. E. g.

import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  sigupForm: FormGroup;

  ngOnInit(): void {

    this.sigupForm = new FormGroup({

      'username': new FormControl(null, Validators.required),

      'email': new FormControl(null, [Validators.required, Validators.email]),

      'gender': new FormControl('male'),

    });

  }

### Getting Access to Controls

The way to access the controls by accessing our overall form and here, we have a **get** method. The get method allows us to get access to our controls easily, here you can either specify the control name. Now we can determine if the controls are valid or have been touched. E.g.

      <form [formGroup]="signupForm" (ngSubmit)="onSubmit()">

        <div class="form-group">

          <label for="username">Username</label>

          <input

            type="text"

            id="username"

            formControlName="username"

            class="form-control">

          <span \*ngIf="!signupForm.get('username').valid &&   
 signupForm.get('username').touched"

            class="help-block">Please enter a valid username!</span>

        </div>

        <div class="form-group">

          <label for="email">email</label>

          <input

            type="text"

            id="email"

            formControlName="email"

            class="form-control">

            <span \*ngIf="!signupForm.get('email').valid && signupForm.get('email').touched"

            class="help-block">Please enter a valid email!</span>

        </div>

        <div class="radio" \*ngFor="let gender of genders">

          <label>

            <input

              type="radio"

              formControlName="gender"

              [value]="gender">{{ gender }}

          </label>

        </div>

        <span \*ngIf="!signupForm.valid && signupForm.touched"

            class="help-block">Please enter a valid data!</span>

        <button class="btn btn-primary" type="submit">Submit</button>

      </form>

Reactive: Grouping Controls

In the TS, we could create a FormGroup named userData and now this is a new FormGroup again. FormGroup is not only there to be used on the overall form, that just happens to be a FormGroup too but you can still have form groups in the form groups.

On the other hand in the template we just add a div inside my form and use the directive forGroupName setting as parameter the name of the grouping formgroup. Also when we get the element have to put the path separated with dot (signupForm.get('userData.username')) E.g.

app.component.ts

  ngOnInit(): void {

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

        'username': new FormControl(null, Validators.required),

        'email': new FormControl(null, [Validators.required, Validators.email]),

      }),

      'gender': new FormControl('male'),

    });

  }

app.component.html

<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">

  <div formGroupName="userData">

    <div class="form-group">

      <label for="username">Username</label>

      <input

        type="text"

        id="username"

        formControlName="username"

        class="form-control">

      <span \*ngIf="!signupForm.get('userData.username').valid && signupForm.get('userData.username').touched"

        class="help-block">Please enter a valid username!</span>

    </div>

    <div class="form-group">

      <label for="email">email</label>

      <input

        type="text"

        id="email"

        formControlName="email"

        class="form-control">

        <span \*ngIf="!signupForm.get('userData.email').valid && signupForm.get('userData.email').touched"

        class="help-block">Please enter a valid email!</span>

    </div>

  </div>

  <div class="radio" \*ngFor="let gender of genders">

    <label>

      <input

        type="radio"

        formControlName="gender"

        [value]="gender">{{ gender }}

    </label>

  </div>

  <span \*ngIf="!signupForm.valid && signupForm.touched"

      class="help-block">Please enter a valid data!</span>

  <button class="btn btn-primary" type="submit">Submit</button>

</form>

### Arrays of Form Controls (FormArray)

We can also create an array of html elements in our Form with the type **FormArray**, and inside of it every element will be of type FormControl. To get the information of the array we have to use a cast to <FormArray>. On the other hand in the template we use a div with the directive **formArrayName** (formArrayName="hobbies") passing as parameter the name of the array in our object. E.g.

app.component.ts

import { FormGroup, FormControl, Validators, FormArray } from '@angular/forms';

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  signupForm: FormGroup;

  ngOnInit(): void {

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

        'username': new FormControl(null, Validators.required),

        'email': new FormControl(null, [Validators.required, Validators.email])

      }),

      'gender': new FormControl('male'),

      'hobbies': new FormArray([])

    });

  }

  onAddHobby(){

    const control = new FormControl(null, Validators.required);

    (<FormArray>this.signupForm.get('hobbies')).push(control);

  }

  getControls() {

    return (<FormArray>this.signupForm.get('hobbies')).controls;

  }

app.component.html

        <div formArrayName="hobbies">

          <h4>your hobbies</h4>

          <button class="btn btn-default" type="button"

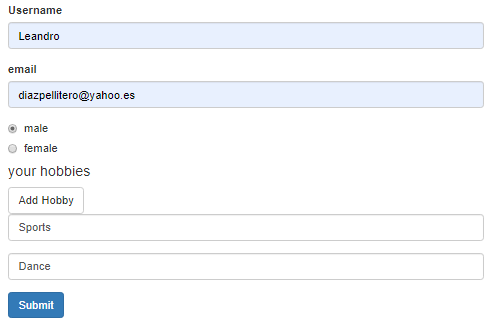
            (click)="onAddHobby()">Add Hobby</button>

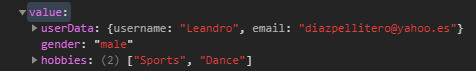
          <div class="form-group"   
 \*ngFor="let hobbyControl of getControls(); let i = index">

            <input type="text" class="form-control" [formControlName]="i">

          </div>

        </div>





Note: As of **Angular 8+**, there's a new way of **clearing all items**in a FormArray.

(<FormArray>this.recipeForm.get('ingredients')).clear();

The *clear()*method automatically loops through all registered FormControls (or FormGroups) in the FormArray and removes them.

It's like manually creating a loop and calling *removeAt()* for every item.

### Creating Custom Validators

Thus far, we always used the built-in ones and to be honest, with all the built-in ones, you should be able to cover most of your use cases but in some occasion you need to create your own validations. Let's say we have some usernames we don't want to allow the user to use.

A validator in the end is just a function which gets executed by Angular automatically when it checks the validity of the FormControl and it checks that validity whenever you change that control.

Now a validator to work correctly needs to receive an argument which is the control it should check, so this will be of type FormControl, a validator also needs to return something for Angular to be able to handle the return value correctly.

This something should be a Javascript object and now the following notation might look strange, it should have any key which can be interpreted as a string (in our case 'nameIsForbidden') and this is just TypeScript syntax for saying hey we want to have a key-value pair where the key again can be interpreted as a string which is true for a key in an object in general.

More importantly, the value of that key-value pair, that should be a boolean.

So in our example if our name is in the list we return true, which means it is invalid, on the other hand we have to return null or nothing. This is important, if validation is successful, you have to pass nothing or null, this might sound counter-intuitive but that's just how it works, it should be null or you simply omit the return statement. This is how you tell Angular that the FormControl is valid.

Then we add the validation to our component, but how we're not calling the function from inside this class, Angular will call it when it checks the validity, at this point of time, this will not refer to our class here. So to fix this, I actually need to ***bind*** this, the good old Javascript trick to make sure that this refers to what we want it to refer to. Bind function is necessary because we use **this**.forbiddenUserNames inside our validating function. Any time we use **this** operator inside a validation function we have to use bind.

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  signupForm: FormGroup;

  forbiddenUserNames = ['Crish', 'Anna'];

  ngOnInit(): void {

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

        'username': new FormControl(null, [Validators.required,  this.forbiddenNames.bind(this)]),

        'email': new FormControl(null, [Validators.required, Validators.email])

      }),

      'gender': new FormControl('male'),

      'hobbies': new FormArray([])

    });

  }

  forbiddenNames(control: FormControl): {[s: string]: boolean} {

    if(this.forbiddenUserNames.indexOf(control.value) !== -1){

      return {'nameIsForbidden': true}

    } else {

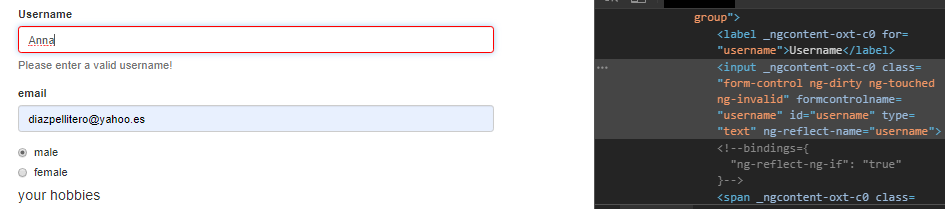
      return null;

    }

  }

}

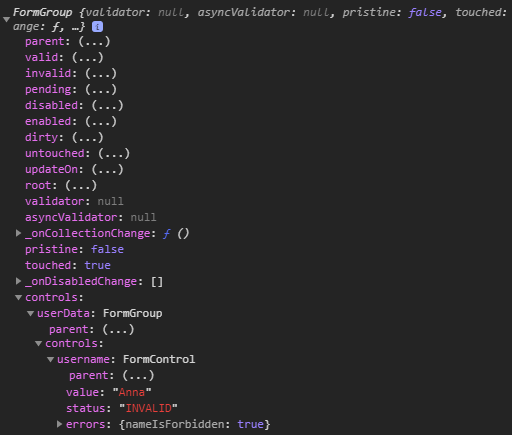
How if we type a name for user how is in the forbidden list, we show the invalid class.



Note: Function.bind() lets you specify the context that the function will execute in (that is, it lets you pass in what object the this keyword will resolve to in the body of the function.   
It seems that if I use operator this inside the function that going to be called, them, bind its necessary.

### Using Error Codes

In the last example we store an error of validation with the key:value , {'nameIsForbidden': true}, now we going to access to that error. Angular actually adds the error codes **on the individual controls on the errors object**. In our example we going to access to that error code in the form object to show a personalized error message to the user in the template.



<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">

  <div formGroupName="userData">

    <div class="form-group">

      <label for="username">Username</label>

      <input

        type="text"

        id="username"

        formControlName="username"

        class="form-control">

      <span \*ngIf="!signupForm.get('userData.username').valid   
 && signupForm.get('userData.username').touched"

class="help-block">

        <span \*ngIf="signupForm.get('userData.username').errors['nameIsForbidden']">

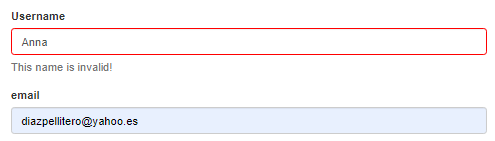
          This name is invalid!</span>

        <span \*ngIf="signupForm.get('userData.username').errors['required']">

          This field is requiered!</span>

      </span>

    </div>



### Creating a Custom Async Validator

Sometimes you might need to reach out to a web server to check the validation. That however is an asynchronous operation because the response is not coming back instantly, instead it just takes a couple of seconds.

So we also kind of need asynchronous validators which are able to wait for a response before returning true or false, is it valid or not.

In our TS we are created the function forbiddenEmail, how return a Promise or and Observable, but inside we return a Promise. Inside we use the setTimeout function to simulate the async process. Inside of our setTimeout function we have our logic, pretty similar to our before validator created. In this case if the email equals to 'test@test.com' we considered invalid and return the key-value 'emailIsForbidden': true, on the other hand return null. In this case how we are in a promise, we don’t return, we resolve.

Finally we can add it to our this.signupForm object on creation. Let's add it to the e-mail and here you don't add it in this array of normal validators, instead we use the third argument we can pass to FormControl. This argument is an asynchronous validator or an array of such validators, just like the normal validators but reserved for the asynchronous ones.  
How we don’t use this operator inside our validation function, there is no net to use the bind() function. Also we pass just the name of the function, because we want pass a reference, we don’t pass the parenthesis because we don’t want the execution on creation.

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  signupForm: FormGroup;

  forbiddenUserNames = ['Crish', 'Anna'];

  ngOnInit(): void {

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

        'username': new FormControl(null, [Validators.required,  this.forbiddenNames.bind(this)]),

        'email': new FormControl(null, [Validators.required, Validators.email],  this.forbiddenEmail)

      }),

      'gender': new FormControl('male'),

      'hobbies': new FormArray([])

    });

  }

  forbiddenNames(control: FormControl): {[s: string]: boolean} {

    if(this.forbiddenUserNames.indexOf(control.value) !== -1){

      return {'nameIsForbidden': true}

    } else {

      return null;

    }

  }

  forbiddenEmail(control: FormControl) : Promise<any> | Observable<any> {

    const promise = new Promise<any>( (resolve, reject) => {

      setTimeout( () => {

        if(control.value === 'test@test.com') {

          resolve ({'emailIsForbidden': true});

        } else {

          resolve (null);

        }

      }, 1500);

    });

    return promise;

  }



### Reacting to Status or Value Changes

As we saw the status of this input switched from invalid to pending to valid for example. There actually is a form state you can track in general this changes. I'll do this here in ngOnInit. On my signupForm and on each control of this form, so on get any control too, you have two observables you can listen to:

* statusChanges
* valueChanges.

So statusChanges and valueChanges, two nice hooks, observables you can listen to or you can subscribe to if you really want to closely watch what happens in your form or again on an individual control and want to react to that.

  ngOnInit(): void {

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

        'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),

        'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmail)

      }),

      'gender': new FormControl('male'),

      'hobbies': new FormArray([])

    });

    // this.signupForm.valueChanges.subscribe(

    //   (value) => console.log(value)

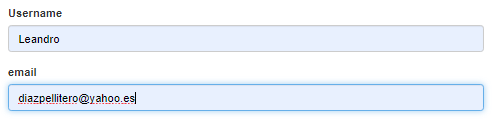
    // )

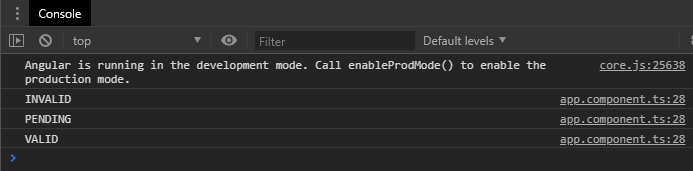
    this.signupForm.statusChanges.subscribe(

      (status) => console.log(status)

    )

  }





### Setting and Patching Values

Just like in the template driven approach, *setValue* and *patchValue* are there for you.

You can of course on your form as a whole call setValue and pass a Javascript object with the correct structure and override the whole object. Or use patchValue to only update a part of the form. E.g.

  ngOnInit(): void {

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

        'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),

        'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmail)

      }),

      'gender': new FormControl('male'),

      'hobbies': new FormArray([])

    });

    this.signupForm.setValue({

      'userData': {

        'username': 'Max',

        'email': 'max@test.com',

      },

      'gender': 'male',

      'hobbies': []

    });

    this.signupForm.patchValue({

      'userData': {

        'username': 'Anna'

      }

    });

  }

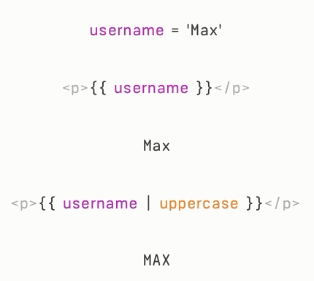
Also if we want to reset the form after submitting it, we can simply call reset here.

  onSubmit() {

    this.signupForm.reset();

  }

# Section 17: Using Pipes to Transform Output

What are pipes? Pipes are a feature built into Angular 2 which basically allows you to transform output in your template. This is the main purpose of a pipe, it transforms some output.

Now there are pipes for different types of output and also for synchronous and asynchronous data.

Sometimes you don't want to change the property value in TS itself because imagine you use that throughout your code and it should still be as you assigned it up there but you want to transform the way it is displayed once you render it to the screen. For this, this, you could use a pipe.

## Using Pipes

The right place to use a pipe logically is the template. Now that's easy to achieve, we just add the pipe symbol (|), hence the name, pipes and then the name of the pipe and as mentioned before, there are a couple of built-in pipes and you can also build pipes on your own. E.g changing the server instance type to uppercase and server date to date format.



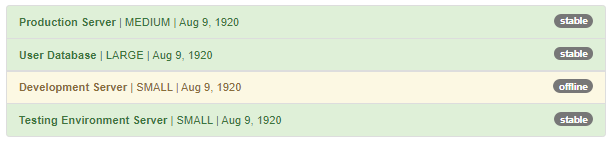
        <li>

          <strong>{{ server.name }}</strong> |

          {{ server.instanceType | uppercase }} |

          {{ server.started | date }}

        </li>



## Parametrizing Pipes

So it would be nice if we could parameterize a pipe and we can. For example on the date here, we can add a parameter to configure the pipe by adding a colon (:) and this is the case for any pipe, you configure it, you pass a parameter to it by adding a colon.   
The date pipe can take a parameter and we simply add it after this colon, also the date pipe expects to receive a string parameter. E.g.

          {{ server.started | date: 'fullDate' }}



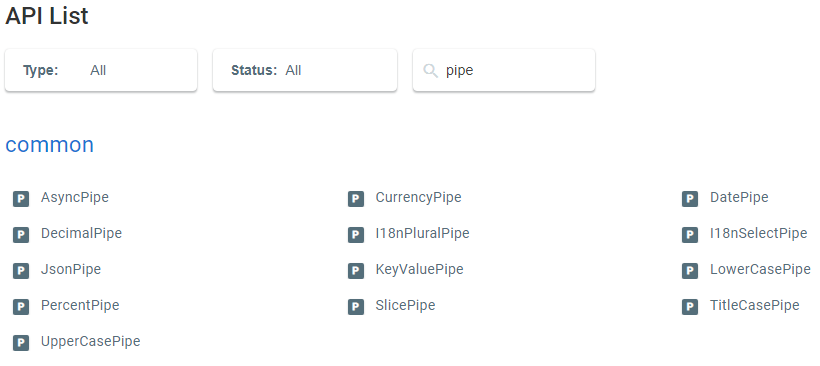
If you would have multiple parameters here, you simply separate them with additional colons.

## Where to learn more about Pipes

The place to learn more is the official documentation, so you can just head to angular.io and there under docs, click on API reference. So here, you can enter pipe as a filter and then you view all the pipes which are built into Angular and there you see we get a couple of them.

<https://angular.io/api?query=pipe>

Date pipe also have a lot of parameter or even you can create your date from cero.



## Chaining Multiple Pipes

Now you can simply chain pipes by adding a pipe symbol after a pipe. The order might be important, generally it will be parsed from left to right.

          {{ server.started | date: 'fullDate' | uppercase }}

## Creating a Custom Pipe

Steps to create your own pipe:

1. Create a file for our pipe for example shorten.pipe.ts.In the file create a class that implements the interface **PipeTransform**. This interface forces you to create the method transform (), who going to receive the parameters of the pipe. Mark the file with the @Pipe decorator and set the name that going to have the pipe. Also we have to import and set in the declarations part the pipe file in app.module.ts.
2. All the step one can be done with **ng g p pipe\_name** command, so this is a better way.

shorten.pipe.ts

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

@Pipe({

  name: 'shorten'

})

export class ShortenPipe implements PipeTransform {

  transform(value: any) {

    if (value.length > 10)

      return value.substr(0, 10) + ' ...';

    return value;

  }

}

app.module.ts

import { ShortenPipe } from './shorten.pipe';

@NgModule({

  declarations: [

    AppComponent,

    ShortenPipe

  ],

app.component.html

          <strong>{{ server.name | shorten }}</strong> |



## Parametrizing a Custom Pipe

We just have to add a new parameter or various one, and after that passed it colon.

shorten.pipe.ts

  transform(value: any, limit: number) {

    if (value.length > limit)

      return value.substr(0, limit) + ' ...';

    return value;

  }

app.component.html

          <strong>{{ server.name | shorten : 10 }}</strong> |

## Applying pipes to \*NgFor

Before we only used pipes in string interpolation but keep in mind that that pipes transform your output and, **an ngFor loop is simply part of your output**. Therefore of course you can add a pipe here too.

In this case we going to put an input and we going to filter the elements of the list, to show only the elements that match with the input filter. E.g.

filter.pipe.ts

@Pipe({

  name: 'filter'

})

export class FilterPipe implements PipeTransform {

  //the value in this case is an array of objects

  transform(value: any, filterString: string, propName: string): any {

    if(value.length === 0 || filterString === '') {

      return value;

    }

    const resultArray = [];

    for(const item of value) {

      if(item[propName] === filterString) {

        resultArray.push(item);

      }

    }

    return resultArray;

  }

}

app.component.ts

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  servers = [

    {

      instanceType: 'medium',

      name: 'Production Server',

      status: 'stable',

      started: new Date(15, 1, 2017)

    },

    {

      instanceType: 'large',

      name: 'User Database',

      status: 'stable',

      started: new Date(15, 1, 2017)

    },… continua pero lo he cortado

  ];

  filteredStatus = '';

app.component.html

      <input type="text" [(ngModel)]="filteredStatus">

      <hr>

      <ul class="list-group">

        <li

          class="list-group-item"

          \*ngFor="let server of servers | filter:filteredStatus:'status'"

          [ngClass]="getStatusClasses(server)">

          <span class="badge">

            {{ server.status }}

          </span>

          <strong>{{ server.name | shorten: 10 }}</strong> |

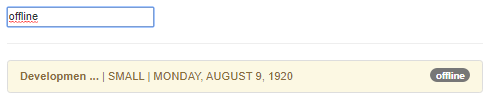
          {{ server.instanceType | uppercase }} |

          {{ server.started | date: 'fullDate' | uppercase }}

        </li>

      </ul>

    </div>



## Pure and Impure Pipes (or: How to "fix" the Filter Pipe)

If we change the data list of our page that we are filtering, that data update will not be reflected. Now the reason for this behavior is that Angular is, thankfully I should say, not rerunning our pipe on the data whenever this data changes.

So as soon as we change the filter and even if we would only add a blank space and then remove it, we would update our pipe again as you can see. So adding the input or changing the input of the pipe will trigger a recalculation, really will trigger the pipe being applied to the data again but changing the data won't trigger this and this is a good behavior because otherwise, Angular would have to run this pipe or rerun the pipe whenever any data on the page changes.

This would be really bad because that would cost a lot of performance and this is also the reason why no built-in filter pipe exists because filtering you would say is a pretty common task but the Angular team decided against adding such a pipe because you typically have a high performance cost if you want to enforce it being updated even if you are in filter mode.

So by default, it doesn't work here but we can force it to work but again, be aware that the following change will make sure that whenever we change data on the page, our pipe is recalculated you could say. So this might lead to performance issues, so you should know what you're doing which is why I'm really emphasizing this.

You can force this pipe to be updated whenever the data changes by adding a second property to the pipe decorator, it's called pure and you can set it to false. By default, this is true and doesn't need to be added.

@Pipe({

  name: 'filter',

  pure: false

})

Understanding the "async" Pipe

Angular also have a built-in pipe to be used with asynchronous methods. So, if we have an async method that will delay the response, and this response will be showed by angular, we can use the pipe to do show nothing until the promise will be solved. E.g.

app.component.ts

export class AppComponent {

  appStatus = new Promise((resolve, reject) => {

    setTimeout(() => {  //simulating async method

      resolve ('stable');

    }, 2000);

  });

app.component.html

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

      <input type="text" [(ngModel)]="filteredStatus">

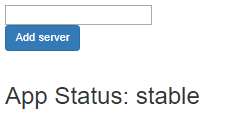
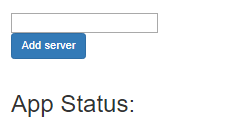
      <br>

      <button class="btn btn-primary" (click)="onAddServer()">  
 Add server</button>

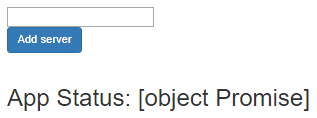
      <br><br>

      <h2>App Status: {{ appStatus | async }}</h2>

      <hr>

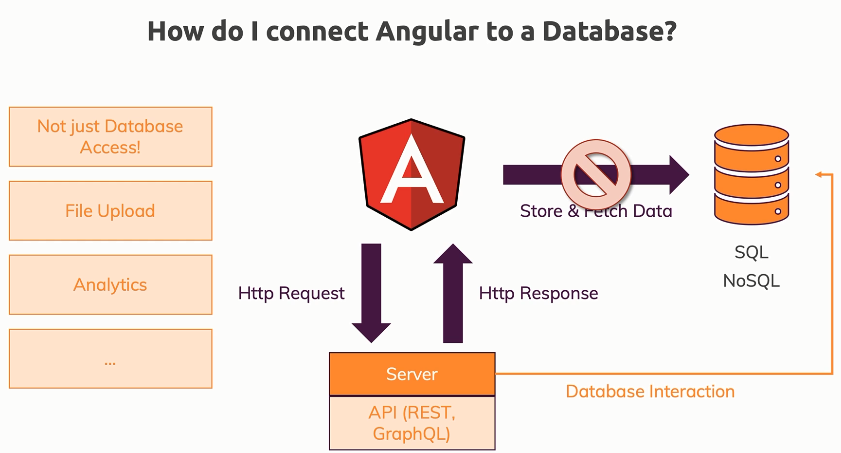
Result:  


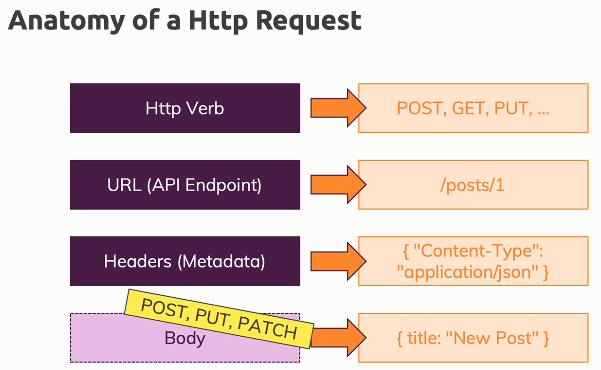
Result without async:



# Section 18: Making Http Requests

Note about the IDE used in the course. We have been working until now with [WebStorm](https://www.google.com/search?client=firefox-b-d&sxsrf=ALeKk01rFE4zu2eXmkHaKiAXW2_BvitcnA:1586504088197&q=angular+WebStorm&stick=H4sIAAAAAAAAAOOQU-LSz9U3SC8rSikzjhJISi0uUUjMSy_NSSxSyExJPcWIJH2KkRvEMTQ2qSgvyoZKJZllGFrkQTnZRtlZJqZQTnyFaYVx-ilGThAnrcyozOgRoy-3wMsf94SlXCatOXmN0Y6LKzw1yTGvuDy1qFjIgIsrsDS1qNI5J7G4WEiJS0CKx8czOCQ-2NXV29PPXYNRio8LRYTnF6OoY3JyflFKZl66Qkm-QkZicnaRXmb-LyYBmB_iXf1CPEMi4ycx62aUlBQUW-nrw1TpJ-Xkp-uDfKwLVa0L9LFuakpmSX7RIla4EUAnBgNFcgHaCNCSLAEAAA&sa=X&ved=2ahUKEwiN7vGkrN3oAhWGx4UKHRJ4A4wQxA0wAHoECAoQBA), however at the beginning of this module we change to Visual Studio Code, it seems it’s because Visual Studio have better options for http working.





## Backend (Firebase) Setup

Now let's dive into using Angular's HTTP features and for that, we need a back-end.

Now as I mentioned before, we'll not write our own server-side solution here because you would do that with a server-side language and not with Angular and therefore this is of course not the scope of this course.

By the way if you want to see how to build a complete Angular front-end and a custom back-end,

My Mean Course which you also find here on Udemy or on my page, academind.com, is a great place to go but here, we'll simply use Firebase. Firebase sounds like it's a database only, instead it's a whole back-end solution which gives us a REST API as well and it's a perfect solution for this course in this module because it's free to get started with and we can send requests there, we can send different types of request, we can easily see the changes we make through these requests and we can store data there and therefore, it's perfect.

Now you need a Google account for that, once you got that you can go to the console on firebase.com and there, simply click on add project. Now give that project any name you want and you can leave the other default settings and then check both checkmarks here.

Now this will create a new Firebase project for you and again, getting started here is free, you can of course check the pricing pages of Firebase to learn what else you can use for free and when it would start cost money.

Now once it's loaded, you'll see an interface that looks something like this and here, we'll go to database. Now again, Firebase is not just a database, it's a complete back-end solution that also offers the database but it's not just a database, it's not an alternative to MongoDB or anything like that, it's more than a database, it's a complete back-end service. Now here, we'll not use Cloud Firestore but instead if you scroll down, we'll use that real-time database, it's a bit easier to use and gives us a nicer visual feedback. So click on create database here and important, start in test mode, that's important.

Later, we will add authentication but for now we'll use that test mode to be able to send requests and work with data without being restricted.

Normally, you are of course protecting against unwanted access, here we are temporarily disabling this so that we can interact with that without issues. Now and that is our basic setup we'll need, the URL you see here will already be the URL you can send requests to.

So let's do that and let's make sure that we now use that to store data by sending a post request.

## Sending a POST Request

In this example, let's say go with creating a new post request, a simple one just to start.

1. To use the Angular http functionality we have to include in the app.module.ts the HttpClientModule in the import section. This now unlocks the client Angular offers in your whole project.
2. Now we create in our constructor a field of type HttpClient to do the request.
3. In post method we pass the url, and the body we want to send (so the data we want to store). Now, very important, you normally send JSON data when interacting with a RESTful API and actually, that will happen here as well but the Angular HttpClient will take our Javascript object here and automatically convert it to JSON data for us. So it will still send JSON data and we'll see this in a second when we inspect that request in the browser developer tools but Angular will transform this automatically for us.
4. Angular heavily uses observable as you already learned and HTTP requests are also managed via observable because they are a perfect scenario or a use case for observables. We can wrap them and we can then subscribe to them to get informed about the response and to handle errors and so on. So we have to use subscribe to receive the answers.   
   Note: how this observable is created by Angular we don’t have to unsubscribe, he does for us.

app.module.ts

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

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule, FormsModule, HttpClientModule],

  providers: [],

  bootstrap: [AppComponent]

})

app.component.ts

import { HttpClient } from '@angular/common/http';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

  loadedPosts = [];

  constructor(private http: HttpClient) {}

  onCreatePost(postData: { title: string; content: string }) {

    // Send Http request

    this.http

      .post(

        'https://backend-angularcourse.firebaseio.com/posts.json',

        postData

      )

      .subscribe(responseData => {

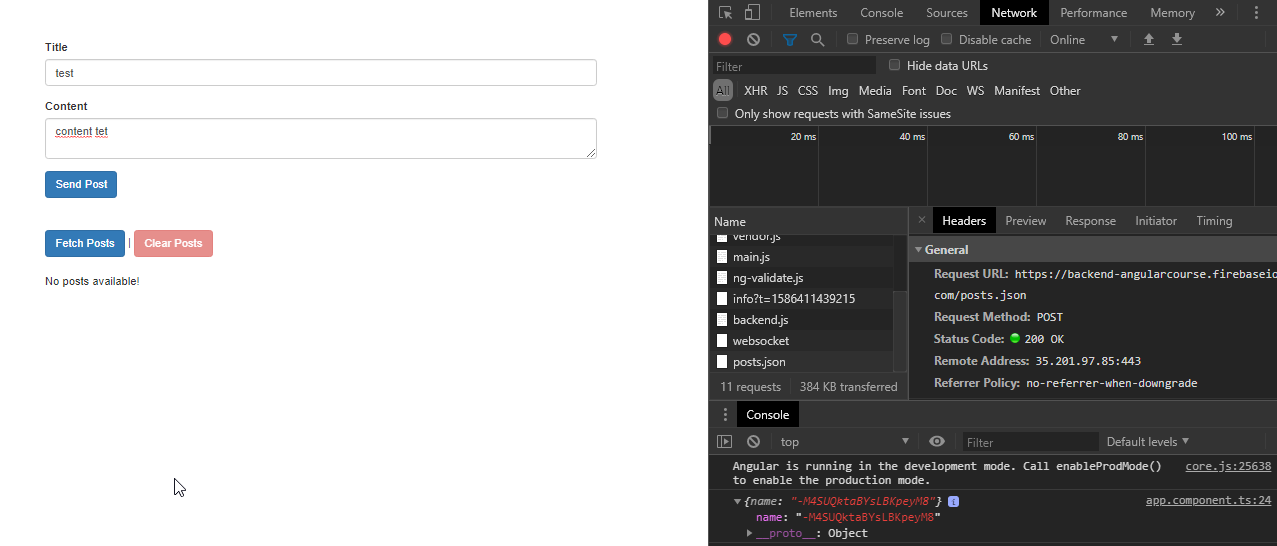
        console.log(responseData);

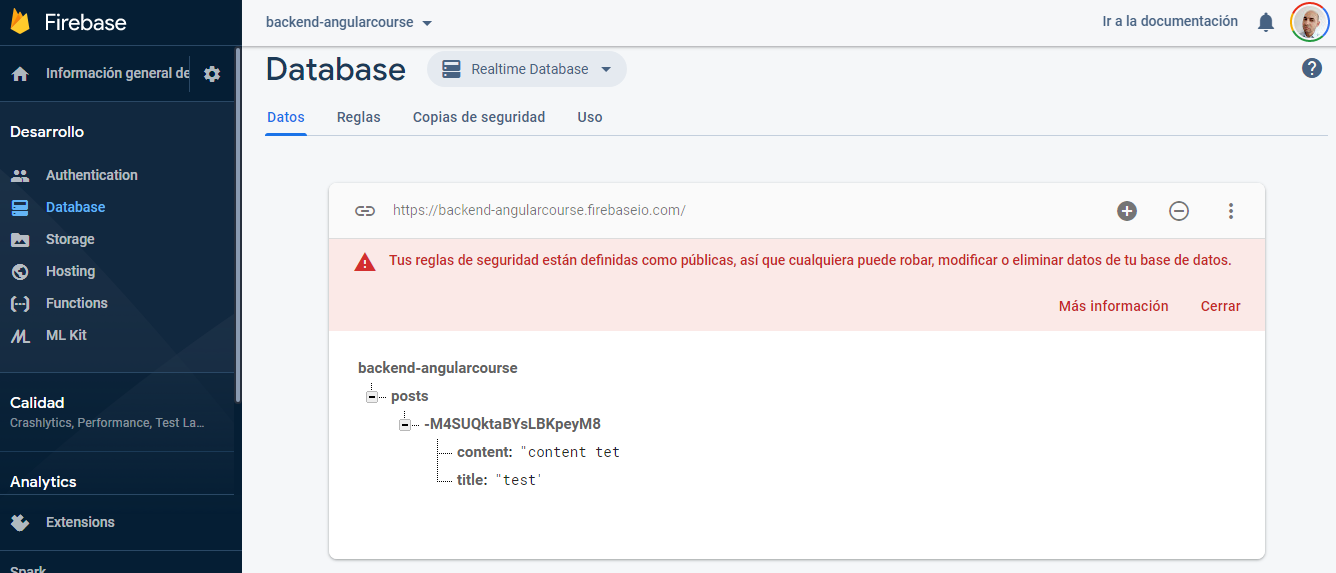
      });

  }

Note: Now on a real REST APIs you were communicating with, you would have clearly defined endpoints, like /posts/add or something like this, to which you have to send your request and the official docs of the API you're working with would tell you which API endpoint to send the request to. For Firebase, it's a bit differently, you have this starting point URL here and then you can add your own segments after that and this will get replicated as folders in Firebase (keep in mind that you never communicate directly with a database from inside your Angular app).

So let's say here, we want to have a node, a folder so to say named posts, well then we would simply add posts as a segment here and important, for Firebase you need to add .json, this simply is a Firebase requirement, it's not an Angular requirement, not a REST API requirement, only a requirement by that Firebase REST API.





## GETting Data

So requests are sent with the post method and we do only send them when we subscribe. Let's now add another request that actually gets us all the posts created.

We pass the same url of the post because we want to retrieve that information. Therefore, we're preparing this get request and now since this is a get request, this also needs no second argument because get requests have no request body because you're not sending any data, you're instead just requesting data. One thing you need to do here too however is you need to subscribe, no subscription, no request. So you need to subscribe here and we should get back our posts here, that's the hope.

We have put in the example the get inside a private method because we can now call this from two different places. E.g.

export class AppComponent implements OnInit {

  loadedPosts = [];

  constructor(private http: HttpClient) {}

  ngOnInit() {

    this.fetchPosts();

  }

  onCreatePost(postData: { title: string; content: string }) {

    // Send Http request

    this.http

      .post(

        'https://backend-angularcourse.firebaseio.com/posts.json',

        postData

      )

      .subscribe(responseData => {

        console.log(responseData);

      });

  }

  onFetchPosts() {

    this.fetchPosts();

  }

  private fetchPosts() {

    this.http.get('https://backend-angularcourse.firebaseio.com/posts.json')

    .subscribe(posts => {

      console.log(posts);

    });

  }

}

## Using RxJS Operators to Transform Response Data

Transforming data is of course something we could also do here inside of subscribe and that would generally not be a problem but it is a good practice to use **observable operators** because it simply allows us to write cleaner code with different steps we funnel our data through that can easily be swapped or adjusted.

Therefore here before we subscribe, we can now call pipe because pipe as you learned is a method that allows you to funnel your observable data through multiple operators before they reach the subscribe method.

Now the operator I need here is the **map operator** and therefore, I will import map from rxjs/operators. The map operator allows us to get some data and return new data which is then automatically re-wrapped into an observable so that we can still subscribe to it.

e.g.

  private fetchPosts() {

    this.http.get('https://backend-angularcourse.firebaseio.com/posts.json')

    .pipe(

      map( responseData => {

        const postArray = [];

        for(const key in responseData) {

          if(responseData.hasOwnProperty(key)) {

            postArray.push({...responseData[key], id: key});

          }

        }

        return postArray;

      })

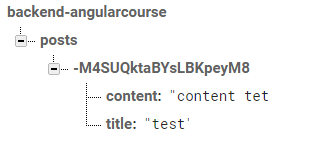
    )

    .subscribe(posts => {

      console.log(posts);

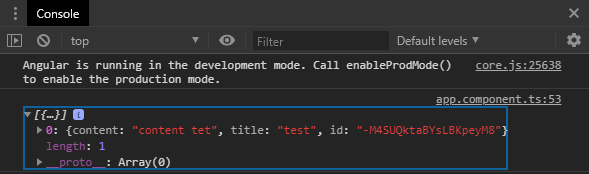
    });

  }



As we can see about firebase store our information using a key, so we use a “for” to moving for all the keys (now we only have one). After that we use the function *hasOwnProperty* that determines whether an object has a property with the specified name. If exists, then we push in our new array and object with the information in the response (we use spread operator … to put all elements in the array), we also put a field called id in our object with the key of firebase because we probably going to need it in the future to delete or update the element in firebase.

The result in the explorer.



## Using Types with the HttpClient

We can improve our last code explicitly telling to the http request (post, get), what value they going to receive. We can do this using angled brackets in the http request (post, get, put, etc.) and between the angled brackets, you store the type which this response will actually return as a body once it's done. So it's the response body type which is stored here and that will then automatically be handled by the Angular HttpClient and TypeScript understands this and now knows that the response data will have this type format. E.g.

post.model.ts (the model object where we going to store data)

export interface Post {

  title: string;

  content: string;

  id?: string;  //the symbol ? means that this paramter is optional

}

app.component.ts

  onCreatePost(postData: { title: string; content: string }) {

    // Send Http request

    this.http

      .post<{ name: string}>('https://backend-angularcourse.firebaseio.com/posts.json',

        postData

      )

      .subscribe(responseData => {

        console.log(responseData);

      });

  }

  private fetchPosts() {

    this.http

    .get<{ [key: string]: Post }>('https://backend-angularcourse.firebaseio.com/posts.json')

    .pipe(

      map( responseData => {

        const postArray = [];

        for(const key in responseData) {

          if(responseData.hasOwnProperty(key)) {

            postArray.push({...responseData[key], id: key});

          }

        }

        return postArray;

      })

    )

    .subscribe(posts => {

      console.log(posts);

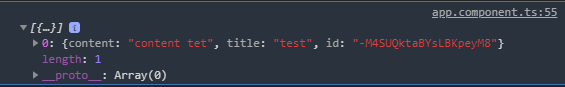
    });

  }

Remember that post return this:



And get return this:



## Using a Service for Http Requests

**How Http & Components Working Together ?**

The first alternative is that we use a subject in the post service where we next our posts when we got them and we subscribe to that subject in the app component and this would be a fine pattern, nothing wrong with that.

The alternative and maybe a bit more suited here because the subject would be perfect if we have multiple components interested in the posts, that's not the case here, so a simpler and for this scenario, a better solution would be to simply return the result of our get method (*fetchPost*) and that would be our observable.

So I don't want to subscribe in the service, instead I only return the prepared observable here in fetch posts. Now, I can and I have to subscribe in the app component. E.g.

app.component.ts

export class AppComponent implements OnInit {

  loadedPosts: Post[] = [];

  isFetching = false;

  constructor(private http: HttpClient, private postsService: PostsService) {}

  ngOnInit() {

    this.fetchingPost();

  }

  onCreatePost(postData: Post) {

    // Send http request

    this.postsService.createAndStorePost(postData.title, postData.content);

  }

  onFetchPosts() {

    this.fetchingPost();

  }

  private fetchingPost() {

    this.isFetching = true;

    this.postsService.fetchPost().subscribe( posts => {

      console.log(posts);

      this.isFetching = false;

      this.loadedPosts = posts;

    });

  }

}

post.service.ts

export class PostsService {

  constructor(private http: HttpClient) {}

  createAndStorePost(title: string, content: string) {

    const postData: Post = {title: title, content: content};

    // Send Http request

    this.http

      .post<{ name: string}>(

        'https://backend-angularcourse.firebaseio.com/posts.json', postData

      )

      .subscribe(responseData => {

        console.log(responseData);

      });

  }

  fetchPost() {

    return this.http

    .get<{ [key: string]: Post }>('https://backend-angularcourse.firebaseio.com/posts.json')

    .pipe(

      map( responseData => {

        const postArray = [];

        for(const key in responseData) {

          if(responseData.hasOwnProperty(key)) {

            postArray.push({...responseData[key], id: key});

          }

        }

        return postArray;

      })

    );

  }

}

Post.model.ts

export interface Post {

  title: string;

  content: string;

  id?: string;  //the symbol ? means that this paramter is optional

}

So now we moved the result handling so to say into the component but the more heavy lifting, the part detached from the template and from the UI which is the sending of the request and the transformation of the data, that now lives in the service and this is a best practice when working with Angular and HTTP requests. Move the part that is related to your template, which in my case here is managing the loading status and managing the loaded data, move that into the component and be informed about the result of your HTTP request by subscribing in the component but move the rest into the service and simply return the observable there so that you set up everything in the service but you can subscribe in the component.

Now of course I'm using a different pattern for creating a post, there I am subscribing in the service and this can be fine too. If your component doesn't care about the response and about whether the request is done or not, as it is the case here in our application, if the component doesn't care about it, then there is no reason to subscribe in the component, then you can just subscribe in the service as we're doing it here but if it does care about the response and the response status as it does for fetching posts, then having that service component split is great.

## Sending a DELETE Request

This is pretty similar, and the delete method have a lot of argument to inform you even about the progress. E.g.

app.component.ts

export class AppComponent implements OnInit {

  loadedPosts: Post[] = [];

  isFetching = false;

  constructor(private http: HttpClient, private postsService: PostsService) {}

  onClearPosts() {

    this.postsService.deletePost().subscribe( () => {

      this.loadedPosts = [];

    });

  }

}

post.service.ts

export class PostsService {

  constructor(private http: HttpClient) {}

  deletePost() {

    //this api of firebase will delete the entire elements stored

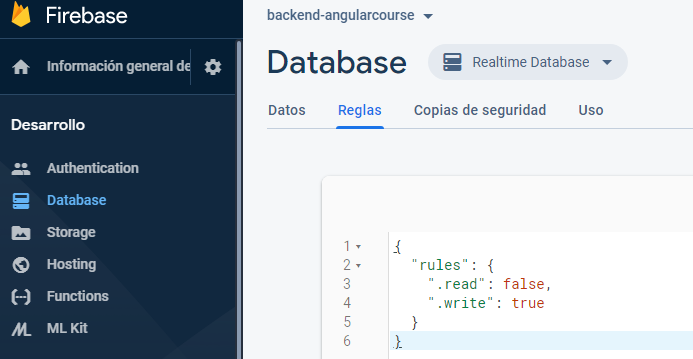
    return this.http.delete('https://backend-angularcourse.firebaseio.com/posts.json');

  }

}

## Handling Errors

Thus far our requests always succeeded but of course when interacting with servers, things can go wrong. You might not have an internet connection or you're sending a request with incorrect data somehow because you have a bug in your program or whatever it is or your server fails. Maybe it's offline or there's an error on the server or you're not authenticated, there are dozens of things that can go wrong.

For this example to simulate an error we going to firebase and set read permission of our api to false.  


Now how can we handle errors?

There are different ways of doing that. Let me demonstrate the first possible way, here onFetchPosts fails because we have an error. Thus far, we only passed one function to subscribe and that is the function that fires when new data is emitted.

Now as you learned in the observables section, you can pass more arguments to subscribe and the second argument is a function that triggers whenever an error is thrown and there, we will get the error object as an argument and now here, we can do something to handle that error, to do something to provide a better user interface or a better user experience. E.g.

app.component.html

      <div class="alert alert-danger" \*ngIf="error">

        <h1>An Error Ocurred !</h1>

        <p>{{ error }}</p>

      </div>

app.component.ts

  private fetchingPost() {

    this.isFetching = true;

    this.postsService.fetchPost().subscribe( posts => {

      console.log(posts);

      this.isFetching = false;

      this.loadedPosts = posts;

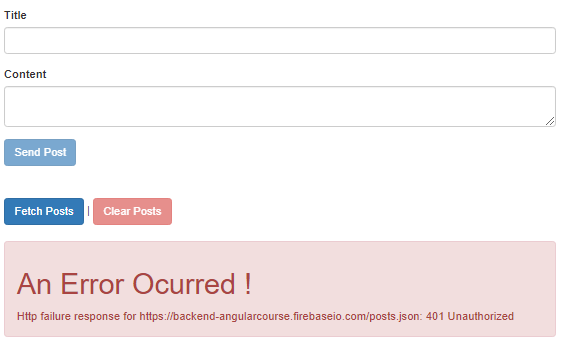
    }, error => {

      this.error = error.message;

      console.log(error);

    });

  }



### Using Subjects for Error Handling

This other way of handling errors could be interesting in cases like when you send a request and don’t subscribe to it in your component.

You could use a subject and that is especially useful if you have multiple places in the application that might be interested in your error. If in the service our post function return an error we emit with next an event that should be catch it for all components interest in that error. E.g.

Post.service.ts

export class PostsService {

  error = new Subject<string>();

  constructor(private http: HttpClient) {}

  createAndStorePost(title: string, content: string) {

    const postData: Post = {title: title, content: content};

    // Send Http request

    this.http

      .post<{ name: string}>(

        'https://backend-angularcourse.firebaseio.com/posts.json',

        postData

      )

      .subscribe(responseData => {

        console.log(responseData);

      }, error => {

        this.error.next(error.message);

      });

  }

app.component.ts

export class AppComponent implements OnInit, OnDestroy {

  loadedPosts: Post[] = [];

  isFetching = false;

  error: string = null;

  private errorSub: Subscription;

  constructor(private http: HttpClient, private postsService: PostsService) {}

  ngOnInit() {

    this.errorSub = this.postsService.error.subscribe( errorMessage => {

      this.error = errorMessage;

    });

    this.fetchingPost();

  }

  onCreatePost(postData: Post) {

    // Send http request

    this.postsService.createAndStorePost(postData.title, postData.content);

  }

  ngOnDestroy(): void {

    this.errorSub.unsubscribe();

  }

}

### Using the catchError Operator

No matter how you handle your error, what can be useful is a special operator that can assist you with handling errors and that special operator needs to be imported from rxjs/operators and it's named *catchError* and it does exactly what the names suggests.

Now let's say here when we *fetchPost()* in the service, where we already pipe some data, we got an error and we want to handle that. Now we can simply add the catch error operator here, we get our error response in here, so we get exactly the same data we would get in that second argument of the subscribe method and in here, you could now do stuff like send to analytics server or anything like that, so some generic error handling task you might want to do. Maybe not related to the UI, though of course you could use the subject and next the error message here too but maybe you have some behind the scenes stuff you want to do when an error occurs, log it somewhere, send it to your own server, your analytics server, anything like that.

And once you're done handing that error, you should pass it on though, it definitely needs to be able to reach subscribe, just as you need to pass something here in map as well.

So we also now need to create a new observable that wraps that error and for that, you can import *throwError*  from RxJS , that is a function that will yield a new observable and it yields a new observable by wrapping an error.

So here, we can now throw the error response, you could also throw a new custom error which you generated, whatever you want. Important, you need to return that observable which is created by throw error. And now with that, this of course doesn't do anything useful here but it's just an idea that you could consider using catch error if you have some generic error handling task you also want to execute. E.g

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

import { HttpClient } from '@angular/common/http';

import { map, catchError } from 'rxjs/operators';

import { Subject, throwError } from 'rxjs';

import { Post } from './post.model';

@Injectable({

  providedIn: 'root'

})

export class PostsService {

  error = new Subject<string>();

  constructor(private http: HttpClient) {}

  fetchPost() {

    return this.http

    .get<{ [key: string]: Post }>('https://backend-angularcourse.firebaseio.com/posts.json')

    .pipe(

      map( responseData => {

        const postArray = [];

        for(const key in responseData) {

          if(responseData.hasOwnProperty(key)) {

            postArray.push({...responseData[key], id: key});

          }

        }

        return postArray;

      }),

      catchError(errorRes => {

        //send to analytics server

        return throwError(errorRes);

      })

    );

  }

}

## Setting Headers

Now let me dive in the a little bit more advanced parts of the Angular HttpClient and how you can configure your requests and which kind of different responses you get back and so on. Let's start with headers - when sending an HTTP request.

Sometimes you also need to set some special headers, for example when you have a back-end that requires authorization and looks for an authorization header or if you want to set your own content type or you need to attach a custom header because your API you are sending the request to needs it.

Now setting your own headers is extremely simple, any HTTP request method, no matter if it's post or get or delete or patch or whatever, any of these methods has an extra last argument, so in the case of get it's the second argument, in the case of post it would be the third argument, which is an object where you can configure that request and there, you can configure a bunch of stuff as you can see and we'll start with the headers here and headers now takes a headers object which allows you to set the headers you want to send. To be precise, it's a *HttpHeaders* object which you need to import from @angular/common/http, so you need to add this import at the top and this allows you to create a new instance of this object with the new keyword and to this object, you can pass a Javascript object here with the object literal notation where you can have key-value pairs of your headers and headers are of course key-value pairs. E.g.

import { HttpClient, HttpHeaders } from '@angular/common/http';

export class PostsService {

  error = new Subject<string>();

  constructor(private http: HttpClient) {}

  fetchPost() {

    return this.http

    .get<{ [key: string]: Post }>(

      'https://backend-angularcourse.firebaseio.com/posts.json',

      {

        headers: new HttpHeaders({'Custom-Header': 'Hello'})

      }

    )

    .pipe(

      map( responseData => {

        const postArray = [];

        for(const key in responseData) {

          if(responseData.hasOwnProperty(key)) {

            postArray.push({...responseData[key], id: key});

          }

        }

        return postArray;

      }),

      catchError(errorRes => {

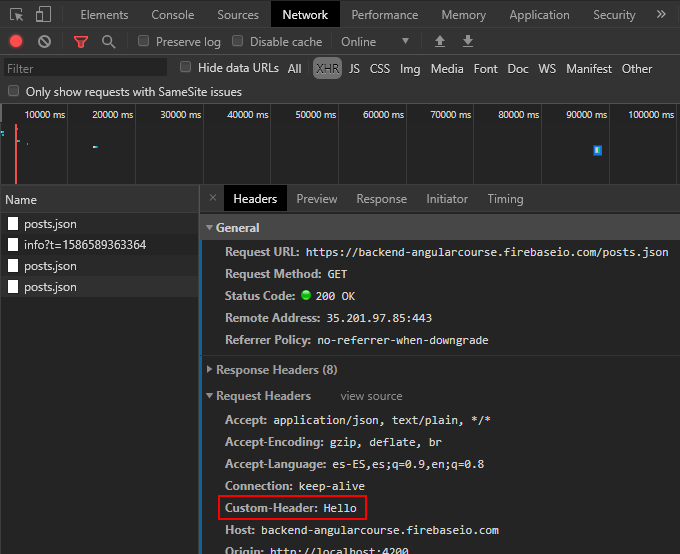
        //send to analytics server

        return throwError(errorRes);

      })

    );

  }



### Adding Query Params

Headers aren't the only thing you can set, for many API endpoints, you can also attach certain query parameters and of course again, it depends on the API endpoint you're sending the request to, which query parameters are supported.

For example, the Firebase REST API and you can learn that in the official docs, supports one query parameter which effect we can immediately see. You set parameters by adding the params key here in that same config object where you added headers and you set params equal to new HttpParams.

Now just as HTTP headers, you import HttpParams from @angular/common/http. Now HttpParams works a bit differently than HTTP headers, there you can call set and now you can set a param name and a value for that.

This query parameter attached to the end of the URL and of course you could have added this to the URL yourself up there too, you could have added print equals pretty up there and add, with an ampersand in-between, as many key value pairs as you need or as your API endpoint supports.

You can do that here in the URL but by using this params config, it's a bit more convenient. E.g.

Post.service.ts

  fetchPost() {

    let customParams = new HttpParams();//passing multiple parameters

    customParams = customParams.append('print', 'pretty');

    customParams = customParams.append('custom', 'key');

    return this.http

    .get<{ [key: string]: Post }>(

      'https://backend-angularcourse.firebaseio.com/posts.json',

      {

        headers: new HttpHeaders({'Custom-Header': 'Hello'}),

        //params: new HttpParams().set('print', 'pretty') passing one parameter example

        params:customParams

      }

    )

    .pipe(

      map( responseData => {

        const postArray = [];

        for(const key in responseData) {

          if(responseData.hasOwnProperty(key)) {

            postArray.push({...responseData[key], id: key});

          }

        }

        return postArray;

      }),

      catchError(errorRes => {

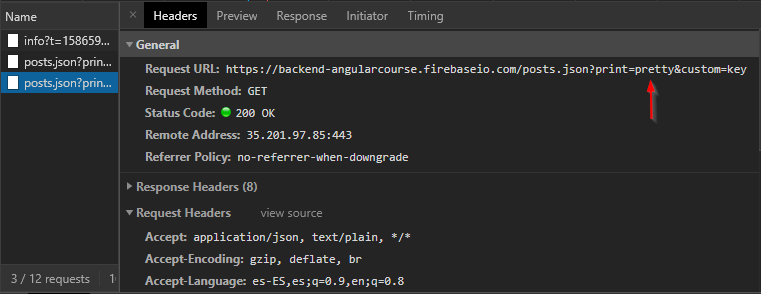
        //send to analytics server

        return throwError(errorRes);

      })

    );

  }



## Observing Different Types of Responses

Thus far, we always were only interested in the response data of our request. For example when we fetch the post, we get the response data, we map that and we return the subscription and then we use that data in the component.

Nothing wrong with that and absolutely what we need here but sometimes you need access to the entire response object and not just to the extracted body data, sometimes you need to find out which status code was attached or you need some response headers and in such cases, you can change the way the Angular HttpClient parses that response and you can basically tell Angular, "hey, please don't give me the unpacked, the extracted response data you found in the body, give me the full response instead". We do that specifying a new param in the request (observe: 'response'). E.g.

  createAndStorePost(title: string, content: string) {

    const postData: Post = {title: title, content: content};

    // Send Http request

    this.http

      .post<{ name: string}>(

        'https://backend-angularcourse.firebaseio.com/posts.json',

        postData,

        {

          observe: 'response'

        }

      )

      .subscribe(responseData => {

        console.log(responseData);

      }, error => {

        this.error.next(error.message);

      });

  }

In the browser:



By default the observe parameter will take { observe: 'body' }, and going to retrieve only the body element of the response.

Now observe could take another value { observe: 'events' }, now what is events?

Let's simply have a look at this events thing by using one other operator which we can be used inside a pipe, and that's the *tab* operator and we import that from rxjs/operators. Tab operator simply allows us to execute some code without altering the response, so that we basically just can do something with the response but not disturb our subscribe function and the functions we passed as arguments to subscribe.

If I now click clear posts, we see we get two outputs now and these outputs stem from post service line 73 which clearly is this console log (console.log(event);).

So it seems like we got two events here - the first one logs an empty object or an almost empty object where we have type 0 and the second one is the HTTP response object. In the end, you have different type of events and they are encoded with numbers.

However here in code, you don't have to use these numbers, you have a more convenient way. We can check event type and compare this to *HTTpEventType* which is an enum you can import from @angular/common/http. This is supported in TypeScript only and for Javascript, this in the end just is a map of numbers, so of zero and so on and it understands which type of event, so which number here stands for which internal type of event but here we have a more human readable way.

import { HttpClient, HttpHeaders, HttpParams, HttpEventType } from '@angular/common/http';

import { map, catchError, tap } from 'rxjs/operators';

deletePost() {

    //this api of firebase will delete the entire elements stored

    return this.http.delete('https://backend-angularcourse.firebaseio.com/posts.json',

      {

        observe: 'events'

      }

    ).pipe (

      tap (event => {

        console.log(event);

        if(event.type === HttpEventType.Sent) {

          //...

        }

        if(event.type === HttpEventType.Response) {

          console.log(event.body);

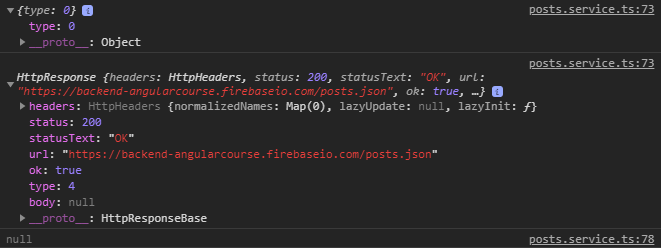
        }

      })

    );

  }

In the browser:



So these are events, obviously something you'll not use all the time but can be very helpful if you need a fine grained control over your request status or if you need very granular control over how you update the UI and in which phase your request currently is.

## Changing the Response Body Type

You can not only configure the observe mode here, you can also configure the response type. The default here is JSON, which means the response data, so the data in the body of your response is JSON and that tells Angular that it should automatically parse it and convert it to a Javascript object.

You could however also tell Angular the response side will be text and please keep it as text, don't try to convert it to a Javascript object. E.g.

  deletePost() {

    //this api of firebase will delete the entire elements stored

    return this.http.delete('https://backend-angularcourse.firebaseio.com/posts.json',

      {

        observe: 'events',

        responseType: 'text'

      }

    ).pipe (

      …

    );

  }

It is just important to also be aware of this option which you can use to tell Angular in case you are really getting back some text and you don't want Angular to parse it to a Javascript object or maybe you are getting back JSON but you still don't want to let Angular parse it because you want to parse it on your own at a later point of time or anything like that, for this you can set the response type. The default is JSON and most of the time, like in probably 99% of all cases, this is the type you want to keep. So typically, you don't need to overwrite that.

## Introducing Interceptors

Let's imagine we want to attach a custom header to all our outgoing requests and a more realistic scenario would be that you want to authenticate your user and you need to add a certain param or a certain header to every outgoing request therefore so that the back-end can read that, you don't want to manually configure every request because that is very cumbersome and for that, you can add interceptors.

Steps:

1. Basically you have to create a service that implements the HttpInterceptor interface that force you to override the intercept method.
2. So here, we get two arguments and the first one is a request object, which is of type HTTP request This is a generic type, a generic object, so you can use the angled brackets to basically inform Angular about the kind of data this request will yield and I want to write a generic interceptor here, so I will use any because this will work for any data.  
   The second argument which is passed to intercept and that is typically called *next*, *next* is a function you need to call to let the request continue its journey. The interceptor will basically run code before your request leaves your app, so you should have always this line (return next.handle(req);)
3. The import in the app module should be different. Add a new element to the providers array and that should be a Javascript object with three keys – values.
4. The first key is the provide key and there, you have to use **HTTP\_INTERCEPTORS**. This is the token by which this injection can later be identified by Angular, so it will basically know that all the classes you provide on that token, so by using that identifier, should be treated as HTTP interceptors and should therefore run their intercept method whenever a request leaves the application.
5. The second key you pass to that object is the use class key where you now point at your interceptor class you want to add as an interceptor.
6. And last but not least, you can have more than one interceptor and you inform Angular about that and that it should not replace the existing interceptors with this one by adding multi and setting this to true.

auth-interceptor.service.ts

import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http';

import { HttpEvent } from '@angular/common/http';

import { Observable } from 'rxjs';

export class AuthInterceptorService implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    console.log('Request is on its way');

    return next.handle(req);

  }

}

app.module.ts

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

import { AppComponent } from './app.component';

import { AuthInterceptorService } from './auth-interceptor.service';

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule, FormsModule, HttpClientModule],

  providers: [

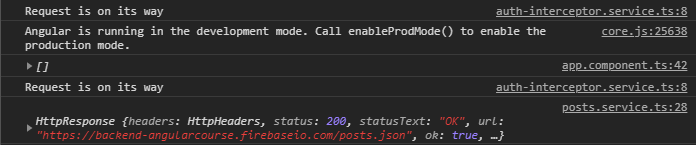
    { provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptorService, multi: true}

  ],

  bootstrap: [AppComponent]

})

Browser with two request done:



### Manipulating Request Objects

Inside of an interceptor, you can not only log data, you can also modify the request object.

However, the request object itself is immutable, so you can't set request URL to a new URL, that will not work and you also get an error here.Instead if you want to modify the request, you have to create a new one, like modified request sounds like a fitting name for the constant, where you call request clone and inside of clone, you pass in a Javascript object where you now can overwrite all the core things. In our example we only append a value to the header.

export class AuthInterceptorService implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    console.log('Request is on its way');

    console.log(req.url);

    const modifiedRequest = req.clone({

      headers: req.headers.append('Auth', 'xyz')

    });

    return next.handle(modifiedRequest);

  }

}

This of course is a typical use case for an interceptor, you change the request and then you forward that modified request and if you only wanted to append the auth header for certain URLs, then you could use an if check with the request URL key for example.

### Response Interceptors

You're also not limited to interacting with the request in an interceptor, you can also interact with the response.

You do this by adding something here to handle because handle actually gives you an observable, which I guess makes sense because in the end, your request is an observable to which you subscribe in the end.

So this in the end is the request with the response in it you could say, wrapped into an observable and therefore here, you can of course add pipe and do something with the response if you want to. You could add the map operator here, to change the response for example, of course you should be careful with that and not change it in a way that the rest of your application breaks or what I'll do here, you use *tap* which you need to import from rxjs/operators to simply look into the response. Now here in tap, you get an event and that's important, you always get events here.

Remember I showed you that you can observe different kinds of responses with the Angular HttpClient, no matter what you chose there, here in the interceptor, you always get an event so that you have the most granular access to the response you could possibly have.

So here for example, we could check if the event type is equal to the HTTP event type which you also need to import, of response, because here maybe I only care about that response, I will log the event before that however, so that we can see all events we got and then in here, we could say response arrived, body data and then simply console log the event body which here is the response body of course.

Now here again, I'm just doing some logging but as I just mentioned, you could use other operators like map here and even transform the response, that would be possible. E.g.

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    console.log('Request is on its way');

    console.log(req.url);

    const modifiedRequest = req.clone({

      headers: req.headers.append('Auth', 'xyz')

    });

    return next.handle(modifiedRequest).pipe(

      tap( event => {

        console.log(event);

        if(event.type == HttpEventType.Response) {

          console.log('Response arrived, body data: ');

          console.log(event.body);

        }

      })

    );

  }

### Multiple Interceptors

Providing multiple interceptors is extremely simple, you just have to add both to the app.module.ts file but the order matters here, because that will be the order in which they are executed.

In our example one interceptor is now only attaching the auth header and the other interceptor is responsible for logging, now we just need to add them both and now here it comes something super important, the order in which you provide them matters because that will be the order in which they are executed.

So in the app module here, of course we provide the other interceptor, the logging interceptor.

E.g.

auth-interceptor.service.ts

export class AuthInterceptorService implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    console.log('Request is on its way');

    const modifiedRequest = req.clone({

      headers: req.headers.append('Auth', 'xyz')

    });

    return next.handle(modifiedRequest);

  }

}

logging-interceptor.service.ts

export class LoggingInterceptorService implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    return next.handle(req).pipe(

      tap( event => {

        if(event.type == HttpEventType.Response) {

          console.log('Incoming response: ');

          console.log(event.body);

        }

      })

    );

  }

}

app.module.ts

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule, FormsModule, HttpClientModule],

  providers: [

    { provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptorService, multi: true},

    { provide: HTTP\_INTERCEPTORS, useClass: LoggingInterceptorService, multi: true}

  ],

  bootstrap: [AppComponent]

})

**Useful Resources:**

* Official Docs: <https://angular.io/guide/http>

# 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>

## bind()

El método **bind()** crea una nueva función, que cuando es llamada, asigna a su operador  *this* el valor entregado, con una secuencia de argumentos dados precediendo a cualquiera entregados cuando la función es llamada.

El valor de *this* es ignorado cuando la función es llamada con el operador *new*.

<https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Function/bind>

# 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/>