# 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. |
| **ng build --prod** | Iit generated a new dist folder. This folder control contains a folder with your project name. And then there you got multiple files optimized in size. This should be do before you want to deploy it somewhere. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**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"

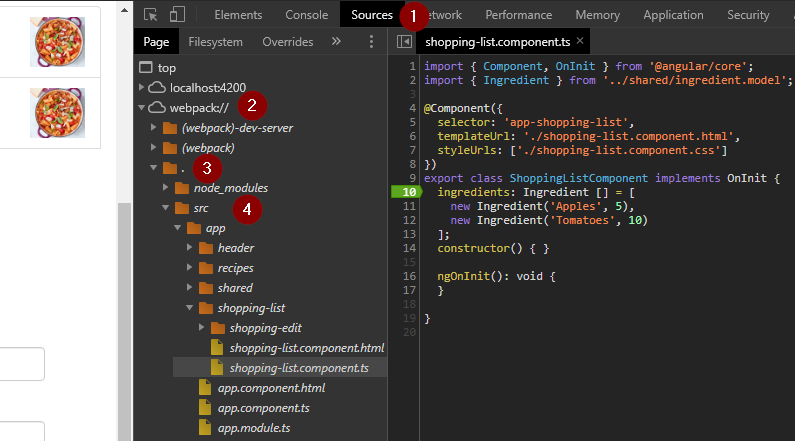
    [ngStyle]="{'background-color': getColor()}"

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

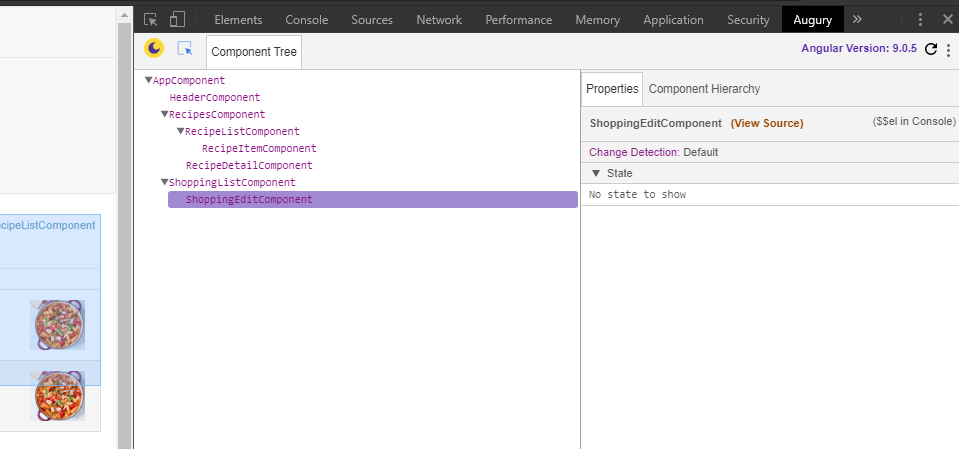
</div>

## Session 4

### How debug in Chrome

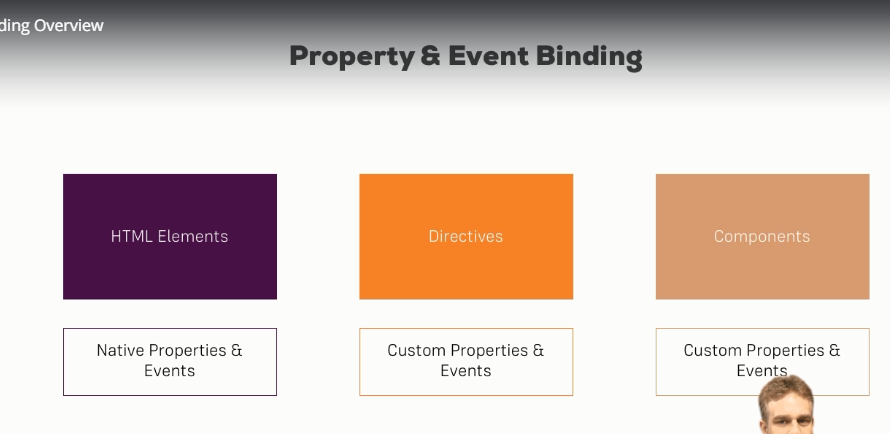


### Using Augury to Dive into Angular Apps



## 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>

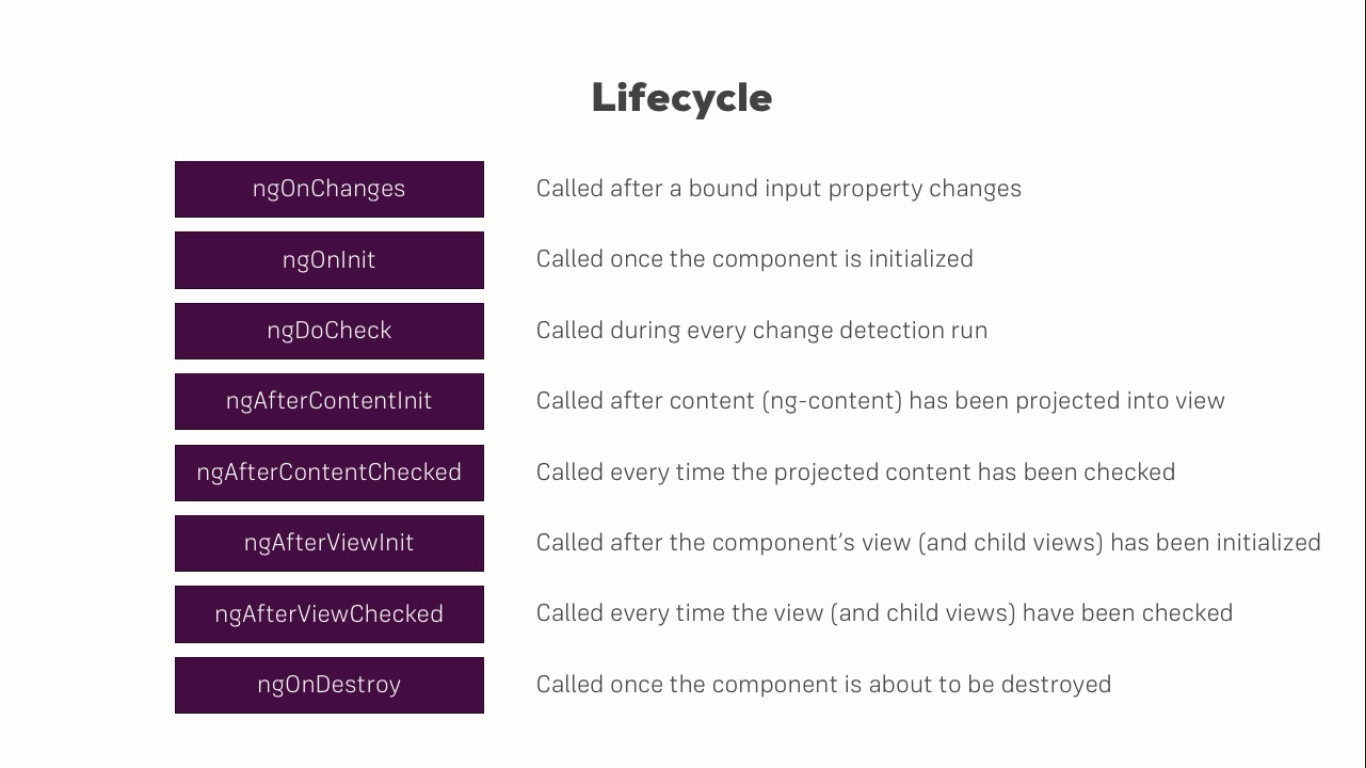
</p>

</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 {

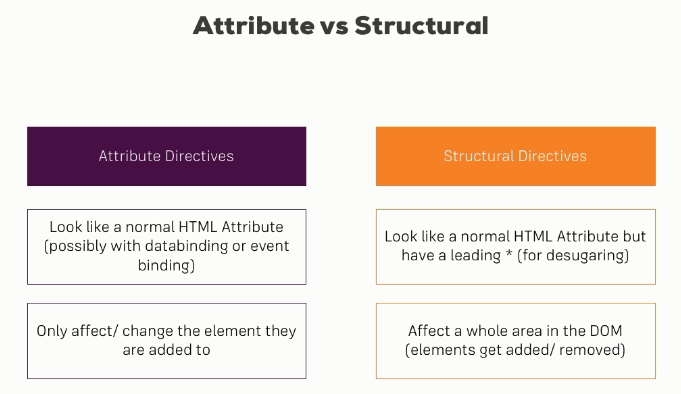
val: string = '';

}

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) {

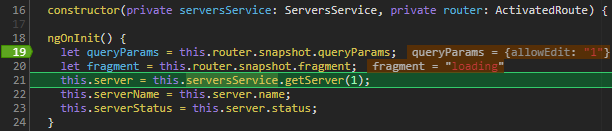
**this**.router.navigate(['/servers', id, 'edit'], {queryParams: {allowEdit: '1'}, fragment: 'loading'});

}

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]

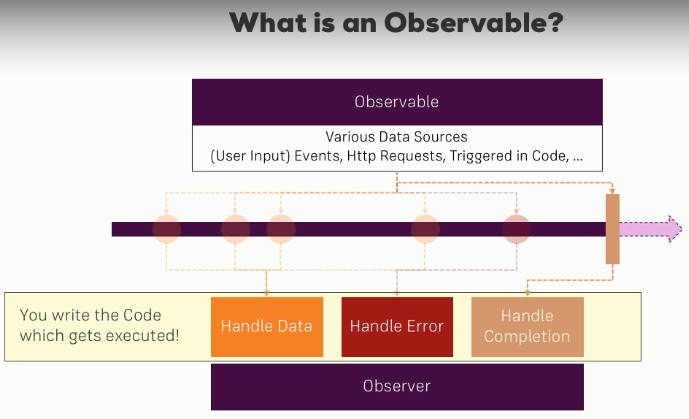
})

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 troubles of memory links. For that to the class that implemented the 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 click 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);

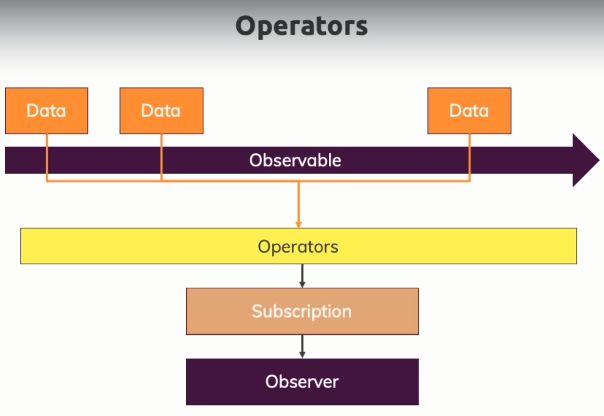
}, () **=>** {

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 {

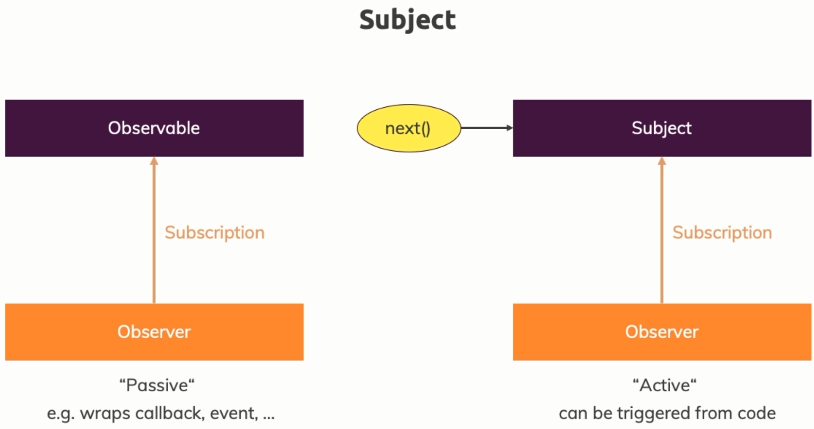
**this**.firstObsSubscription.unsubscribe();

}

}

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.

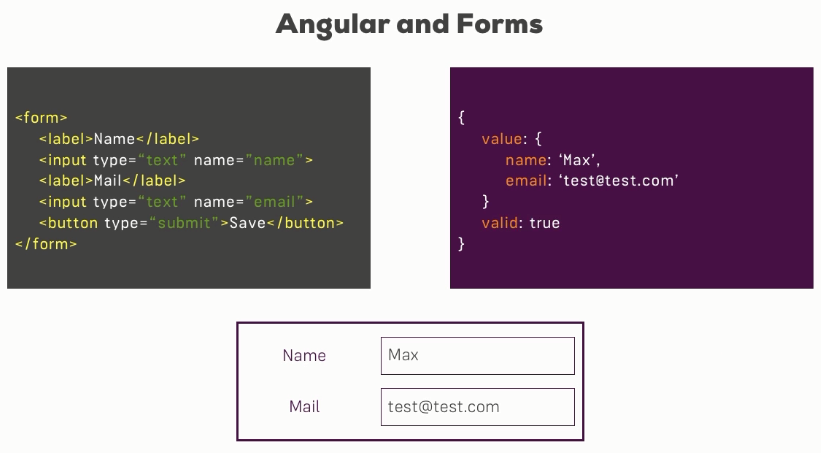
See an example in: Services communication with Subject (Observable).

**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

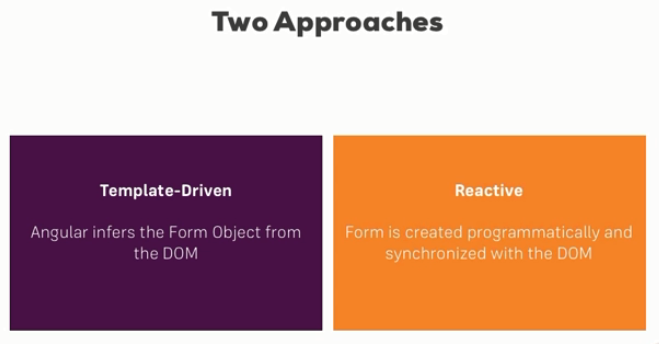
Angular help us to work with forms. This is what Angular does for you, it gives you actually a Javascript object representation of your form, making it simple for you to retrieve user values and to see the state of the form and to work with it.



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']

})

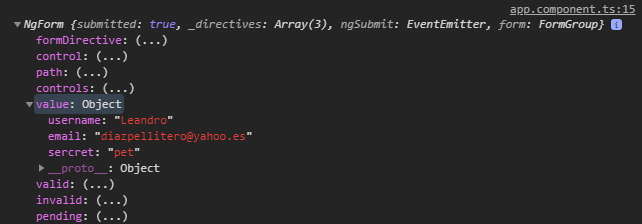
**export** **class** AppComponent {

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

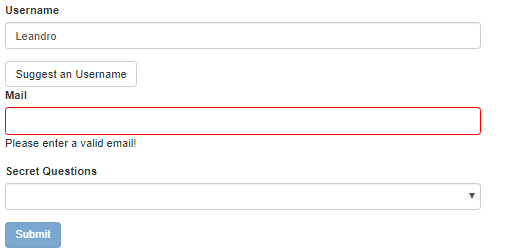
#email="ngModel">

<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">

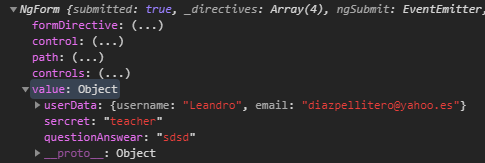
Please enter a valid email!

</span>

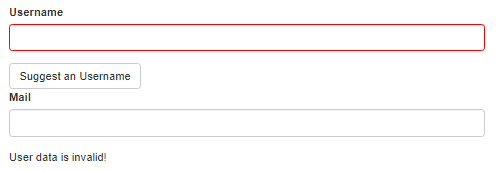
</div>

</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

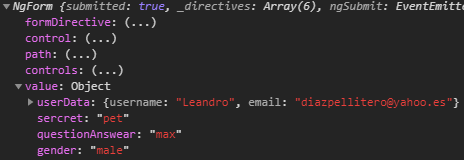
[value]="gender"

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"

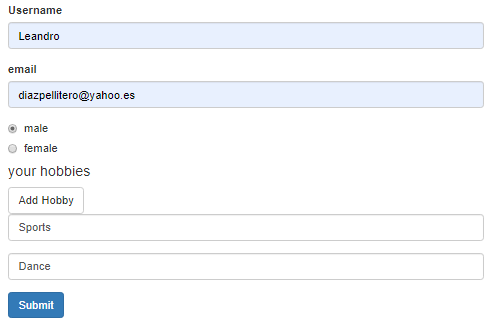
            (click)="onAddHobby()">Add Hobby</button>

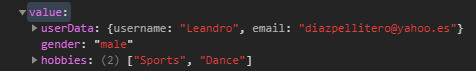
          <div class="form-group"   
 \*ngFor="let hobbyControl of getControls(); let i = index">

            <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 {

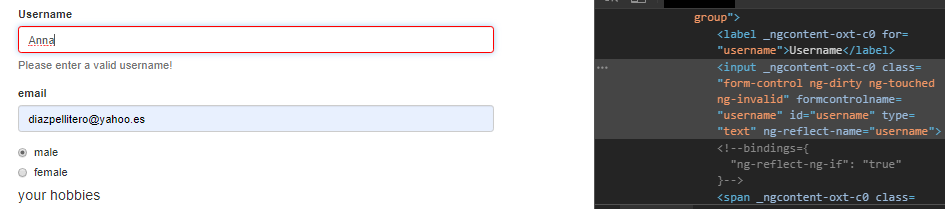
      return null;

    }

  }

}

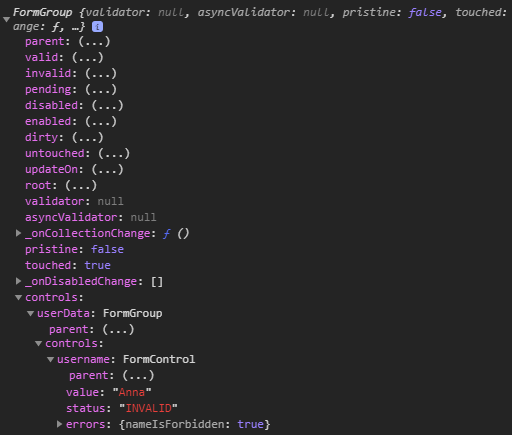
How if we type a name for user how is in the forbidden list, we show the invalid class.



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']">

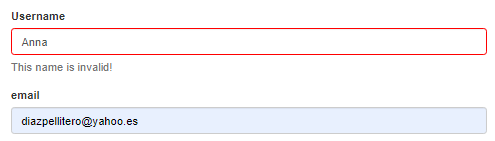
          This name is invalid!</span>

        <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)

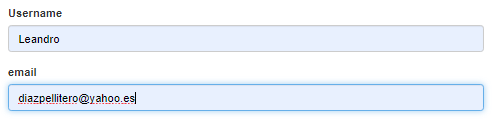
    // )

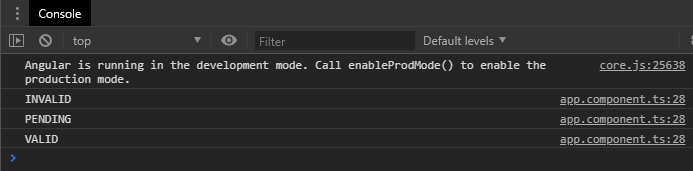
    this.signupForm.statusChanges.subscribe(

      (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'

      }

    });

  }

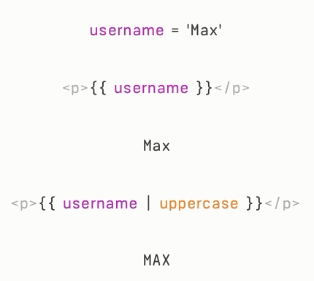
Also if we want to reset the form after submitting it, we can simply call reset here.

  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.



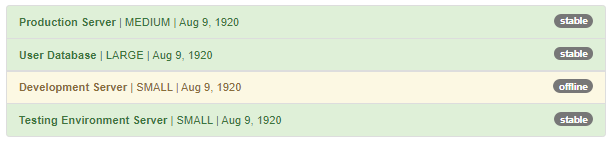
        <li>

          <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' }}



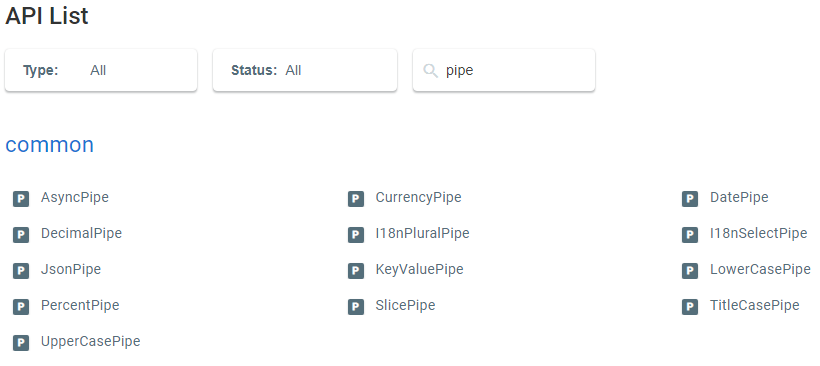
If you would have multiple parameters here, you simply separate them with additional colons.

## 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> |

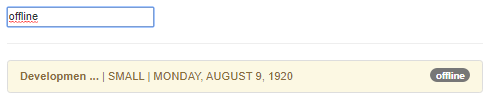
          {{ server.instanceType | uppercase }} |

          {{ 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">

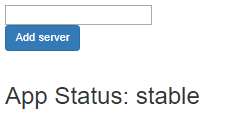
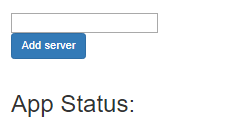
      <br>

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

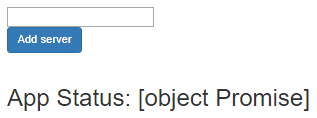
      <br><br>

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

      <hr>

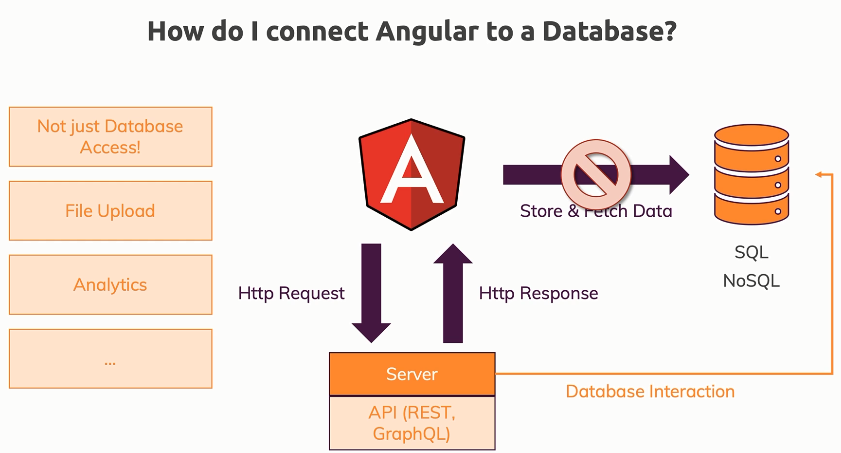
Result:  


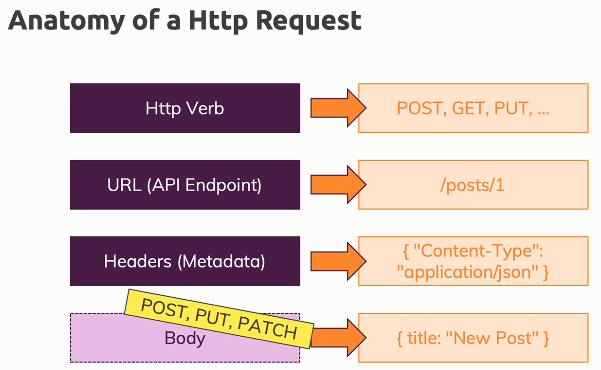
Result without async:



# 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 => {

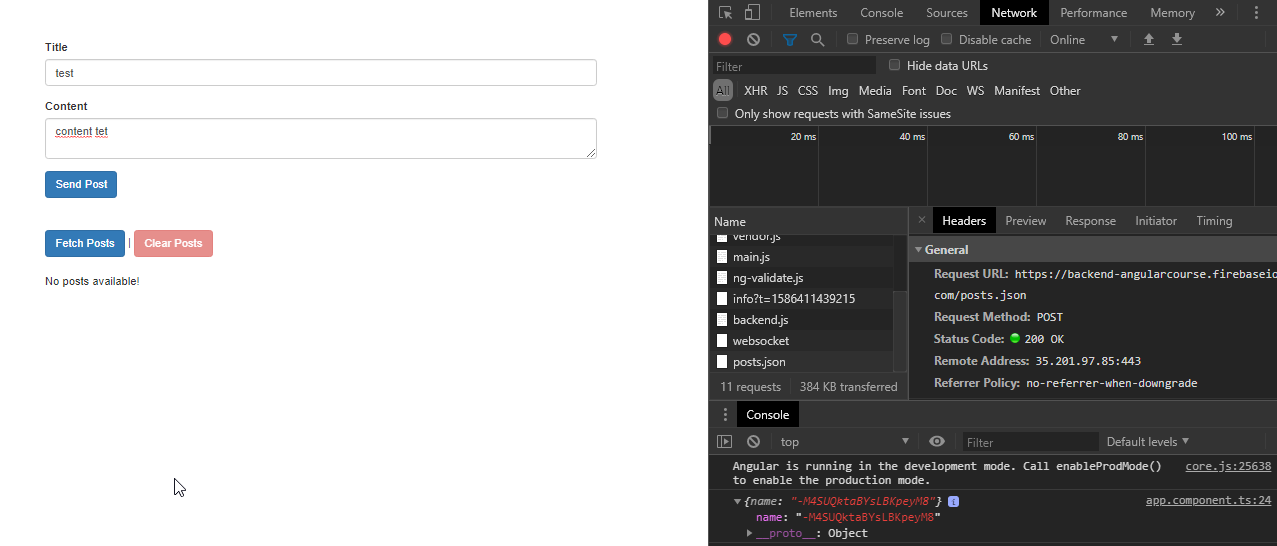
        console.log(responseData);

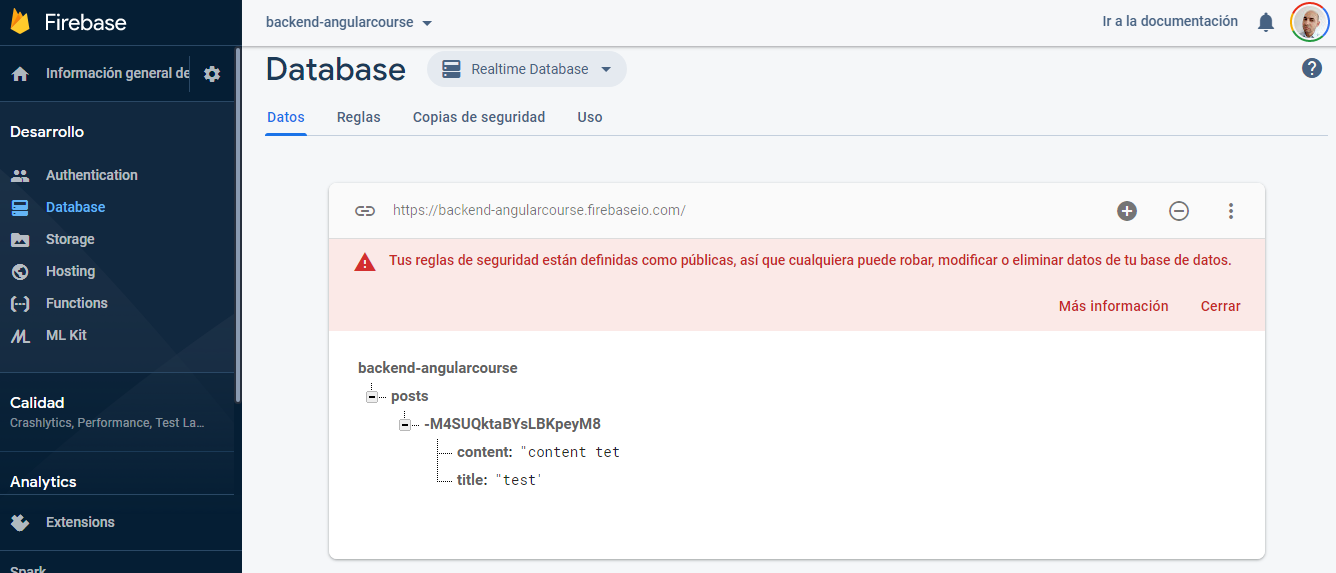
      });

  }

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;

      })

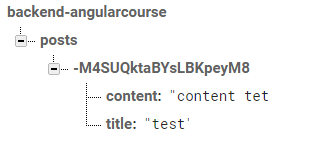
    )

    .subscribe(posts => {

      console.log(posts);

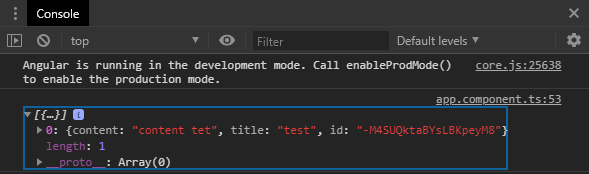
    });

  }



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);

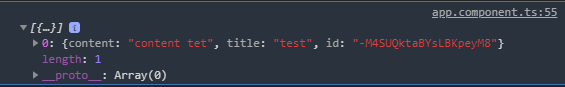
    });

  }

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

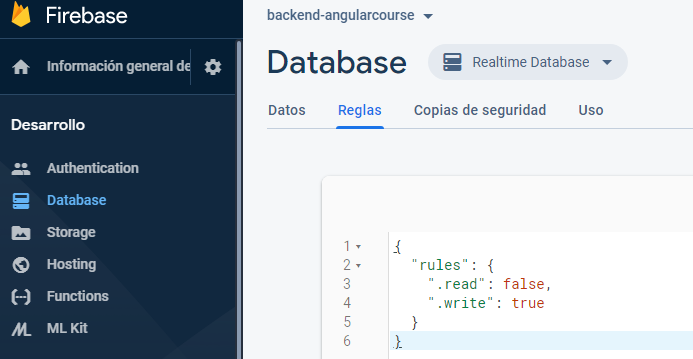
    return this.http.delete('https://backend-angularcourse.firebaseio.com/posts.json');

  }

}

## 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;

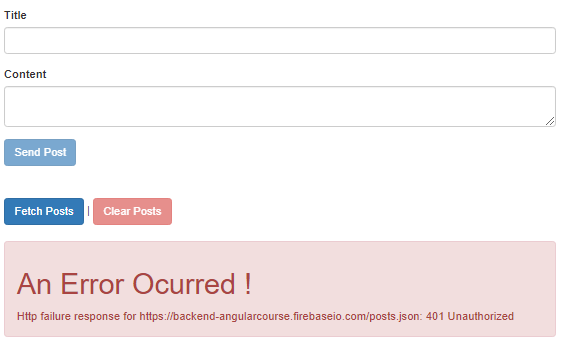
    }, error => {

      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 => {

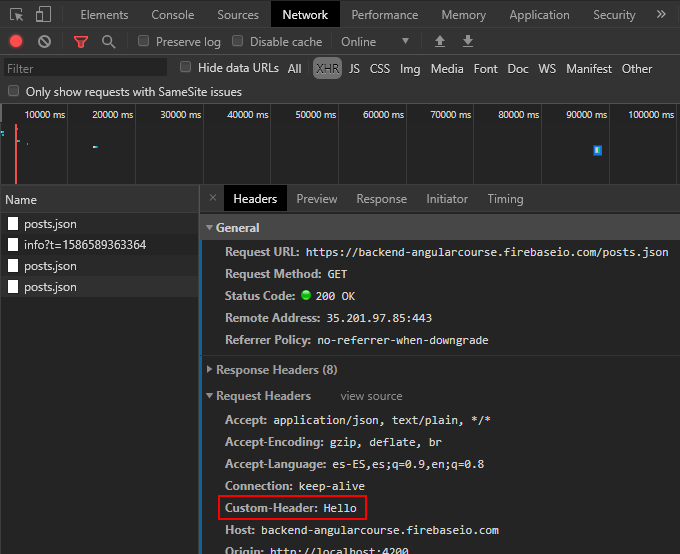
        //send to analytics server

        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 => {

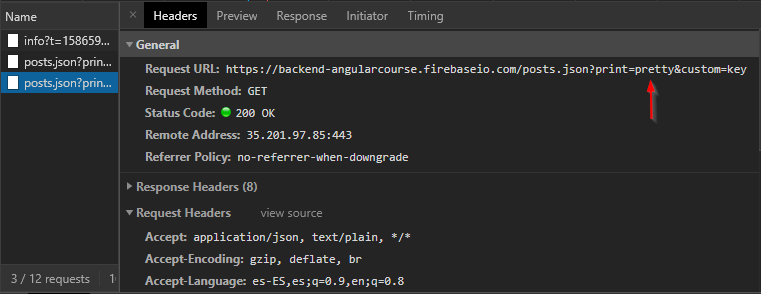
        //send to analytics server

        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);

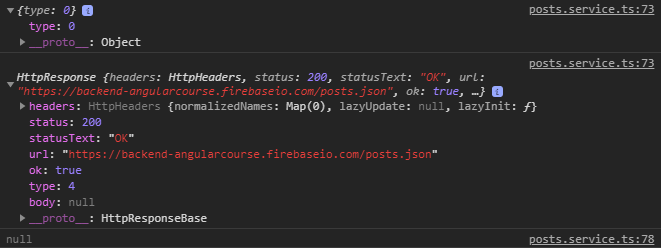
        }

      })

    );

  }

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: [

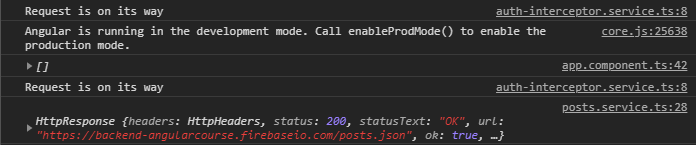
    { provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptorService, multi: true}

  ],

  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]

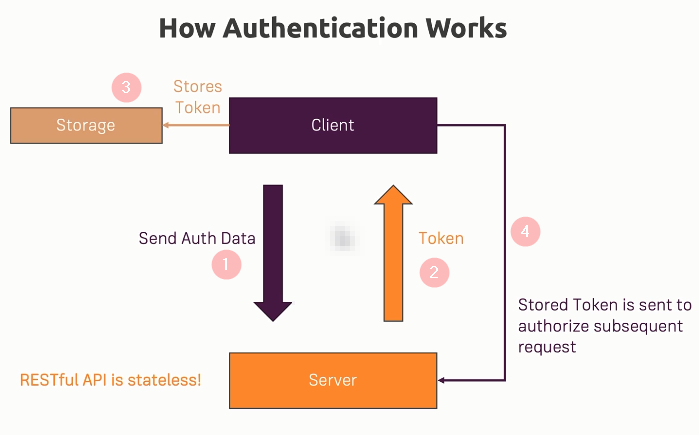
})

**Useful Resources:**

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

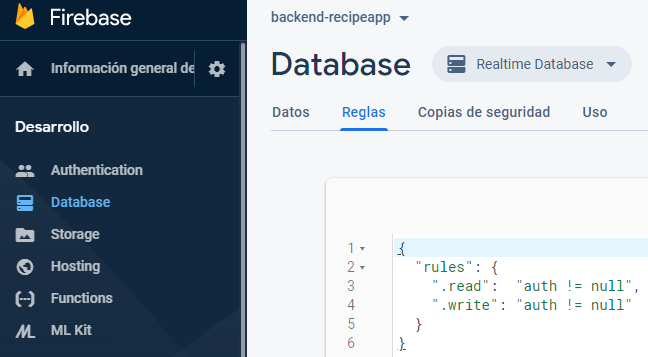
# Section 20: Authentication & Route Protection in Angular

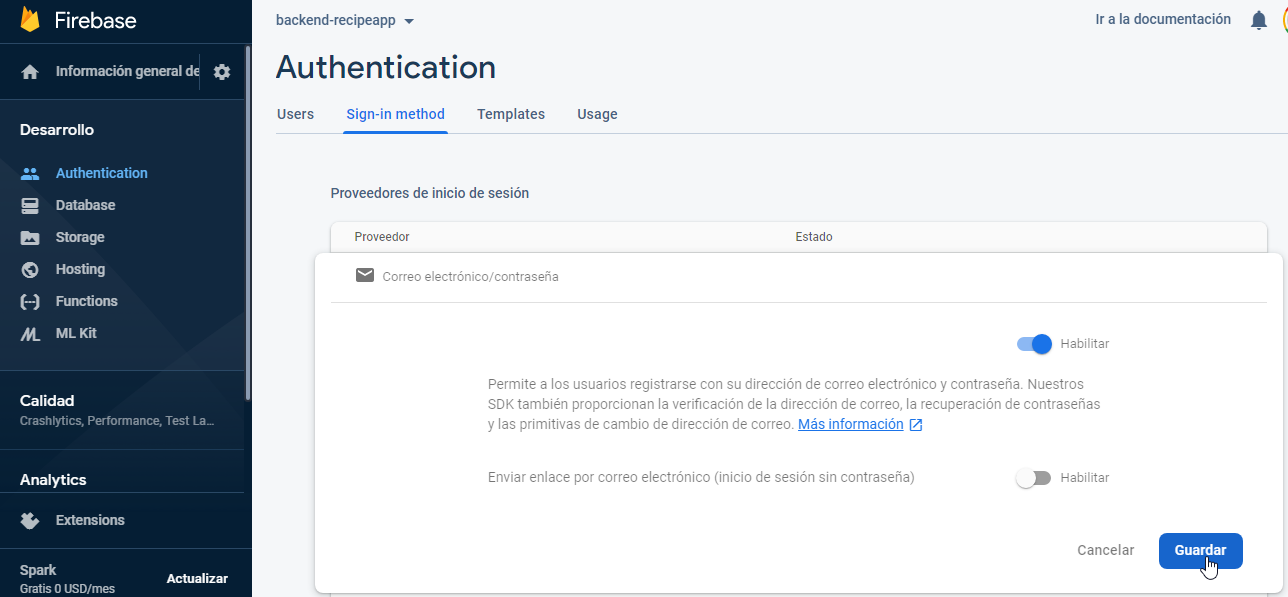
## How Authentication Works

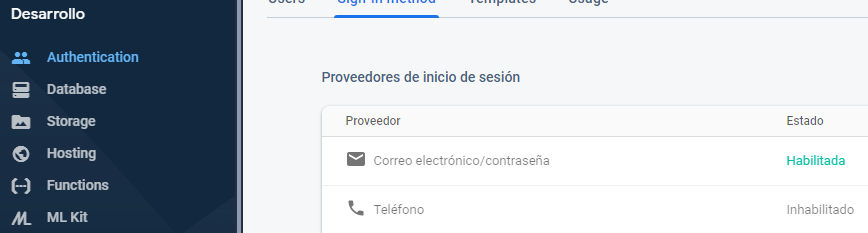


## Preparing the Backend to authentication

The fallowing changes have been done in the backend to enable the authentication in our app.



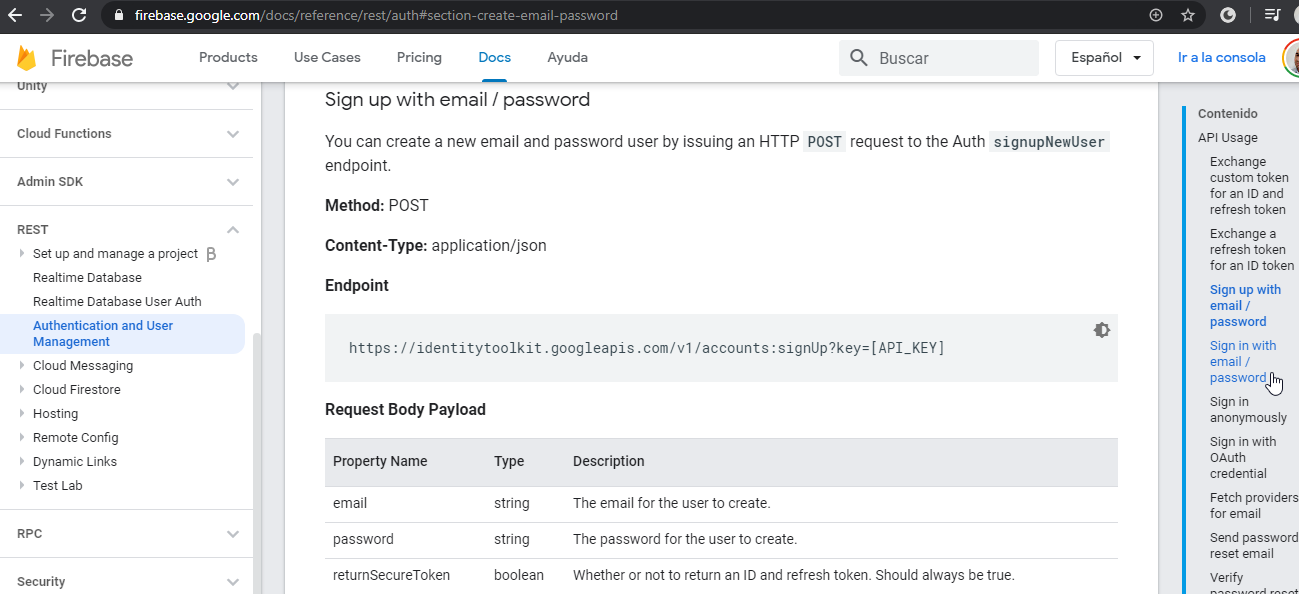




## Preparing the Signup Request

If we search in google: “firebase auth rest api” we can find this site with information about the API we going to use to our project.

<https://firebase.google.com/docs/reference/rest/auth>



1. **Sign up with email / password**

You can create a new email and password user by issuing an HTTP POST request to the Auth signupNewUser endpoint.

**Method:** POST

**Content-Type:** application/json

**Endpoint**

https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=[API\_KEY]

**Request Body Payload**

|  |  |  |
| --- | --- | --- |
| Property Name | Type | Description |
| email | string | The email for the user to create. |
| password | string | The password for the user to create. |
| returnSecureToken | boolean | Whether or not to return an ID and refresh token. Should always be true. |

**Response Payload**

|  |  |  |
| --- | --- | --- |
| Property Name | Type | Description |
| idToken | string | A Firebase Auth ID token for the newly created user. |
| email | string | The email for the newly created user. |
| refreshToken | string | A Firebase Auth refresh token for the newly created user. |
| expiresIn | string | The number of seconds in which the ID token expires. |
| localId | string | The uid of the newly created user. |

In our project we are created a service to do the sign up. In this service we use the Endpoint of the exposing API but we have to substitute the [API\_KEY] URL parameter with the one of our project. E.g.  
auth.service.ts

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

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

interface AuthResponseData {

  idToken: string;

  email: string;

  refreshToken: string;

  expiresIn: string;

  localId: string;

}

@Injectable({ providedIn: 'root' })

export class AuthService {

  constructor(private http: HttpClient) { }

  signup(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=AIzaSyDKNsEn6rrkJE7XUMh2R3e74OW8YfwUh7k',

      {

        email: email,

        password: password,

        returnSecureToken: true

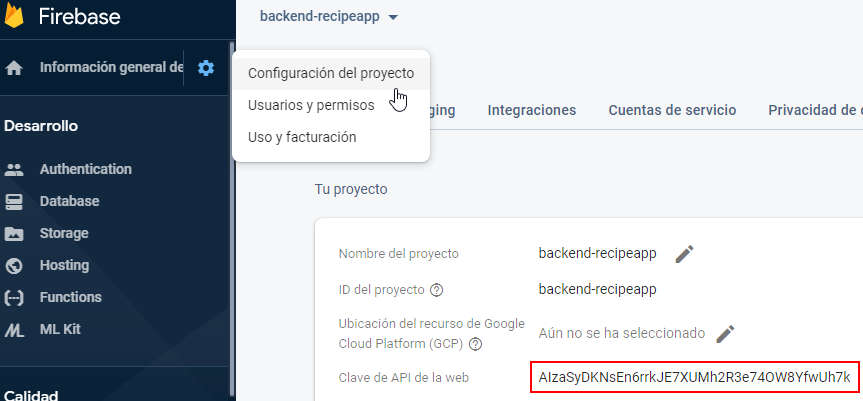
      }

    )

  }

}

The [API\_KEY] URL parameter of our project can be found in:



Now with our auth component we can do the sign-up. E.g.

auth.component.html

<div class="row">

  <div class="col-xs-12 col-md-6 col-md-offset-3">

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

      <div class="form-group">

        <label for="email">E-mail</label>

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

          ngModel name="email" required email>

      </div>

      <div class="form-group">

        <label for="password">Password</label>

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

          ngModel name="password" required minlength="6">

      </div>

      <div>

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

[disabled]="!authForm.valid">

          {{ isLoginMode ? 'Login' : 'Sign Up'}}</button> |

        <button class="btn btn-primary" type="button" (click)="onSwitchMode()">

          Switch to {{ isLoginMode ? 'Sign Up' : 'Login'}}</button>

      </div>

    </form>

  </div>

</div>

auth.component.ts

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

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

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

@Component({

  selector: 'app-auth',

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

  styleUrls: ['./auth.component.css']

})

export class AuthComponent implements OnInit {

  //this variable control if we are in login mode or signup mode

  isLoginMode = true;

  constructor(private authService: AuthService) { }

  onSwitchMode() {

    this.isLoginMode = !this.isLoginMode;

  }

  onSubmit(form: NgForm) {

    if(!form.valid) {

      return;

    }

    const email = form.value.email;

    const password = form.value.password;

    if(this.isLoginMode){

      //...

    } else {

      this.authService.signup(email, password).subscribe(

        resData => {

          console.log(resData);

        },

        error => {

          console.log(error);

        }

      );

    }

    form.reset();

  }

}

Note: I have created in firebase the following credentials:

user: [test@test.com](mailto:test@test.com)

pass: 123456



### Adding a Loading Spinner & Error Handling Logic

Let's start with the loading spinner and for this, you can enter CSS loading spinners in Google to find <https://loading.io/css/> which is a page where you find a lot of beautiful loading spinners.

Now we create a component for the spinner and we copy from the spinner page the CSS and HTML of the spinner we want. We copy in the CSS the spinner code and there is no need for an HTML page in this case E.g. of my component created.

loading-spinner.component.ts

@Component({

  selector: 'app-loading-spinner',

  template: '<div class="lds-dual-ring"></div>',

  styleUrls: ['./loading-spinner.component.css']

})

export class LoadingSpinnerComponent {}

Now we used in out auth component. Also we add error handler to our component.

auth.component.html

<div class="row">

  <div class="col-xs-12 col-md-6 col-md-offset-3">

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

      <p>{{ error }}</p>

    </div>

    <div \*ngIf="isLoading" style="text-align: center;">

      <app-loading-spinner></app-loading-spinner>

    </div>

    <form #authForm="ngForm" (ngSubmit)="onSubmit(authForm)" \*ngIf="!isLoading">

      <div class="form-group">

        <label for="email">E-mail</label>

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

          ngModel name="email" required email>

      </div>

      <div class="form-group">

        <label for="password">Password</label>

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

          ngModel name="password" required minlength="6">

      </div>

      <div>

        <button class="btn btn-primary" type="submit" [disabled]="!authForm.valid">

          {{ isLoginMode ? 'Login' : 'Sign Up'}}</button> |

        <button class="btn btn-primary" type="button" (click)="onSwitchMode()">

          Switch to {{ isLoginMode ? 'Sign Up' : 'Login'}}</button>

      </div>

    </form>

  </div>

</div>

auth.component.ts

export class AuthComponent implements OnInit {

  //this variable control if we are in login mode or signup mode

  isLoginMode = true;

  isLoading = false;

  error: string = null;

  constructor(private authService: AuthService) { }

  onSwitchMode() {

    this.isLoginMode = !this.isLoginMode;

  }

  onSubmit(form: NgForm) {

    if(!form.valid) {

      return;

    }

    const email = form.value.email;

    const password = form.value.password;

    this.isLoading = true;

    if(this.isLoginMode){

      //...

    } else {

      this.authService.signup(email, password).subscribe(

        resData => {

          console.log(resData);

          this.isLoading = false;

        },

        errorMessage => {

          console.log(errorMessage);

          this.error = errorMessage;

          this.isLoading = false;

        }

      );

    }

    form.reset();

  }

}

Despite we subscribe to our response in the component auth.component.ts, an not in the service, for error handler, it is better handler the error in the service, and just receive the error in the component, in that way the logic its encapsulated in the service. So:

auth.service.ts

  signup(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=AIzaSyDKNsEn6rrkJE7XUMh2R3e74OW8YfwUh7k',

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(errorRes => {

        let errorMessage = 'An unknown error occurred!';

        if (!errorRes.error || !errorRes.error.error) {

          return throwError(errorMessage);

        }

        switch (errorRes.error.error.message) {

          case 'EMAIL\_EXISTS':

            errorMessage = 'The email address is already in used.'

        }

        return throwError(errorMessage);

      })

    );

  }

## Sending Login Requests

Sending the login request it’s pretty similar to signup request, we have to call another API that we can find in the <https://firebase.google.com/docs/reference/rest/auth> page.

The response of both APIs are similar with the different that the login API send and extra field in the response (registered), so, we going to use the same interface for both response (AuthResponseData) and we going to mark the field *registed* with the symbol **?** watch’s means optional.

Also, how the error handler of both methods are similar we create a function that return the type of error expected the pipe *catchError*, and we call this function on pipe for both methods.

Auth.service.ts

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

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

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

export interface AuthResponseData {

  idToken: string;

  email: string;

  refreshToken: string;

  expiresIn: string;

  localId: string;

  registered?: boolean

}

const FIREBASE\_PROJECT\_KEY = 'AIzaSyDKNsEn6rrkJE7XUMh2R3e74OW8YfwUh7k';

@Injectable({ providedIn: 'root' })

export class AuthService {

  constructor(private http: HttpClient) { }

  signup(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key='+FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(catchError(this.handleError));

  }

  login(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key='+

FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(catchError(this.handleError));

  }

  private handleError(errorRes: HttpErrorResponse) {

    let errorMessage = 'An unknown error occurred!';

    if (!errorRes.error || !errorRes.error.error) {

      return throwError(errorMessage);

    }

    switch (errorRes.error.error.message) {

      case 'EMAIL\_EXISTS':

        errorMessage = 'The email address is already in used.'

        break;

      case 'EMAIL\_NOT\_FOUND':

        errorMessage = 'The email does not exist.'

        break;

      case 'INVALID\_PASSWORD':

        errorMessage = 'The password is invalid.'

        break;

    }

    return throwError(errorMessage);

  }

}

On the other hand in the auth.component.ts how the response and handler error of both services (login and signup) it is the same so we create the Observable authObs and return the response of both services to this one. It is a good way to avoid repeat the code.

export class AuthComponent implements OnInit {

  //this variable control if we are in login mode or signup mode

  isLoginMode = true;

  isLoading = false;

  error: string = null;

  constructor(private authService: AuthService) { }

  onSwitchMode() {

    this.isLoginMode = !this.isLoginMode;

  }

  onSubmit(form: NgForm) {

    if(!form.valid) {

      return;

    }

    const email = form.value.email;

    const password = form.value.password;

    this.isLoading = true;

    let authObs: Observable<AuthResponseData>;

    if(this.isLoginMode){

      authObs = this.authService.login(email, password);

    } else {

      authObs = this.authService.signup(email, password);

    }

    authObs.subscribe(

      resData => {

        console.log(resData);

        this.isLoading = false;

      },

      errorMessage => {

        console.log(errorMessage);

        this.error = errorMessage;

        this.isLoading = false;

      }

    );

    form.reset();

  }

}

## Creating & Storing the User Data

Here we first going to create a User model file where we going to store the information we need for the current user in the app. Also have a method to retrieve the token, this is important to check if token has expired or just to call another API of Firbase setting this token in the header.

user.model.ts

export class User {

  constructor(

    public email: string,

    public id: string,

    private \_token: string,

    private \_tokenExpirationDate: Date

  ) { }

  get token() {

    if(!this.\_tokenExpirationDate || new Date() > this.\_tokenExpirationDate) {

      return null;

    }

    return this.\_token;

  }

}

Now in our auth.service.ts we going to store our user in the model every time the signup or login, and we going to do this using a Subjet (observable).

I our calls to signup and login we going to use the *tap* operator of Redux, who let us do what we want with the response without change it. In this case we store the user.

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

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

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

import { throwError, Subject } from 'rxjs';

import { User } from './user.model';

/\*\*

 \* Response of firebase

 \* idToken:      A Firebase Auth ID token for the newly created user or authenticated user.

 \* email:        The email for the newly created user or authenticated user.

 \* refreshToken: A Firebase Auth refresh token for the newly created user or authenticated user.

 \* expiresIn:    The number of seconds in which the ID token expires.

 \* localId:      The uid of the newly created user or authenticated user.

 \* registered:   Whether the email is for an existing account.

 \*/

export interface AuthResponseData {

  idToken: string;

  email: string;

  refreshToken: string;

  expiresIn: string;

  localId: string;

  registered?: boolean

}

const FIREBASE\_PROJECT\_KEY = 'AIzaSyDKNsEn6rrkJE7XUMh2R3e74OW8YfwUh7k';

@Injectable({ providedIn: 'root' })

export class AuthService {

  user = new Subject<User>();

  constructor(private http: HttpClient) { }

  signup(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key='+FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(this.handleError),

      tap(resData => {

        this.handleAuthentication(resData.email, resData.localId, resData.idToken, +resData.expiresIn);

      })

    );

  }

  login(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=' + FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(this.handleError),

      tap(resData => {

        this.handleAuthentication(resData.email, resData.localId, resData.idToken,

+resData.expiresIn);

      })

    );

  }

  private handleAuthentication(email:string, userId:string, token: string, expiresIn: number) {

    const expirationDate = new Date(new Date().getTime() + expiresIn \* 1000);

    const user = new User(

      email,

      userId,

      token,

      expirationDate

    )

    this.user.next(user);

  }

}

Note: when we call our *handleAuthentication* function we expects in *expiresIn* parameter a number, with the + operator we convert the string response of Firebase to number.

Another important thing is that in *handleAuthentication* function is the *expirationDate*. It is calculated with (new Date().getTime()) what going to take the current date of the system and also gettime() get the time in milliseconds, then we plus the *expiresIn* firebase field that contain the a number of second that going to leave the token, how this value is in seconds, we multiply for 1000 to convert it to milliseconds.

## Reflecting the Auth State in the UI

Let's start by forwarding the user to a different route once the user is logged in. Now since we always want to redirect the user once we authenticate it, we of course can do this in two different places - we can either do it here in handle authentication or we do it in the component, here inside of subscribe in the success case only of course.

Now it's up to you when you do that, if you prefer redirecting in the component or from inside a service. Both is valid.

In this case we do it in the component to maintain clean the service. E.g.

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

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

import { AuthService, AuthResponseData } from './auth.service';

import { Observable } from 'rxjs';

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

@Component({

  selector: 'app-auth',

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

  styleUrls: ['./auth.component.css']

})

export class AuthComponent implements OnInit {

  //this variable control if we are in login mode or signup mode

  isLoginMode = true;

  isLoading = false;

  error: string = null;

  constructor(private authService: AuthService,

              private router: Router) { }

  onSwitchMode() {

    this.isLoginMode = !this.isLoginMode;

  }

  onSubmit(form: NgForm) {

    if(!form.valid) {

      return;

    }

    const email = form.value.email;

    const password = form.value.password;

    this.isLoading = true;

    let authObs: Observable<AuthResponseData>;

    if(this.isLoginMode){

      authObs = this.authService.login(email, password);

    } else {

      authObs = this.authService.signup(email, password);

    }

    authObs.subscribe(

      resData => {

        console.log(resData);

        this.isLoading = false;

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

      },

      errorMessage => {

        console.log(errorMessage);

        this.error = errorMessage;

        this.isLoading = false;

      }

    );

    form.reset();

  }

}

On the other hand we want only show some element in the header menu if we are logged and others if we not. For that in the header.component.ts we going to create a flag (isAuthenticated) to know if the user is authenticated. Also we need to subscribe to the user Subject in auth.service.ts, in that way every time the user change, we going to receive an event in out header file. E.g.

auth.service.ts

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

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

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

import { throwError, Subject } from 'rxjs';

import { User } from './user.model';

/\*\*

 \* Response of firebase

 \* idToken:      A Firebase Auth ID token for the newly created user or authenticated user.

 \* email:        The email for the newly created user or authenticated user.

 \* refreshToken: A Firebase Auth refresh token for the newly created user or authenticated user.

 \* expiresIn:    The number of seconds in which the ID token expires.

 \* localId:      The uid of the newly created user or authenticated user.

 \* registered:   Whether the email is for an existing account.

 \*/

export interface AuthResponseData {

  idToken: string;

  email: string;

  refreshToken: string;

  expiresIn: string;

  localId: string;

  registered?: boolean

}

const FIREBASE\_PROJECT\_KEY = 'AIzaSyDKNsEn6rrkJE7XUMh2R3e74OW8YfwUh7k';

@Injectable({ providedIn: 'root' })

export class AuthService {

  user = new Subject<User>();

  constructor(private http: HttpClient) { }

  signup(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=' + FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(this.handleError),

      tap(resData => {

        this.handleAuthentication(resData.email, resData.localId, resData.idToken, +resData.expiresIn);

      })

    );

  }

  login(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=' + FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(this.handleError),

      tap(resData => {

        this.handleAuthentication(resData.email, resData.localId, resData.idToken, +resData.expiresIn);

      })

    );

  }

  private handleAuthentication(email: string, userId: string, token: string, expiresIn: number) {

    const expirationDate = new Date(new Date().getTime() + expiresIn \* 1000);

    const user = new User(

      email,

      userId,

      token,

      expirationDate

    )

    this.user.next(user);

  }

  private handleError(errorRes: HttpErrorResponse) {

    let errorMessage = 'An unknown error occurred!';

    if (!errorRes.error || !errorRes.error.error) {

      return throwError(errorMessage);

    }

    switch (errorRes.error.error.message) {

      case 'EMAIL\_EXISTS':

        errorMessage = 'The email address is already in used.'

        break;

      case 'EMAIL\_NOT\_FOUND':

        errorMessage = 'The email does not exist.'

        break;

      case 'INVALID\_PASSWORD':

        errorMessage = 'The password is invalid.'

        break;

    }

    return throwError(errorMessage);

  }

}

Header.component.ts

@Component({

    selector: 'app-header',

    templateUrl: './header.component.html'

})

export class HeaderComponent implements OnInit, OnDestroy{

  isAuthenticated = false;

  private userSub: Subscription;

  constructor(

    private dataStorageService: DataStorageService,

    private authService: AuthService

  ) {}

  ngOnInit(): void {

    this.userSub = this.authService.user.subscribe( user => {

      // this.isAuthenticated = !user ? false : true; If user is null

      this.isAuthenticated = !!user; //this is the same of about line

    });

  }

header.components.html

    <div class="collapse navbar-collapse">

      <ul class="nav navbar-nav">

        <li routerLinkActive="active" \*ngIf="isAuthenticated">

          <a routerLink="/recipes">Recipes</a>

        </li>

        <li routerLinkActive="active" \*ngIf="!isAuthenticated">

          <a routerLink="/auth">Authenticate</a>

        </li>

        <li routerLinkActive="active">

          <a routerLink="/shopping-list">Shopping List</a>

        </li>

      </ul>

Adding the Token to Outgoing Requests

Now for every call to the APIs of firebase we have to attach the token retrieved in the login. The token that we are storing in the user object in the auth service.

Now we could set up a subscription to the user but I don't care about every change to the user, instead when we try to store or fetch recipes, I just want to get the token of the currently authenticated user.   
I don't want to have an ongoing subscription. Hence in the auth service, we now should also store the token or the user in a way that we can access it through different means, so not through a subject which is great for reactively updating the user interface but that we also have a way of on-demand fetching of the user data.   
For that we can actually also use a different type of subject here.There is a different type of subject RxJS offers which is called *behavior subject*. Now this is also imported from RxJS and generally, it behaves just like the other subject, which means we can call next, to emit a value and we can subscribe to it to be informed about new values.

The difference is that behavior subject also gives subscribers immediate access to the previously emitted value even if they haven't subscribed at the point of time that value was emitted. That means we can get access to be currently active user even if we only subscribe after that user has been emitted. So this means when we fetch data and we need that token at this point of time, even if the user logged in before that point of time which will have been the case, we get access to that latest user. E.g.

export class AuthService {

  user = new BehaviorSubject<User>(null);

…..

Now in the service where we call this BehabiorSubject we still I need to subscribe to get that user, so to make sure I only get the user once and thereafter I'm done, we could of course manually immediately call unsubscribe. But there is a special operator RxJS named ***take*** that can be used in pipe. We have to pass a parameter to take, the parameter is a number that means the number of times that the observable will be esecuted, after that he automatically unsubscribe. So this manages the subscription for me, gives me the latest user and unsubscribes.

The problem just is here, we're in subscribe of an observable and here, we create yet another observable. In the end, I want to return this observable here but returning from inside subscribe doesn't work, we have to return on the top level of this method. The solution is that we pipe these two observables, the user observable and the HTTP observable together into one big observable and now this is getting a bit more advanced here, related to RxJS but still, pretty easy to understand in the end. You simply add another operator here to pipe for the user observable and there, we can use ***exhaustMap***. exhaustMap is also imported from rxjs/operators and whilst the name might look strange, it does something very simple. It waits for the first observable, for the user observable to complete which will happen after we took the latest user. Thereafter, it gives us that user, so in exhaustMap we pass in a function, there we get the data from that previous observable and now we return a new observable in there which will then replace our previous observable in that entire observable chain.

So we start of with a user observable and once we're done with that, this will be replaced in that observable chain with the inner observable we return inside of that function we pass to exhaustMap, in this case the http observable.

Finally we have data-storage.service.ts:

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

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

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

import { RecipeService } from '../recipes/recipe.service';

import { Recipe } from '../recipes/recipe.model';

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

@Injectable({ providedIn: 'root' })

export class DataStorageService {

  constructor(private http: HttpClient, private recipeService: RecipeService,

    private authService: AuthService) { }

  fetchRecipes() {

    return this.authService.user.pipe(

      take(1),

      exhaustMap( user => {

        return this.http.get<Recipe[]>(

          'https://backend-recipeapp.firebaseio.com/recipes.json',

          {

            params: new HttpParams().set('auth', user.token)

          }

        );

      }),

      map(recipes => {

        //if the recipe have no ingredients stored, we create and empty array

        return recipes.map( recipe => {

          return {...recipe, ingredients: recipe.ingredients ? recipe.ingredients : []}

        });

      }),

      tap(recipes => {

        this.recipeService.setRecipes(recipes);

      })

    );

  }

}

Note: the token to firebase is passed as a parameter in the URL and not in the header how in almost all the APIs, but that is the way firebase function.

## Attaching the Token with an Interceptor

A better way or another way to do this is using an interceptor for attach the token to the requests of storeRecipes and fetchRecipes. E.g.

auth-interceptor.service.ts

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

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

import { take, exhaustMap } from 'rxjs/operators';

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

@Injectable()

export class AuthInterceptorService implements HttpInterceptor {

  constructor(private authService: AuthService) { }

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

    return this.authService.user.pipe(

      take(1),

      exhaustMap(user => {

        if (!user) {//if we are calling signup or logging api don't attach the user param

          return next.handle(req);

        }

        const modifiedReq = req.clone({

          params: new HttpParams().set('auth', user.token)

        });

        return next.handle(modifiedReq);

      })

    );

  }

}

data-storage.service.ts

@Injectable({ providedIn: 'root' })

export class DataStorageService {

  constructor(private http: HttpClient, private recipeService: RecipeService,

    private authService: AuthService) { }

  storeRecipes() {

    const recipes = this.recipeService.getRecipes();

    this.http

      .put('https://backend-recipeapp.firebaseio.com/recipes.json', recipes)

      .subscribe(response => {

        console.log(response);

      });

  }

  fetchRecipes() {

    return this.http.get<Recipe[]>(

      'https://backend-recipeapp.firebaseio.com/recipes.json'

    )

    .pipe(

      map(recipes => {

        //if the recipe have no ingredients stored, we create and empty array

        return recipes.map(recipe => {

          return { ...recipe, ingredients: recipe.ingredients ? recipe.ingredients : [] }

        });

      }),

      tap(recipes => {

        this.recipeService.setRecipes(recipes);

      })

    );

  }

}

## Adding Logout

Here, we simply call the user handlSubject throughout the next method and pass null instead of the User object. Also in the header component where we have the logout menu we going to call the logout method created in this service. E.g.

export class AuthService {

  user = new BehaviorSubject<User>(null);

  constructor(private http: HttpClient, private router: Router) { }

  logout() {

    this.user.next(null);

    this.router.navigate(['/auth']);

  }

## Adding Auto-Login

First of all though, I want to make sure that once we are authenticated, we keep that status as long as we're navigating around in the app and also keep that token which we have once we reload the page because at the moment when we reload the page we lose all that status because when you reload the page, your Angular application restarts, the old one is basically dumped and since right now, we're managing everything in memory. Remember that we're storing the token in a user model which happens in Javascript only and which therefore happens in memory, since we manage that all in memory, we'll lose all that state whenever the application restarts because that memory gets cleared automatically, there is no connection between the last time we ran this Angular app on this web page and the current time.

So there is no connection and the memory is lost, so if we want to store that token, we have to store it somewhere else and not in a Javascript variable or class property, we have to store it in a persistent storage that survives page reloads and browser restarts, that we can write to and that we can fetch data from.

Now what would be such a storage? In a browser, you've got different storages, you can't access the normal file system on your device but you can work with cookies or with **local storage**, which is an API exposed by the browser to store simple key-value pairs basically on the filesystem but controlled by the browser and we will use local storage here to store that token.

For this we going to:

1. In handleAuthentication method called in loging method we have to store in localStorage the user object we need. How in localStorage we store pair of key value of type string we convert the object to string with JSON.stringify(user)
2. Create an autoLogin() method in our auth service. This method retrieve form the local storage the user object and emit an event with the object retrieved.
3. Finally in the main app.component.ts in the onInit method we call autoLogin() method, so every time the page is loaded and the main component restart, retrieve the object.

auth.service.ts

@Injectable({ providedIn: 'root' })

export class AuthService {

  user = new BehaviorSubject<User>(null);

  constructor(private http: HttpClient, private router: Router) { }

  login(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=' + FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(this.handleError),

      tap(resData => {

        this.handleAuthentication(resData.email, resData.localId, resData.idToken, +resData.expiresIn);

      })

    );

  }

  autoLogin() {

    const userData: {

      email: string,

      id: string,

      \_token: string,

      \_tokenExpirationDate: string

    } = JSON.parse(localStorage.getItem('userData'));

    if(!userData) {

      return;

    } else {

      const loadedUser = new User(userData.email, userData.id, userData.\_token,

        new Date(userData.\_tokenExpirationDate));

      if(loadedUser.token){

        this.user.next(loadedUser);

      }

    }

  }

  logout() {

    this.user.next(null);

    this.router.navigate(['/auth']);

  }

  private handleAuthentication(email: string, userId: string, token: string, expiresIn: number) {

    const expirationDate = new Date(new Date().getTime() + expiresIn \* 1000);

    const user = new User(email, userId, token, expirationDate);

    this.user.next(user);

    localStorage.setItem('userData', JSON.stringify(user)); //storing in browser user information

  }

}

app.component.ts

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

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit{

  constructor(private authService: AuthService) {}

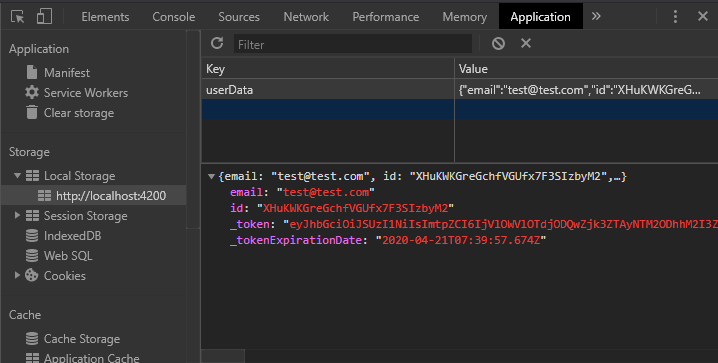
  ngOnInit(): void {

    this.authService.autoLogin();

  }

}

In Crome browser on in any other, we can see the local storage:



## Adding Auto-Logout

For this we going to create a function named autoLogout(expirationDuration: number) who going to receive the time for the expiration send by firebase, then we create a timer with this token, and when finish this time we call the logout function. In Logout function we going to clean the local storage and also clean the timer.

We should call the logout function any time we login in the application or auto login and pass the correct expirationDuration parameter in milliseconds. E.g.

auth.service.ts

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

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

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

import { throwError, Subject, BehaviorSubject } from 'rxjs';

import { User } from './user.model';

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

/\*\*

 \* Response of firebase

 \* idToken:      A Firebase Auth ID token for the newly created user or authenticated user.

 \* email:        The email for the newly created user or authenticated user.

 \* refreshToken: A Firebase Auth refresh token for the newly created user or authenticated user.

 \* expiresIn:    The number of seconds in which the ID token expires.

 \* localId:      The uid of the newly created user or authenticated user.

 \* registered:   Whether the email is for an existing account.

 \*/

export interface AuthResponseData {

  idToken: string;

  email: string;

  refreshToken: string;

  expiresIn: string;

  localId: string;

  registered?: boolean

}

const FIREBASE\_PROJECT\_KEY = 'AIzaSyDKNsEn6rrkJE7XUMh2R3e74OW8YfwUh7k';

@Injectable({ providedIn: 'root' })

export class AuthService {

  user = new BehaviorSubject<User>(null);

  private tokenExpirationTimer: any;

  constructor(private http: HttpClient, private router: Router) { }

  login(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=' + FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(this.handleError),

      tap(resData => {

        this.handleAuthentication(resData.email, resData.localId, resData.idToken, +resData.expiresIn);

      })

    );

  }

  autoLogin() {

    const userData: {

      email: string,

      id: string,

      \_token: string,

      \_tokenExpirationDate: string

    } = JSON.parse(localStorage.getItem('userData'));

    if(!userData) {

      return;

    } else {

      const loadedUser = new User(userData.email, userData.id, userData.\_token,

        new Date(userData.\_tokenExpirationDate));

      if(loadedUser.token){

        this.user.next(loadedUser);

        const expirationDuration =

          new Date(userData.\_tokenExpirationDate).getTime() -  
   new Date().getTime();

        this.autoLogout(expirationDuration);

      }

    }

  }

  logout() {

    this.user.next(null);

    this.router.navigate(['/auth']);

    localStorage.removeItem('userData');

    if(this.tokenExpirationTimer) {

      clearTimeout(this.tokenExpirationTimer);

      this.tokenExpirationTimer = null;

    }

  }

  /\*\*

   \* Automatically logout from the application

   \* @param expirationDuration This time is in milliseconds

   \*/

  autoLogout(expirationDuration: number) {

    this.tokenExpirationTimer = setTimeout(() => {

      this.logout();

    }, expirationDuration);

  }

  private handleAuthentication(email: string, userId: string, token: string, expiresIn: number) {

    const expirationDate = new Date(new Date().getTime() + expiresIn \* 1000);

    const user = new User(email, userId, token, expirationDate);

    this.user.next(user);

    this.autoLogout(expiresIn \* 1000);

    localStorage.setItem('userData', JSON.stringify(user)); //store in browser the user info

  }

}

## Adding an Auth Guard

Right now if we set in browser the url for recipes we can access to that page even if we are not login, so this is an issue. We can solve this putting a Guard on the recipes router. A Guard as we study in Router section it is an event that will be executed just before a specific route for a page is visualized.

Now this Guard for latest Angular version let us retrieve a true or false, to show or not show the page, a Primise, Observable but also a UrlTree, that it is an URL to navigate.

So in this example if user has been login, we return true and show the recipes page, on the other hand we send the use to the authentication page. E.g.

auth.guard.ts

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

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

import { Observable } from 'rxjs';

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

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

@Injectable({providedIn: 'root'})

export class AuthGuard implements CanActivate {

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

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):

    boolean | UrlTree | Promise<boolean | UrlTree> | Observable<boolean | UrlTree>{

    return this.authService.user.pipe(

      take(1),

      map( user => {

      //return true if user exists, or if it is null send you to auth page

      const isAuth = !!user;

      if(isAuth) {

        return true;

      } else {

        return this.router.createUrlTree(['/auth']);

      }

    }));

  }

}

app-routing.module.ts

import { AuthGuard } from './auth/auth.guard';

const AppRoute: Routes = [

    { path: '', redirectTo: '/recipes', pathMatch: 'full' },

    { path: 'recipes', component: RecipesComponent, canActivate: [AuthGuard],

        children: [

            { path: '', component: RecipeStartComponent },

            { path: 'new', component: RecipeEditComponent },

            { path: ':id', component: RecipeDetailComponent, resolve: [RecipesResolverService] },

            { path: ':id/edit', component: RecipeEditComponent, resolve: [RecipesResolverService] },

        ]

    },

    { path: 'shopping-list', component: ShoppingListComponent },

    { path: 'auth', component: AuthComponent },

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

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

]

## Useful Resources & Links

Useful Docs:

* Firebase Auth REST API Docs: <https://firebase.google.com/docs/reference/rest/auth>
* More on JWT: [https://jwt.io](https://jwt.io/)

# Section 21: Dynamic Components

In this module, we'll have a look at components again but now, add dynamic components.

Now what are dynamic components? Dynamic components are essentially components which you create dynamically at runtime.

So let's say you want to show an alert, you want to show a modal, some overlay which should only be loaded upon a certain action, for example you have an error and you want to not show that error box which we're currently showing in our course project but we want to show an overlay on the entire screen or something like that, that could be done or could be implemented by using dynamic components. And dynamic components is not a specific term, not a specific feature provided by Angular but we can simply add components that we load through our code and that's exactly what we'll do in this module. You will learn how to create such a component and how to then load it on demand, how to communicate with it and also how to get rid of it.

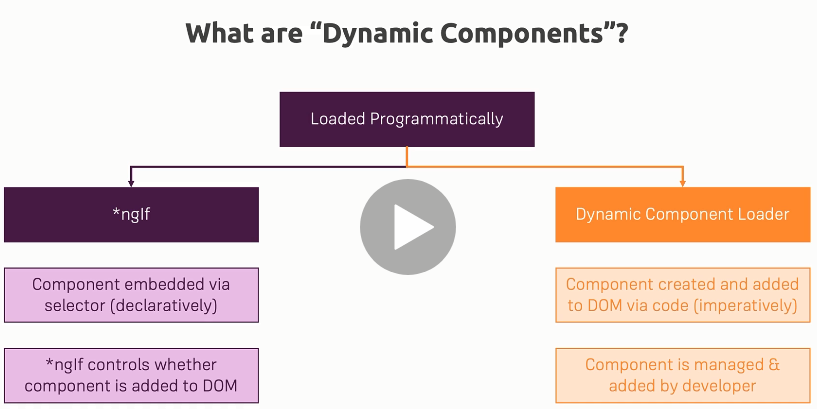
## Adding an Alert Modal Component

For that, I'm back in the course project and the part I want to handle with dynamic components is here on the login when I enter incorrect credentials, at the moment we then have that spinner and then we see that error message there.

Now nothing wrong with that way of displaying an error, it's perfectly fine, you don't have to change that but to demonstrate how you could load a component dynamically, I will change it.

I will not use that box here but instead, we'll build a very simple alert box which is presented as an overlay here on the screen with a nice backdrop that hides the background and in that alert box, we can then simply click OK to get rid of it and have another try here. So that's the idea.

## Understanding the Different Approaches



It is important to understand that dynamic components in general just means that you want to display some component dynamically, which means it's not always there but it's there once something specific happens in your code.

Therefore, you typically want to control programmatically when that component should be loaded, when it should be displayed and you already learned about a way of doing that and we're currently using that way and that way is ngIf. ngIf is awesome because it allows you to use a declarative approach, which means you simply add that component selector in your template and then you use ngIf to still only load it upon a certain condition, so you use ngIf to control whether the DOM really has that component or not and therefore ngIf definitely is a great approach of loading a component dynamically.

It saves you a lot of extra work which you will have to do with the alternative approach you're about to learn and it's definitely worth a thought and has no major downsides and at the moment as I mentioned, we are using ngIf here in this project, we're using it in many places and we're also using it for this alert box and I'll show you how to finish the entire alert flow before we then have a look at that alternative.

Nonetheless, what would be the alternative?

The alternative is that you use something which in the past was named dynamic component loader. Now this was a helper utility that doesn't exist anymore or that you shouldn't use anymore but in the end, it's now about a general concept you could say of creating a component in code and then manually attaching it to the DOM.

So there, you have to control, in your code by yourself, how that component is instantiated, that data is passed into it and also that it is removed.

So everything ngIf does for you, you have to do on your own there. Still, this approach can be useful, it also of course allows you to control it entirely from code and you don't have to touch the template.

Sometimes you might not want to do that or you have no real way of adding the component to the template and therefore, this is also an interesting approach.

Now we'll have a look at both here, we started with ngIf, let's finish it up with that approach before we then dive into creating components and managing components from code.

## Using ngIf

So we started with ngIf and therefore at the moment, our alert component here is simply included with its selector and controlled with ngIf.

The huge advantage of this is that we don't have to manually create it and we also can pass data in as we learn it here, with property binding. It also means that listening to events is super simple and straightforward.

If we want to make sure we can also get rid of that alert, all we need to do is we need to emit an event and we make that event listenable from outside. E.g.

alert.component.html

<div class="backdrop" (click)="onClose()"></div>

<div class="alert-box">

  <p>{{ message }}</p>

  <div class="alert-box-action">

    <button class="btn btn-primary" (click)="onClose()">Close</button>

  </div>

</div>

alert.component.ts

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

@Component({

  selector: 'app-alert',

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

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

})

export class AlertComponent {

  @Input() message: string;

  @Output() close = new EventEmitter<void>();

  onClose() {

    this.close.emit();

  }

}

auth.component.html (where is used)

<div class="row">

  <div class="col-xs-12 col-md-6 col-md-offset-3">

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

      <p>{{ error }}</p>

    </div> -->

    <app-alert [message]="error" \*ngIf="error" (close)="onHandleError()"></app-alert>

    <div \*ngIf="isLoading" style="text-align: center;">

      <app-loading-spinner></app-loading-spinner>

    </div>

    <form #authForm="ngForm"

…….

auth.component.ts (where is used)

  onHandleError() {

    this.error = null;

  }

So now this is the finished flow with ngIf and I strongly recommend using that flow. There are rarely situations where you absolutely need to use the othe approach, using this approach with ngIf is so much easier and therefore really is an approach you should use when you have a scenario where you need to show some component dynamically when some condition is met.

Nonetheless, let's now see how we could create that alert component entirely in code and control it from there.

## Creating Component Programmatic

So now the alert will not be shown here by using ngIf. Now my goal is to instead create that alert component dynamically and show it dynamically and dynamically is the wrong word maybe, I mean programmatically, from inside code.

For this we need to use the class **ComponentFactoryResolver** of Angular, who let us create instance of components. We don’t create a new object of ComponentFactoryResolver, we statically call resolveComponentFactory method and pass the component to be created, tha the way it Works. This will return a component factory or to be precise, an alert component factory, not the component itself, just the factory.

So this now is essentially an object that knows how to create alert components, which is a great first step at least. Now with that factory, we can use that factory to create a concrete component but for this, we also need a place where we can attach it in our DOM and we don't have that place yet.

Angular needs a view container ref which you might remember from the directives deep dive section. A view container ref is essentially an object managed internally by Angular, which gives Angular a reference, a pointer to a place in the DOM you could say, with which it can interact and this object has more than just like the coordinates where it sits, it has methods like "hey, please create a component here" and we need something like this. Now to get access to such a view container reference, there is a nice trick or actually not a hack, it's the official approach for getting access to it, we can create a helper directive.   
Now that directive could be named placeholder. And in is contructor we going to create a public property of type **ViewContainerRef**.

Now why do we need that?

Because now, we will be able to add the directive to some place in our DOM, in our templates and then get access to it with **@ViewChild** and then get access to the public view container ref to work with that view container ref of that directive.

With the placeholder directive added, let's go back to the auth component and now we need to add a new element here. Now you could add a div but there is a better element, you could should use **ng-template** here. This is a directive Angular ships with, which will not actually render anything to the DOM but still is accessible in the Angular templating language.

In the auth.component.ts file, we can now get access to that directive with the help of @ViewChild which you already learned about in the component deep dive section.

Now with the @ViewChild created of type PlaceholderDirective we can access to the viewContainerRef. With viewContainerRef we have methods like for example *createComponent* that finally let us create the component passing the factory object.

placeholder.directive.ts

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

@Directive({

  selector: '[appPlaceholder]'

})

export class PlaceholderDirective {

  constructor(public viewContainerRef: ViewContainerRef) {}

}

auth.component.html

<ng-template appPlaceholder></ng-template>

<div class="row">

  <div class="col-xs-12 col-md-6 col-md-offset-3">

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

      <p>{{ error }}</p>

    </div> -->

    <!-- <app-alert [message]="error" \*ngIf="error" (close)="onHandleError()"></app-alert>-->

    <div \*ngIf="isLoading" style="text-align: center;">

      <app-loading-spinner></app-loading-spinner>

    </div>

    <form #authForm="ngForm" (ngSubmit)="onSubmit(authForm)" \*ngIf="!isLoading">

      <div class="form-group">

        <label for="email">E-mail</label>

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

          ngModel name="email" required email>

      </div>

      <div class="form-group">

        <label for="password">Password</label>

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

          ngModel name="password" required minlength="6">

      </div>

      <div>

        <button class="btn btn-primary" type="submit" [disabled]="!authForm.valid">

          {{ isLoginMode ? 'Login' : 'Sign Up'}}</button> |

        <button class="btn btn-primary" type="button" (click)="onSwitchMode()">

          Switch to {{ isLoginMode ? 'Sign Up' : 'Login'}}</button>

      </div>

    </form>

  </div>

</div>

auth.component.ts

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

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

import { AuthService, AuthResponseData } from './auth.service';

import { Observable, Subscription } from 'rxjs';

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

import { AlertComponent } from '../shared/alert/alert.component';

import { PlaceholderDirective } from '../shared/placeholder/placeholder.directive';

@Component({

  selector: 'app-auth',

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

  styleUrls: ['./auth.component.css']

})

export class AuthComponent implements OnDestroy{

  //this variable control if we are in login mode or signup mode

  isLoginMode = true;

  isLoading = false;

  error: string = null;

  @ViewChild(PlaceholderDirective) alertHost: PlaceholderDirective;

  private closeSub: Subscription;

  constructor(private authService: AuthService,

              private router: Router,

              private componentFactoryResolver: ComponentFactoryResolver) { }

  onSwitchMode() {

    this.isLoginMode = !this.isLoginMode;

  }

  onSubmit(form: NgForm) {

    if(!form.valid) {

      return;

    }

    const email = form.value.email;

    const password = form.value.password;

    this.isLoading = true;

    let authObs: Observable<AuthResponseData>;

    if(this.isLoginMode){

      authObs = this.authService.login(email, password);

    } else {

      authObs = this.authService.signup(email, password);

    }

    authObs.subscribe(

      resData => {

        console.log(resData);

        this.isLoading = false;

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

      },

      errorMessage => {

        console.log(errorMessage);

        this.error = errorMessage;

        this.showErrorAlert(errorMessage);

        this.isLoading = false;

      }

    );

    form.reset();

  }

  onHandleError() {

    this.error = null;

  }

  ngOnDestroy() {

    if(this.closeSub) {

      this.closeSub.unsubscribe();

    }

  }

  private showErrorAlert(message: string) {

    const alertCmpFactory =

    this.componentFactoryResolver.resolveComponentFactory(AlertComponent);

    const hostViewContainerRef = this.alertHost.viewContainerRef;

    hostViewContainerRef.clear();

    //creating the component

    const cmpRef = hostViewContainerRef.createComponent(alertCmpFactory);

    cmpRef.instance.message = message;

    this.closeSub = cmpRef.instance.close.subscribe( () => {

      this.closeSub.unsubscribe();

      hostViewContainerRef.clear();

    })

  }

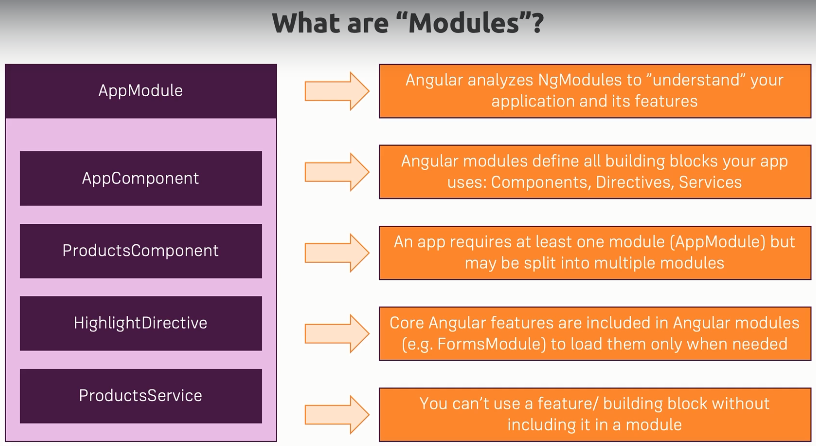
}

**Useful Resources:**

* Official Docs: <https://angular.io/guide/dynamic-component-loader>

# Section 22: Angular Modules & Optimizing Angular Apps

**What are Modules?**



And with that, I mean the Angular modules. You will also hear the term module where you are talking about the imports you have in your files when you import something from another file into another file, that also is sometimes referred to as modules but these would be Javascript modules, they're totally not related to Angular and they're not what I mean.

I mean that NgModule thing that we have in our app, like the app module or the app routing module. So modules in the end are ways of bundling Angular building blocks together, that would be components, directives, services, pipes.

You have to group that all together into such Angular modules so that Angular is aware of these features because Angular doesn't automatically scan all the files in your project, it doesn't search all the code you write and automatically detect all components and services, instead you need to tell Angular what is a component, which components do you have and you can then bundle them together into modules. Every Angular app needs to have at least one module, that app module, you can't have an Angular app without it.

So Angular analyzes these NgModules to understand your application and its features, you use them to define all the core building blocks - components, directives, services, pipes and so on and app as I just mentioned requires at least to one such module, the app module but you may split it into multiple modules and we will do that in this core section.

You also have core Angular features that are wrapped into modules, things like the forms module which we added in the course already, so Angular itself also provides certain features grouped together into such modules, so that you don't have to declare like 10 different forms directives, instead that forms module gives you all these directives, that is how you can think about that. And important also, you can't use a certain feature or building block without including it in a module. How you include it if you add it to declarations or providers depends on the feature you are talking about.

## Analyzing the AppModule

This is our project and there, we have the app module. Actually this is not the only module and again, with module, I mean things with that NgModule decorator, we got a one other module in this app and that would be the app routing module. Here, we also have an NgModule.

So we already do use two modules and we use these modules to set certain things up and group certain things together.

Now let's start with the app module, there we have a long list of declarations for example.

As I mentioned, when we added things to that list, declarations is in the end an array of all the components, directives and custom pipes you're using in your application.

So all these things have to go into **declarations**, otherwise you can't use them in your templates or in your routes.

We also have that **imports array**. The imports array here is important because that allows you to import other modules into this module, so the imports array here is important to split your app into multiple modules.

Thus far we created only one extra module, the app routing module but we're using a couple of modules that ship with Angular, like the forms module, the reactive forms module and so on.

Now if we were to look into such a module, like the forms module, we would find a module with @NgModule as well and in there, we would have a declarations array with all the form related directives Angular supports, so things like automatically detecting a form element to create a form with the template driven approach or the required or the email directives.

So all of that is basically included in the forms module and so that you don't have to add all these features individually to your declarations array, we can simply import the forms module which speeds this up.

The **providers array** here then is another important key in our NgModule, here we define all the services we want to provide, however I'll come back to services and modules because that's kind of special.

It's not the case that you can only use a service in the module where you provide it or at least that's not always the case but I will come back to that.

For now it's important to know any service you plan on injecting needs to be provided and typically, you do this in the app module or you do that by adding that provided in root key here into that object you pass to @injectable, this is an alternative to adding that service to your app module providers array and this is actually the way I would use because it's faster but you could also add a service here to providers and it would also be available application-wide.

Now the **bootstrap array** is important for starting your app. It defines which component is available right in that index.html file and typically, you only have one component there.

You could have multiple components which is why bootstrap is also an array, you could define multiple components but each component would then kind of be detached from the other components, you would be creating different root component trees in your application and that is rarely what you want because it makes working between these components way harder.

So therefore, you typically have one root component and you add that to the bootstrap array.

Well and then we have the entry components here, that simply is important for components you create in code, as you learned in the section on dynamic components, this allows Angular to be aware of this component when it needs to create it programmatically.

Now as I mentioned, we do have one other module right now and that's the app routing module which we also added to imports.

The app routing module is simply there to hold our route configuration.

We could absolutely add this into our app module, we only outsourced it because that's quite a lot of code and it keeps our app module a bit leaner and that is already one reason for why you might want to split your application into multiple modules.

It can make your existing modules leaner and more focused and easier to edit and to maintain.

Now the app routing module simply imports the router module Angular offers us and uses a special method offered by that module, which then takes our route configuration to configure the Angular router with our route config and then something important happens, we export that router module here from the app routing module.

We need to do that because every module and now that is something you have to memorize, every module works on its own in Angular, they don't communicate with each other.

If you declare a component in a certain component, in a certain module like app module, if we declare recipes component here, then we can only use it in here, nowhere else, we can only use it in here. So we can use the selector in our other components we declared here but if we would have another module with other components, then we could not use selectors of components defined here in components of that other module and we'll see this in action very soon.

And therefore here, we have to export the router module with our route configuration, so that it really is not just available in this module which would not be helpful at all but that it is available in the app module where we now import app routing module and when we import another module, we import everything that module exports, which in this case is our configured router module and this is how the configured router module ends up in the app module and is available there.

So that is generally what we're already doing and what we're already using in our application.

We're using these modules and everything works. Now for smaller applications, you can absolutely leave it that way, nothing wrong with that, this app module also is not too big, it's certainly manageable but the bigger your application grows, you might want to split your code to have leaner modules that are easier to manage, easier to understand and also as you will learn later, to enhance performance, not automatically by splitting components but splitting components will be an important prerequisite to then also enhance performance.

App.module.ts

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    RecipesComponent,

    RecipeListComponent,

    RecipeDetailComponent,

    RecipeItemComponent,

    ShoppingListComponent,

    ShoppingEditComponent,

    DropdownDirective,

    ErrorPageComponent,

    RecipeStartComponent,

    RecipeEditComponent,

    AuthComponent,

    LoadingSpinnerComponent,

    AlertComponent,

    PlaceholderDirective

  ],

  imports: [

    BrowserModule,

    FormsModule,

    ReactiveFormsModule,

    HttpClientModule,

    AppRoutingModule

  ],

  providers: [

    ShoppingListService,

    RecipeService,

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

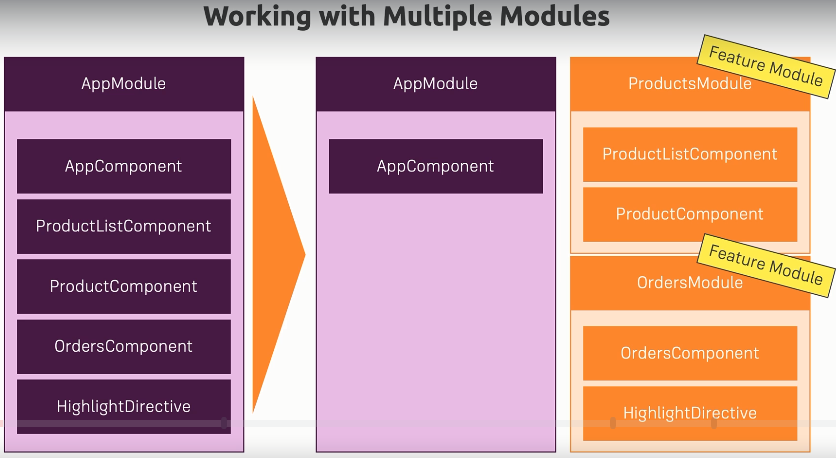
  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

## Getting Started with Feature Modules



So we can work with multiple modules. Instead of having one big app module, we could split that into an app module that defines our app component and then we have, in this slide here, two separate new modules, so-called feature modules.

Now the term feature module does not mean that you create them in a special way, it's still a class with the @ngModule decorator, it just means that this is a module that groups together components, directives, pipes, that are used in a certain feature area of your application, like in this slide when we're building let's say an online shop, we could have components that are primarily involved in rendering products and individual products and we could outsource those into a products module and if we then have an area in our application where we're primarily concerned with orders, then we could also outsource that into our orders module.

Now that is an important step of Angular application development, splitting your app into such feature modules because that is the prerequisite to performance improvements we'll dive into later and that also now ensures that when other developers work on your code or you work on your code, maybe after a short pause, you quickly know where to look for the set up for this specific part of your app because of course you could be building an application with hundreds or thousands of components and directives and then grouping them together into logical modules makes working with them and refactoring your code simply much easier.

Now let's apply that to our course project where we also have multiple features. Which features do we have in our course project there?

Well if we have a look at it in our app folder, I'd say we have three main feature areas - we have the recipes feature area, so all the components related to recipes, we have the shopping list feature area with all the logic related to the shopping list and we have the auth feature area, where we have all the components and all the logic related to authentication even if in this case it's only one component but still, it's a separate feature. The header and the shared components here, I'd say these are not really features of the app,the header is part of the app component, we do render it there, so it definitely belongs to the root of our application and regarding the shared components, in this app we might only use certain features in certain parts of the app, let's say the alert component, we only use it in the auth component but it would make sense that we share that across different features of our application because you could be showing an alert in the recipes area as well when you fail to load recipes. So the alert and so on, these shared things, we'll take care about later, they're not really a feature module of this app.

So I create file called recipes.modules.ts and I have put in the decalrations array all the alement that we have in the app.module.ts relatead to the recipes. We also add this class to the exports array to be abailable fron outside. Secondly I add the RecipesModule to the imports array of the AppModule, so now my RecipeModule will be used by AppModule.

Also in the RecipeModule we have to put in imports array: CommonModule, routerModule and ReactiveFormsModule because we use them in our components of the module.

What it is CommonModule ? So, to be able to use ngFor and ngIf in the recipes module, we also need the browser module. However, this is a special case and the only special case. The browser module must only be used once and that is in the app module because it does more than just add ngIf and ngFor, it does some general application start up work that only has to run once. Instead of the browser module, use the common module in all other places, in all other modules where you want to get access to ngIf and ngFor.

Important, whatever you declare here and whatever you use in the templates of these components, has to be imported in that module, it's not enough if you import it in the app module. The only exception to the rule are services, these only need to be set up once in the app module and you can access them in your whole application, even in components which you added to feature modules but anything that's used in a template, so components, directives, pipes, these things need to be declared or imported into the module where you plan on using them.

It's not enough to use them in another module, even if you export your things to that other module, even then it's not enough, Angular treats and parses every NgModule standalone.

recipes.module.ts

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

import { RecipesComponent } from './recipes.component';

import { RecipeListComponent } from './recipe-list/recipe-list.component';

import { RecipeDetailComponent } from './recipe-detail/recipe-detail.component';

import { RecipeItemComponent } from './recipe-list/recipe-item/recipe-item.component';

import { RecipeStartComponent } from './recipe-start/recipe-start.component';

import { RecipeEditComponent } from './recipe-edit/recipe-edit.component';

import { RouterModule } from '@angular/router';

import { CommonModule } from '@angular/common';

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

@NgModule({

  declarations: [

    RecipesComponent,

    RecipeListComponent,

    RecipeDetailComponent,

    RecipeItemComponent,

    RecipeStartComponent,

    RecipeEditComponent

  ],

  imports: [

    CommonModule,

    RouterModule,

    ReactiveFormsModule

  ],

  exports: [

    RecipesComponent,

    RecipeListComponent,

    RecipeDetailComponent,

    RecipeItemComponent,

    RecipeStartComponent,

    RecipeEditComponent

  ]

})

export class RecipesModule {}

app.module.ts

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

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

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

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

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

import { HeaderComponent } from './header/header.component';

import { ShoppingListComponent } from './shopping-list/shopping-list.component';

import { ShoppingEditComponent } from './shopping-list/shopping-edit/shopping-edit.component';

import { DropdownDirective } from './shared/dropdown.directive';

import { ShoppingListService } from './shopping-list/shopping-list.service';

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

import { ErrorPageComponent } from './shared/error-page/error-page.component';

import { RecipeService } from './recipes/recipe.service';

import { AuthComponent } from './auth/auth.component';

import { LoadingSpinnerComponent } from './shared/loading-spinner/loading-spinner.component';

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

import { AlertComponent } from './shared/alert/alert.component';

import { PlaceholderDirective } from './shared/placeholder/placeholder.directive';

import { RecipesModule } from './recipes/recipes.module';

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    ShoppingListComponent,

    ShoppingEditComponent,

    DropdownDirective,

    ErrorPageComponent,

    AuthComponent,

    LoadingSpinnerComponent,

    AlertComponent,

    PlaceholderDirective

  ],

  imports: [

    BrowserModule,

    FormsModule,

    ReactiveFormsModule,

    HttpClientModule,

    AppRoutingModule,

    RecipesModule

  ],

  providers: [

    ShoppingListService,

    RecipeService,

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

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

## Adding Routes to Feature Modules

We can also move the recipes related route configuration away from the app routing module, so I'm talking about the recipe menu, away from the app routing module and into our recipes module.

Note: Despite I don’t like this solution of split the router module in two modules, I’m going to doid because thei do in the course. But I prefer have all the router logic in one file.

Steps:

1. recipes-routing.module.ts file is created in the recipe folder and will be used by recipe.module.ts. Here we cut the recupes menu from app-routing.module.ts. Important, in the importor we use: RouterModule.forChild(routes)
2. We coment in app-routing.module.ts the recipe menu and mantein the import: RouterModule.forRoot(AppRoute)
3. The routers modules are used in its respective paren modules.

recipes-routing.module.ts

const routes: Routes = [

  {

    path: 'recipes', component: RecipesComponent, canActivate: [AuthGuard],

    children: [

      { path: '', component: RecipeStartComponent },

      { path: 'new', component: RecipeEditComponent },

      { path: ':id', component: RecipeDetailComponent, resolve: [RecipesResolverService] },

      { path: ':id/edit', component: RecipeEditComponent, resolve: [RecipesResolverService]},

    ]

  }

]

@NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class RecipesRoutingModule {

}

recipe.module.ts

@NgModule({

  declarations: [

    RecipesComponent,

    RecipeListComponent,

    RecipeDetailComponent,

    RecipeItemComponent,

    RecipeStartComponent,

    RecipeEditComponent

  ],

  imports: [

    CommonModule,

    RouterModule,

    ReactiveFormsModule,

    RecipesRoutingModule

  ],

app-routing.module.ts

const AppRoute: Routes = [

    { path: '', redirectTo: '/recipes', pathMatch: 'full' },

    // { path: 'recipes', component: RecipesComponent, canActivate: [AuthGuard],

    //     children: [

    //         { path: '', component: RecipeStartComponent },

    //         { path: 'new', component: RecipeEditComponent },

    //         { path: ':id', component: RecipeDetailComponent, resolve: [RecipesResolverService] },

    //         { path: ':id/edit', component: RecipeEditComponent, resolve: [RecipesResolverService] },

    //     ]

    // },

    { path: 'shopping-list', component: ShoppingListComponent },

    { path: 'auth', component: AuthComponent },

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

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

]

@NgModule({

    imports: [

        RouterModule.forRoot(AppRoute)

    ],

    exports: [

        RouterModule

    ]

})

export class AppRoutingModule {

}

app.module.ts

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    ShoppingListComponent,

    ShoppingEditComponent,

    DropdownDirective,

    ErrorPageComponent,

    AuthComponent,

    LoadingSpinnerComponent,

    AlertComponent,

    PlaceholderDirective

  ],

  imports: [

    BrowserModule,

    FormsModule,

    ReactiveFormsModule,

    HttpClientModule,

    AppRoutingModule,

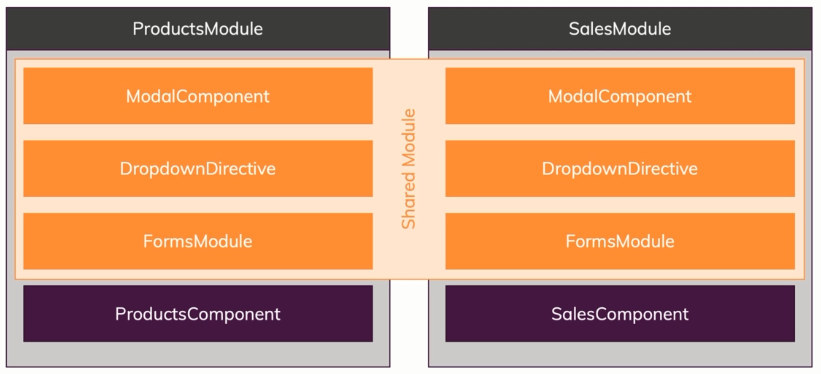
    RecipesModule

  ],

## Understanding Shared Modules

Now what's a shared module?

In this hypothetical app here, we got two feature modules and of course, we would also have an app module but it's not important for the shared module here.



So we got two feature modules - products and sales and as you can see, these two modules only differ in one component.

The products module has the products component, here on the left and the sales module on the bottom right has a sales component. All the other components and directives they use are essentially the same.

Now you can put such shared features, shared components, directives and other modules even into a shared module which you then import into both other modules, this again is a great step to avoid code duplication and have leaner modules. E.g.

Shared.module.ts

@NgModule({

  declarations: [

    AlertComponent,

    LoadingSpinnerComponent,

    PlaceholderDirective,

    DropdownDirective

  ],

  imports: [

    CommonModule

  ],

  exports: [

    CommonModule,

    AlertComponent,

    LoadingSpinnerComponent,

    PlaceholderDirective,

    DropdownDirective

  ],

})

export class SharedModule { }

Here it is important to note that we have to export all the elment that we want to use in other modules that use our shared module.

recipes.module.ts

@NgModule({

  declarations: [

    RecipesComponent,

    RecipeListComponent,

    RecipeDetailComponent,

    RecipeItemComponent,

    RecipeStartComponent,

    RecipeEditComponent

  ],

  imports: [

    RouterModule,

    ReactiveFormsModule,

    RecipesRoutingModule,

    SharedModule

  ]

})

export class RecipesModule {}

shopping-list.module.ts

@NgModule({

  declarations: [

    ShoppingListComponent,

    ShoppingEditComponent,

  ],

  imports: [

    FormsModule,

    //we can create here this route to avoid create a module of routing shopping list component

    RouterModule.forChild([

      { path: 'shopping-list', component: ShoppingListComponent }

    ]),

    SharedModule

  ]

})

export class ShoppingListModule {}

app.module.ts

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    AuthComponent,

  ],

  imports: [

    BrowserModule,

    FormsModule,

    ReactiveFormsModule,

    HttpClientModule,

    AppRoutingModule,

    RecipesModule,

    ShoppingListModule,

    SharedModule

  ],

  providers: [

    ShoppingListService,

    RecipeService,

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

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

Important concept you gotta understand about modules. You can only define or declare components, directives and pipes once, you can't do that multiple times. You can import a module into multiple imports. So it is fine if we import the router module here in the recipes module and also in the shopping list module, imports are OK but declarations are not. If you declare the shopping list components here in the shopping list module, you must not declare it anywhere else, even if you need it in another module, you must not declare it there.

The solution then instead is to export it here and import that module into another module and that's exactly what we're doing with the shared module.

## Understanding the Core Module

Now that we had a look at feature modules and shared modules, let's have a look at the third important module type you can work with and that would be the core module and of course, let me highlight that all these modules are created in the same way with NgModule, just the things you put in there and how we use them differs and that is what I mean with different module types.

So what's the core module?

The core module is basically there to make the app module a bit leaner, let's say in the app module, we have the app component and here we are providing two services.

Well then we can use a core module to move these services out of the app module into the core module which then in turn is added back to the app module.

The alternative to that would of course be to use provided in in @injectable on the services which means we don't have to add them to providers at all but if you do add it to providers in the app module, well then using a core module can be an idea to again keep your app module leaner.

**It's a pattern you see sometimes, not a pattern you absolutely have to follow and indeed, a pattern that you don't need to use or can't use if you are using provided in in @injectable which I would recommend using anyways.**

Now the idea behind the core module is that we simply provide all these application-wide services that are important right from the start of the application, that we provide them in a separate module which we then simply import into the app module, really just to keep this leaner and to have a way or a place where we can quickly see all the core services of this application.

Now as I mentioned, this does only make sense or is only a pattern you may use if you are providing a service like this, by adding its type to the providers array.

If you are providing a service, like we're doing it for the data storage service, where we pass this object to @injectable, where we set provided in root, then this service is also provided application-wide and you can't and shouldn't add it to providers of the app module and this is indeed the approach I would recommend taking, I would not provide services in the app module, I would instead provide them application-wide like this.

The effect is the same but it keeps the app module cleaner and leaner automatically.

Still, for some things like the interceptors, you have no alternative, you have to add them to providers and here for demo purposes, I also add shopping list service and recipes service.

core.module.ts

@NgModule({

  providers: [

    ShoppingListService,

    RecipeService,

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

  ]

})

export class CoreModule { }

So now the core module here is prepared, you don't need to export your services because services work differently than declarations, only declarations and other modules need to be exported, services are automatically injected on a root level, you don't need to export them to make this happen. So services are simply an exception.

app.module.ts

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    AuthComponent,

    //ErrorPageComponent,

  ],

  imports: [

    BrowserModule,

    FormsModule,

    ReactiveFormsModule,

    HttpClientModule,

    AppRoutingModule,

    RecipesModule,

    ShoppingListModule,

    SharedModule,

    CoreModule

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

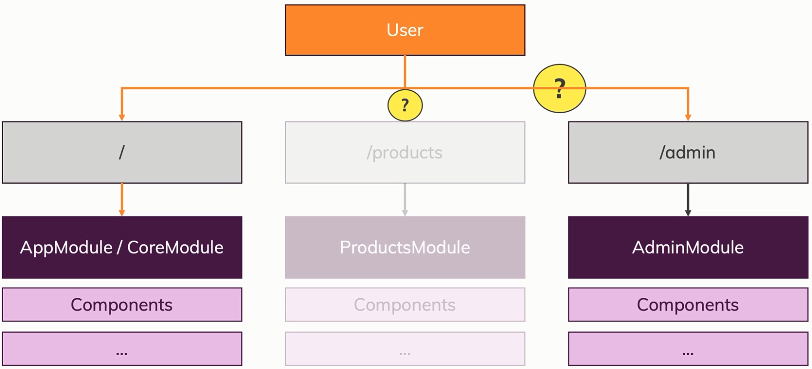
## Understanding Lazy Loading

Now that we covered the important module types you can find in an Angular app, with the feature module being the most important one, let's dive into our first optimization concept because thus far, this entire code splitting of separating our modules into separate modules was only cosmetical, it makes our code in the individual modules leaner and it can help us with maintaining that code but it doesn't influence the performance of our application at all.

This will change now because now we'll have a look at a concept called lazy loading and using multiplemodules, to be precise using multiple feature modules, is a prerequisite for that.

Now what is lazy loading?

Consider this application, we have three routes - our root route, slash nothing, /products and /admin and of course, we have a user visiting this page.



Now every route is associated with a module, our app module and then our different feature modules where each module of course might have multiple components, multiple directives, pipes and so on that belong to it.

Now when our user visits the root route, we of course load that first module and we do the same when we visit the different other pages or do we?

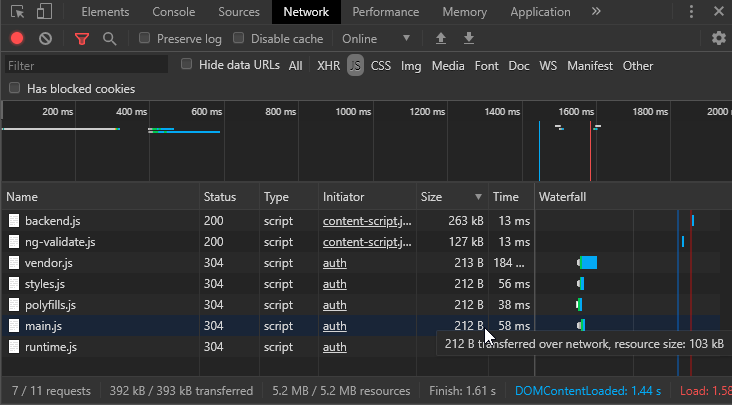
Well actually at the moment, when we're not using lazy loading, whenever we visit any page, we load everything but wouldn't it make more sense to only load products and admin when we actually visit these pages, so that we only load the code that belongs to these areas of our application, that we only load the code that belongs to these areas when we really need it.

That's the idea of lazy loading. With lazy loading, we initially only load our root route content, so only the app module code and the code of all the components that are registered there and we don't load the other modules and only when we visit another module, like the admin module, so the admin route here, we load the admin module code and the code of all the components that belongs to that module and the fact or the advantage of this is that initially, we download a smaller code bundle and we download more code when we need it but initially our app is able to start faster because it has to download and parse less code on the first visit of a certain route and that's the advantage.

So let's now see how we do implement lazy loading in our app and let's see its effect.

## Implementing Lazy Loading

In our browser if we go to the Network tab and you reload the app here on this off page and authenticated. You see we download a bunch of files here. In this place we can check the kb or mb of our files used. Also after the implementation for lazy loading we can check the effect. It seems the main.js file is the file that store our application.



Steps

1. The module that will be lazy loading should have is how module for routing, so, yo should have splited the main router module of the app in routers modules for each logic module.  
   There we have the recipes module and we have its own roots here and the recipes routing module. Now having the roots registered here is an important starting point for a lazy loading to work your feature module needs to bring its own roots.
2. Now in the recipe-routing.module.ts we have to put an empty string '' in the main path.
3. The recipe route path will be now put in the app-routing.module.ts but also using **loadChildren**. And in loadChildren we have to put the fisical **path of the module** without the extension (.t) then put the symbol **#** and the the **name of the module** (RecipesModule).   
   This step in Angular 9 can also be done using promes instead of this string naming convention.
4. Now in the app.module.ts we delete the RecipeModule, beacause now will be loaded in lazy way.
5. It is important to delete all import not used in the app.module.ts on the other hand we don’t going to see an optimization, because all we have in main module will be loaded at beginning.

recipe.module.ts

@NgModule({

  declarations: [

    RecipesComponent,

    RecipeListComponent,

    RecipeDetailComponent,

    RecipeItemComponent,

    RecipeStartComponent,

    RecipeEditComponent

  ],

  imports: [

    RouterModule,

    ReactiveFormsModule,

    RecipesRoutingModule,

    SharedModule

  ]

})

export class RecipesModule {}

recipes-routing.module.ts

const routes: Routes = [

  {

    path: '', component: RecipesComponent, canActivate: [AuthGuard],

     children: [

      { path: '', component: RecipeStartComponent },

      { path: 'new', component: RecipeEditComponent },

      { path: ':id', component: RecipeDetailComponent, resolve: [RecipesResolverService] },

      { path: ':id/edit', component: RecipeEditComponent, resolve: [RecipesResolverService]},

    ]

  }

]

@NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class RecipesRoutingModule {

}

app.module.ts

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent

  ],

  imports: [

    BrowserModule,

    HttpClientModule,

    AppRoutingModule,

    ShoppingListModule,

    SharedModule,

    CoreModule,

    AuthModule

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

app-routing.module.ts

const AppRoute: Routes = [

    { path: '', redirectTo: '/recipes', pathMatch: 'full' },

// { path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule' },

    { path: "recipes",

      loadChildren: () =>

        import("./recipes/recipes.module").then(m => m.RecipesModule)

    },

]

@NgModule({

    imports: [

        RouterModule.forRoot(AppRoute)

    ],

    exports: [

        RouterModule

    ]

})

export class AppRoutingModule {

}

Note: in Angular 9 I have troubles with lazy loding using naming convention, so I fanally have to use promes instead of this string naming convention.

## Preloading Lazy-Loaded Code

I mentioned a way of optimizing lazy loading, which itself of course is already an optimization and with that, I mean how the lazy loaded code is then loaded.

Right now, we load code whenever we need it, so as soon as we visit /recipes, for example after logging in, we bring in the recipes module.

Now the downside of that of course is that this is downloaded and parsed just when we need it, leading to a very tiny delay in our application. We might not see that here because the module is small and the internet connection is fast but the bigger the module and the slower the internet connection or for example if you temporarily have no internet connection, the longer that delay will be and therefore, we can actually tell Angular to preload lazy loaded modules to avoid this delay.

For this, we simply go to our root router module, so here we reconfigure our root routes and you can

pass a second argument to that. This is an object where you can configure that root router and there, you can set up a preloading strategy and you can set this equal to **preload all modules**. The default is no preloading.

But with preload all modules imported and set here as a preloading strategy, you're telling Angular generally we're using lazy loading, so it will not put all that code into one bundle, it will split it as we saw it but it will preload the bundles as soon as possible, so when we are on the auth page, it will already preload recipes and shopping list so that these code bundles are already available when we need them.

The advantage is that the initial download bundles still is kept small because there, that code is not included and therefore the initial loading phase is fast but then when the user is browsing the page and we have some idle time anyways, then we preload these additional code bundles to make sure that subsequent navigation requests are faster, so we're getting the best of both worlds, a fast initial load and thereafter, fast subsequent loads.

app-routing.module.ts

//with loadChildren and this string nomenclature we implement the lazy modules loading

const AppRoute: Routes = [

    { path: '', redirectTo: '/recipes', pathMatch: 'full' },

    { path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule' },

    { path: 'shopping-list', loadChildren: './shopping-list/shopping-list.module#ShoppingListModule' },

    { path: 'auth', loadChildren: './auth/auth.module#AuthModule' },

]

@NgModule({

    imports: [

        RouterModule.forRoot(AppRoute, { preloadingStrategy: PreloadAllModules })

    ],

    exports: [

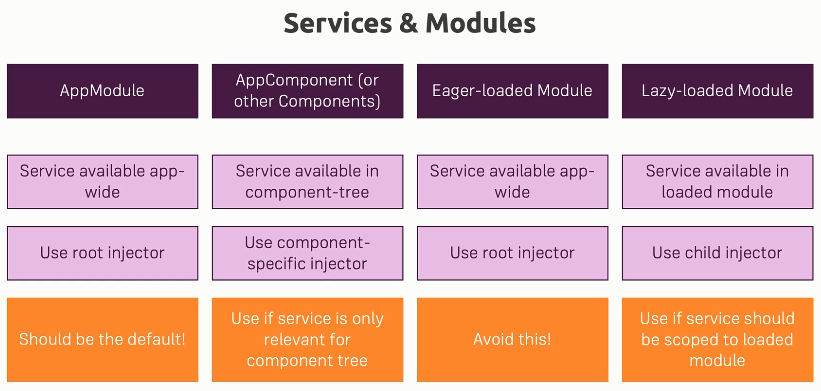
        RouterModule

    ]

})

export class AppRoutingModule { }

## Modules & Services



Now that we learn about lazy loading, I can finally dive into what's special about services and modules because I mentioned that there is something special about that but I could only explain it after diving into lazy loading because it's strongly related to that. I want to dive into where you can provide services and how that affects the instances of the services you're working with.

Now we can provide services in the app module, we can provide services in the app component or other components, if you'll remember the services section, there you also could add a providers key to the component decorator.

We can also load services or add services to providers of any eager loaded module, so in the recipes module before we added lazy loading for example and we're also of course able to add services to providers of lazy loaded modules. Of course, we can also provide services by adding the provided in root configuration in the @injectable decorator and that would be recommended for all services that you planned on providing in app module otherwise.

Now when you provide a service in the app module or with provided in root in @injectable, the service is available application-wide and with the service, I also mean the same instance of that service is available application-wide, you're always working with the same single instance. If you provide a service in a component, no matter if that's the app component or another component, then that service is only available for dependency injection inside of that component tree and then all these components will again share the same instance but we're talking only about that component tree and you provide the service in multiple sibling components of your application, then all these sibling component trees have access to the service but each component tree has access to its own service instance and I mentioned this and showed this in the services module of this course already.

Now the interesting thing is if you do add a service to providers of an eager loaded module, you might think that that service is only available inside of that module and that every module where you add a service to its providers has its own instance of that service but actually, that's not the case.

If a module is eager loaded, everything is bundled together initially and therefore, any services you add to providers in an eager loaded module, let's say the recipes module would be not lazy loaded but still eagerly loaded by importing it into the app module, any service added there will be available application-wide with one and the same instance.

So adding a service to providers of an eagerly loaded module has the exact same effect as adding a service to providers of the app module or as adding provided in root in @injectable. A huge difference can be seen if you add a service to providers of a lazy loaded module though. There the service is only available in that loaded module and it gets its own instance. If you do provide a service both in the app module and in the lazy loaded module, then the service is of course available application-wide but the lazy loaded module will get a separate instance, not the same instance you use in the rest of the app.

Sometimes this can be desirable but in other cases, it might lead to a strange behaviour where you would have expected to work with one and the same instance when you didn't. The reason for that is that when you do provide in the app module or in @injectable, you use the so-called root injector, so the root dependency injection mechanism of Angular. When you do provide it in a component, you use the component specific injector, which is not the root injector and therefore not the entire application has access to that. In an eagerly loaded module, again the root injector is used automatically, with a lazy loaded module, a separate child injector is created by Angular for that module and therefore, that module has its own instance of that service. Therefore, the default should be that you use the app module or again, provided in root in @injectable. You of course might have scenarios where a component tree needs its own service, for example if that component tree somehow has a service that only matters for these components, it doesn't matter application-wide, then there is no need to provide the service application-wide, you can instead provide it in that component. Eagerly loaded modules should pretty much never provide services, you should avoid this because the effect is the same as in the app module but it's kind of unexpected and it's harder to detect services there and therefore other developers diving into your code might look for a certain service, might look for it in the app module, not find it there and therefore be confused whether this service is now intended to be available to the entire app or only to that module.

Now in a lazy loaded module, you can of course add a service to providers but only do this if you deliberately want to have a separate instance of that service there, otherwise you can have strange bugs. So as a rule, generally you want to ensure that services are always available application-wide by using @injectable, provided in root or by adding them to providers in the app module unless you have a strong reason for adding them only in a component tree or in a lazy loaded module, never add services in an eager loaded module.

Now let's see that in practice.

## Loading Services Differently

Now for demo purposes we have created this service that just print the current message and then the previous one.

app/ logging.service.ts

/\*\*

 \* This class is just for demo purposes, it just to see how services in modules work depending of its declaration

 \*/

@Injectable({ providedIn: 'root'})

export class LoggingService {

  private lastLog: string;

  printLog(message: string) {

    console.log(message);

    console.log(this.lastLog);

    this.lastLog = message;

  }

}

Now we inject this service in app.component.ts

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

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

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit{

  constructor(private authService: AuthService,

              private loggingService: LoggingService) {}

  ngOnInit(): void {

    this.authService.autoLogin();

    this.loggingService.printLog('Hello from AppComponent ngOnInit');

  }

}

And in shopping-list.component.ts

export class ShoppingListComponent implements OnInit, OnDestroy {

  ingredients: Ingredient[];

  private subscription: Subscription;

  constructor(private slService: ShoppingListService,

              private loggingService: LoggingService) { }

  ngOnInit(): void {

    this.ingredients = this.slService.getIngredientes();

    this.subscription = this.slService.ingredientChanged.subscribe(

      (ingredients: Ingredient[]) => {

          this.ingredients = ingredients;

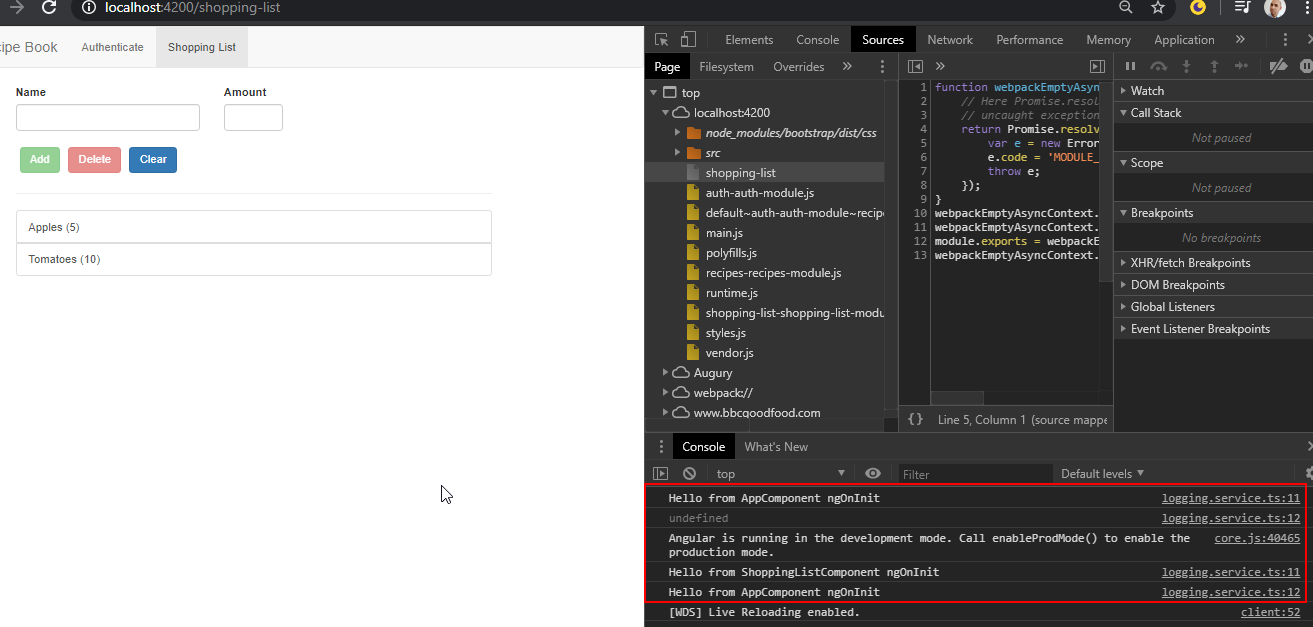
        }

    );

    this.loggingService.printLog('Hello from ShoppingListComponent ngOnInit');

  }

Then restart the page and if we click in loggin page we have null on previos message, but if we click on shopping list the we have loggin on previos message, this means we are using the same service for the entire app.



Now if whe comment in the service the provider and provided the service in the app.module.ts then we going to have the same result, the service is the same for the entired app.

// @Injectable({ providedIn: 'root'})

export class LoggingService {

  private lastLog: string;

  printLog(message: string) {

    console.log(message);

    console.log(this.lastLog);

    this.lastLog = message;

  }

}

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent

  ],

  imports: [

    BrowserModule,

    HttpClientModule,

    AppRoutingModule,

    SharedModule,

    CoreModule,

  ],

  providers: [LoggingService],

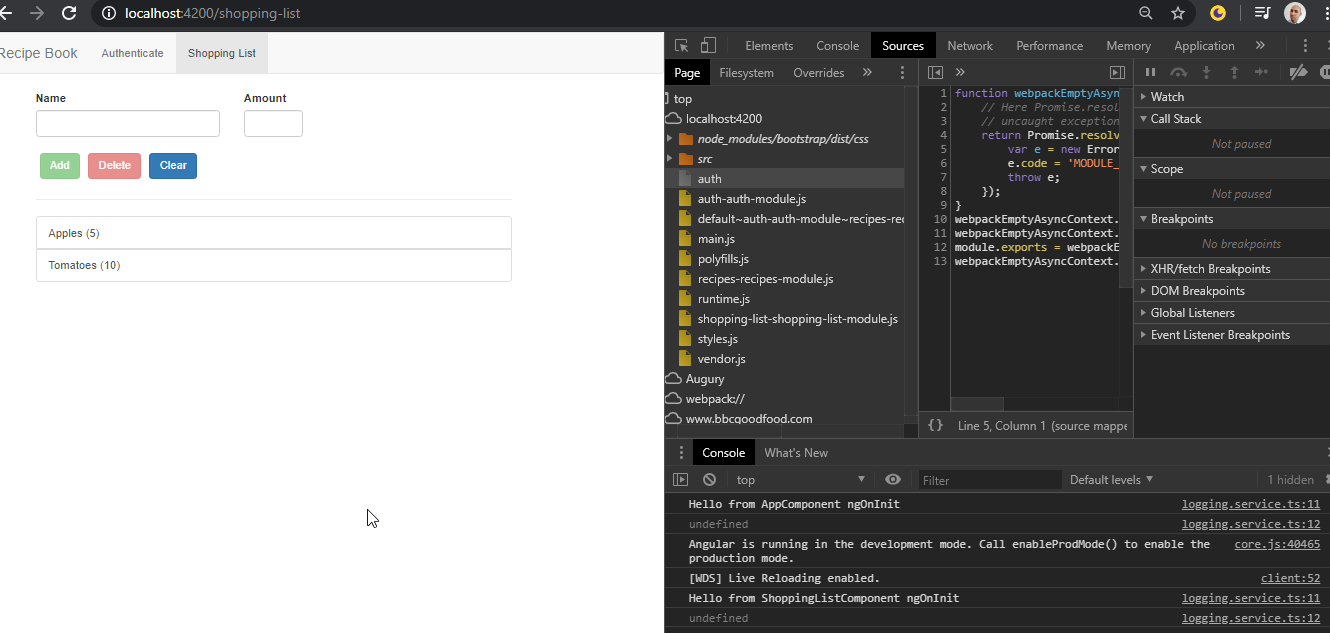
  bootstrap: [AppComponent]

})

export class AppModule { }

In other example, no if we comment the provider loggingService in the app.module.ts and add to the CoreModule (this is an eagerly loaded module, that is eagerly loaded because we import it here into the app module) the service, then the the effect will be the same.

Now, if provide the loggin service to the app.module.ts and in the shopping-list.module.ts. So now I'm providing it in a lazy loaded module, the shopping list module and in an eagerly loaded module, my app module. Let's go to authenticate and reload. Oh now the output is different, because we're now using two different objects, two different instances of that same service.

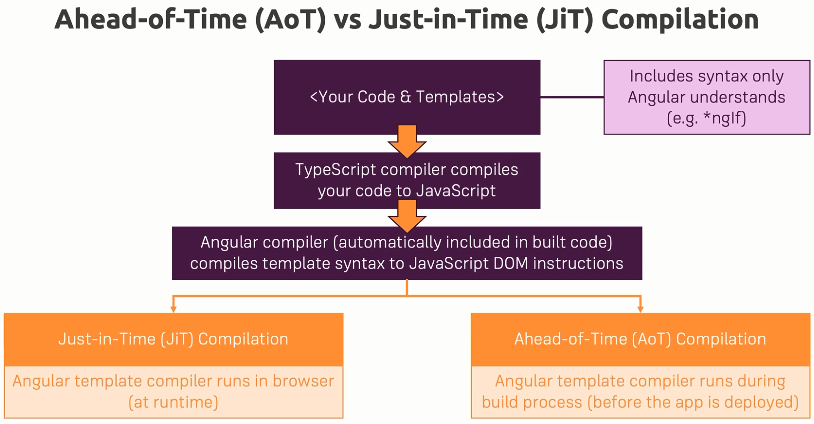


Now if you need that behaviour, definitely use it, nothing wrong with that, this is not a bug, this is an intended feature but in general, you should think twice before you provide a service in any other module than the app module because often, it's not the wanted behaviour and you then suddenly have strange bugs when you're using different instances in your application.

So if you need that behaviour, use it but generally be aware that there are differences in the instances you get when you provide services in lazy loaded modules.

I know this can be difficult to wrap your head around but this is a crucial thing and a common source of bugs and therefore the rule of thumb is that you provide services through @injectable or in the app module, if you do it differently, you should have a strong reason and you should either be able to rule out that this module you're adding the service in is loaded lazily or you deliberately want multiple instances.

## Ahead-of-Time Compilation



We're nearing the end of this module and I want to concluded by diving into one of our optimization technique we can and should perform before we ship our app to production.

So before we deploy our built angular app onto a server and that is ahead of time where is this just in time compilation.

What's that.

Well in our angular application we're of course writing some code are angular code in our template code and our templates of course include special angular syntax like energy f and g for NS on only angular understands this D templates we're creating.

That's not a one to one snapshot that's going to get rendered into the real Dom instead angular will pass our templates and then update the real Dom based on the instructions we placed in our templates so all our components and our directives are in the end translated to commands that will be used to alter the real Dom javascript the browser doesn't understand our templates angular does.

So we write our templates for angular because angular understands all these features so whilst we use the types of compiler to compile the types good code to JavaScript code which of course is understood by the browser we also have another compiler in that build process working behind the scenes as part of Angie's serve which is the angular compiler that is included in the angular framework.

It's a crucial part of the angular framework and it's actually quite big.

The Anglo compiler itself is written in Javascript. It's relatively large and it's part of the angular application you're seeing in the browser with N.G. serve so that angular compiler takes to compile javascript code and your templates and actually translates all the logic there into concrete instructions that update the real Dom D by default happens all in your browser because the angular compiler is part of the angler code you're shipping.

This process is called just in time compilation because the angular templates are compiled just in time when the rendering occurs in your browser.

So the angular compiler runs in the browser at runtime.

The obvious downside of this is that this compilation process takes time.

It's super fast but still it's a small performance hit anti angular compiler so the entire code that is just responsible for compiling your templates is not that small.

It's actually quite large and it's part of your angular application even though it has no purpose it has nothing to do with your business logic other than bringing it onto the screen which arguably is important but still a bit annoying that it has to be part of your application.

The good thing is it doesn't have to be part of your application since the angular compiler is responsible for converting your template code into your template instructions into javascript commands.

The browser understands we can do that during development as part of the build process and that process is called ahead of time compilation because we compile the angular templates ahead of the execution time.

Then the angler template compiler runs during the build process before the app is deployed and not in the browser now.

The energy Cerf process as of now by default uses the Justin Time compiler because that's better for debugging and fast updating of the running application and there's nothing wrong with that during development.

We have no problem with shipping a too big application where the compiler is part of the code because we're not optimizing the code size during development.

We want to have a good development experience with rich error messages and so on but as soon as we're preparing ourselves for production and for building our angular app for production to then upload it on a server we want to optimize our code as much as possible and shrink it to a small of a bunch less possible.

And we do that with a command you'll also see in the deployment section I quit.

**ng build --prod**

This command takes your angular application and now does not spin up a development server where you can see it but it actually builds your entire app into a few files which you can then deploy.

Let's now have a look at what this ng built command generated it generated a new dist folder. This folder control contains a folder with your project name. And then there you got multiple files which are basically which basically contain the content of your application so your code files.  
But now bundled and optimized and decreased in size as much as possible to be as small as possible.

Now here if we have a look at these files they're way smaller they shrank significantly in size because of all these optimizations and ahead of time compilation is one important optimization because it removes the angular compiler from the final bundle which is super important and huge of course.

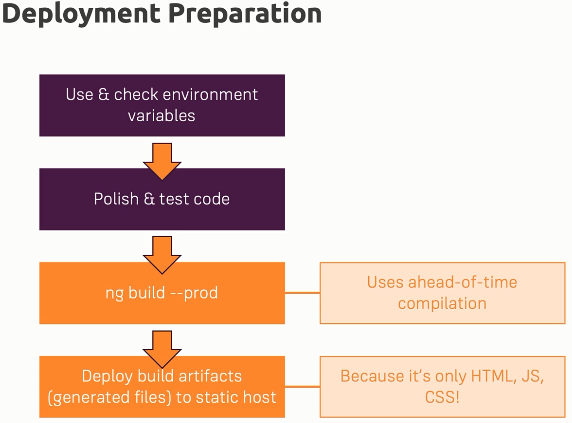
So definitely always build your application with ng build dashes broad before you want to deploy it somewhere.

But speaking off that deployment is also something we'll focus on next in this course.

**Useful Resources:**

* Official Docs: <https://angular.io/guide/ngmodules>
* NgModules FAQ: <https://angular.io/guide/ngmodule-faq>

# Section 23: Deploying an Angular App



So how should you prepare your Angular application for deployment?

There are a couple of steps through which you should walk and through which I'll walk you now and the first is that you should **use and check environment variables**. That's something we haven't done thus far and I will show you what I mean with that point and how you can use environment variables, what that is and why you might want to use that.

Well of course you should also **polish and test your code** and that is what we did throughout this course already, for example in the optimizations and modules course section, I already showed you some important polishments or code optimizations like lazy loading.

So it's of course things like that which you should implement and obviously, you should test your application, run it, try out different things, try working with it as your users would work with it.

Once you're done with all of that, you can **build your application for production** and that means that you run a command that takes all your TypeScript code, compiles it to Javascript, then bundles all your code together, translates it into Angular language so to say, so that the Angular renderer can work with it in the browser and bundle everything up and optimize it to be as small as possible because the application you're shipping has to be downloaded by every user using your website and therefore, it should be as small as possible so that your users have to download as little as possible and you do all that with the ng build --prod command.

You already saw that in the optimizations module because you also need to run this to implement ahead of time compilation and that's just one of the many optimizations the Angular CLI does for you behind the scenes.

Now once you're done with that, once you built your app, you can **deploy** these build artifacts, these generated files to a static host and now what is a static host? A static host is a web server that is capable of serving HTML, Javascript and CSS but that's not capable of running any server-side language, like PHP or NodeJS.

Why?

Because you only need HTML, Javascript and CSS because an Angular app is made up of just that. Of course you can have some back-end code, for example we are using Firebase in this course. This could be your own back-end which you wrote with your own language and you should definitely check out my MEAN course regarding that because there I do show you how to build your own back-end with NodeJS as well but if you're just working with Angular, if you're just building a single page application and you're not working on the back-end on your own, then you in the end have just HTML, Javascript and CSS and all you need for that is a web server that is capable of serving these files.

So now, let's walk through these steps together, most importantly the build and the deployment steps of course and let's see how that works.

## Using Environment Variables

Environment variable is a feature that's actually built into every project you build with the Angular CLI. In the source folder, you have your app folder, there you also have an environments folder. In that environments folder, you have two files: environment.ts and environment.prod.ts

Now we're not using these files but what you could do here, you see you have an exported constant in the environment.ts and in the environment.prod.ts file, it's the same constant with the same name, you can add key-value pairs to that constant.

For example if you were using an API that requires some API key and actually, we are using something like this and hence I will show you how to use this environment variable in a second, if you are using something like this, then you can store that API key here, you can import that environment constant here into your files to use that API key and what the CLI, the Angular CLI does for you is that it will automatically swap these two files when you are building for production.

So during development, this environment.ts file and the value you have in here will be used and during production, this constant will be used instead and this is swapped automatically, you don't need to write any special code for this behavior to occur.

Now why is this useful?

Well you might have certain values that differ from development to production. You might have some API keys that are meant to use during development and then you have different keys, which maybe are more restrictive, for production or any other value.

You can store anything in here, any value which you need in your other files and you have the option of using different values for production and development. E.g.

environment.ts

export const environment = {

  production: false,

  FIREBASE\_PROJECT\_KEY: 'AIzaSyDKNsEn6rrkJE7XUMh2R3e74OW8YfwUh7k'

};

auth.service.ts

import { environment } from '../../environments/environment';

@Injectable({ providedIn: 'root' })

export class AuthService {

  user = new BehaviorSubject<User>(null);

  private tokenExpirationTimer: any;

  constructor(private http: HttpClient, private router: Router) { }

  signup(email: string, password: string) {

    return this.http.post<AuthResponseData>(

      'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key='

+ environment.FIREBASE\_PROJECT\_KEY,

      {

        email: email,

        password: password,

        returnSecureToken: true

      }

    )

    .pipe(

      catchError(this.handleError),

      tap(resData => {

        this.handleAuthentication(resData.email, resData.localId, resData.idToken, +resData.expiresIn);

      })

    );

  }

## Deployment Example: Firebase Hosting

With the environment variables added, it's now time to build the project and you already saw the command for that, it's **ng build --prod** and this builds the entire Angular application for production, optimizes everything and gives you an output which you can deploy onto a static server.

Now again, let me highlight here that what this does is, it compiles your TypeScript code to Javascript, that's one important step, it also compiles all your templates to Javascript instructions so that in the end, only Javascript code is generated, Javascript code that runs in the browser and contains all the logic for updating the DOM in the way you defined it in your multiple components and all the logic you added in your application.

Now after running ng build --prod, you get this dist folder and in there, you have a folder of your project name and in that project name folder, now you have bunch of assets that you might have included and you also got your main polyfill runtime file and a couple of other files, the exact names will differ for you but you've got a couple of Javascript files which essentially contain your Javascript code, your Angular app.

So now, we just have to deploy this, how does this work?

As I mentioned, you now need a static website hosts and of course there are many options for that. You can google for a static website host to find many articles and examples, to learn where you can host such a static website.

I particularly want to highlight AWS S3 which uses a special service of the AWS, so of the Amazon Web Services, that allows you to host static websites, that is definitely worth a look. You need an AWS account for that but thereafter, it's pretty straightforward to use, pretty powerful and pretty popular and I also want to point you at Firebase Hosting.

Now you don't need to have the rest of your back-end on Firebase to use that. It's a coincidence that here in this course we used Firebase for the authentication and for the database because it was easy to use and it is intended to be an all-in-one package but even if you use your own back-end or if you use a totally different API, you can always use Firebase hosting to host static websites and indeed, it is what I want to use here because it's particularly easy to use.

Now here you can learn more about that in the official docs and you see how you can get started with Firebase hosting:

<https://firebase.google.com/docs/cli>

You need the Firebase CLI for that which simply is a tool you have to install to automatically deploy your Firebase application. For that, you need to install the Firebase CLI first, for which you need NodeJS and NPM which we already got because the Angular CLI needed that as well and therefore you can dive right in and run this commands on any console:

Commands of firebase used:

npm install -g firebase-tools

firebase login

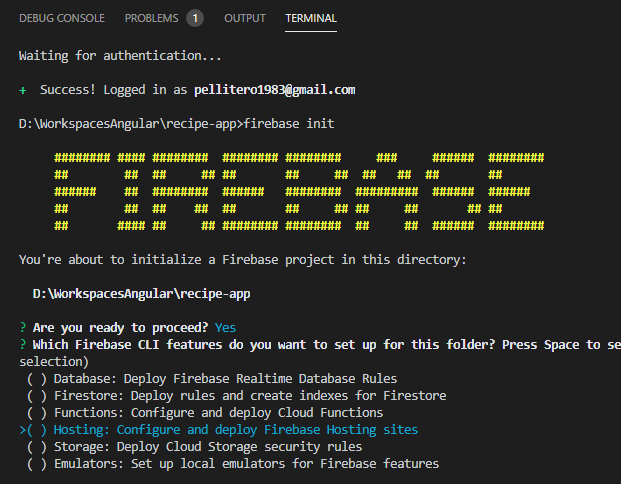
firebase init

firebase deploy

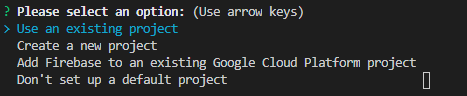
Thir firsth commnad (npm install -g firebase-tools) let us install the firebase toll to to automatically deploy your app.

The second you have to run Firebase login, to log into your Firebase account.

The next step then is to run Firebase init to connect this project here with one of your Firebase projects and for that, you now should be in your Angular project folder, so cd into it if you're not using the terminal built into your IDE. Firebase init now asks you what you want to use, so which Firebase feature you want to use because you could use the Firebase CLI for other things too, not just for hosting but I'm only interested in hosting.



The we select the firebase project we want to use.





The we set the path were our app is, and how we built the app for production we have it in dist/recip-app folder



Now you're asked whether you want to configure this as a single page app and you should definitely choose y for yes here to ensure that all requests that you sent to that back-end are always redirected to index.html because that is a common gotcha indeed, you need to have a server configured such that it always serves the index.html, no matter which URL you entered into the browser and the reason for that is that by default, the server always handles an incoming request first and if it's a URL that's unknown to the server, you would normally get an error.

Of course we have the Angular router handling our different routes but that, always keep in mind, runs

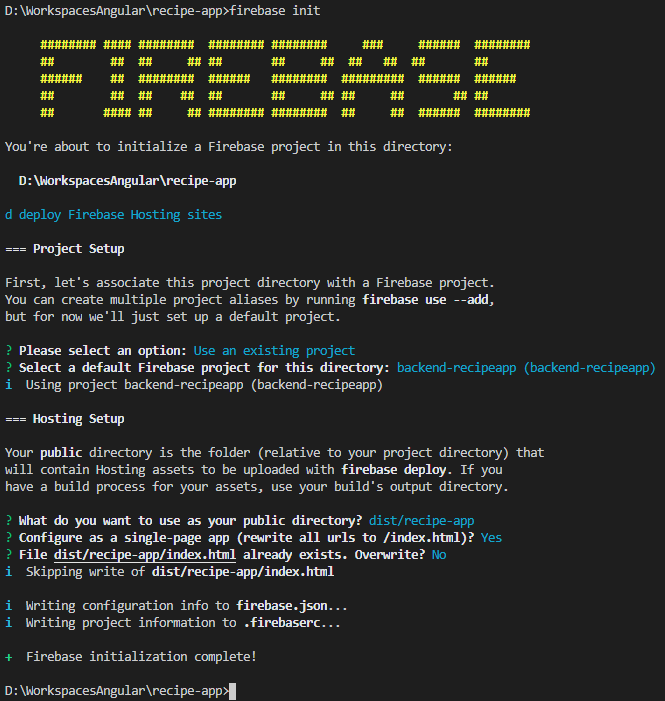
on the client, so in the browser and that only gets a chance to run if the server serves your app.

So any requests have to reach the server first, there they are all redirected to index.html, which in the end is our starting point for the single page application and since any request, no matter which URL you entered reaches that page, that HTML page, now the Angular app can take over for any incoming request with any URL on your server and can then use the Angular router to load the correct content in your Angular app, that's why this redirection is important, so choose y for yes here.

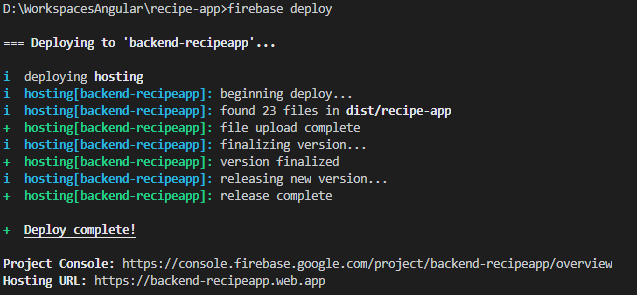
Now the existing HTML file should not be overwritten, so choose no here and with that, you're done.



All process



Finally we use the command (firebase deploy) to deploy our app in firebase. E.g.



The next and last step then is that you run Firebase deploy and this will now deploy the content of this folder onto Firebase. Once it is done, it should give you a URL where you can find your application.

<https://backend-recipeapp.web.app>

So let's grab that URL and enter it in the browser and there, you should be on your application that should work just the way it worked locally but now on a real server. And we're still interacting with our back-end API and everything but now, this is our deployed application.

## Server Routing vs Browser Routing

When deploying your Angular app, it's really important to make sure that your server (like S3) is configured to always serve the index.html file.

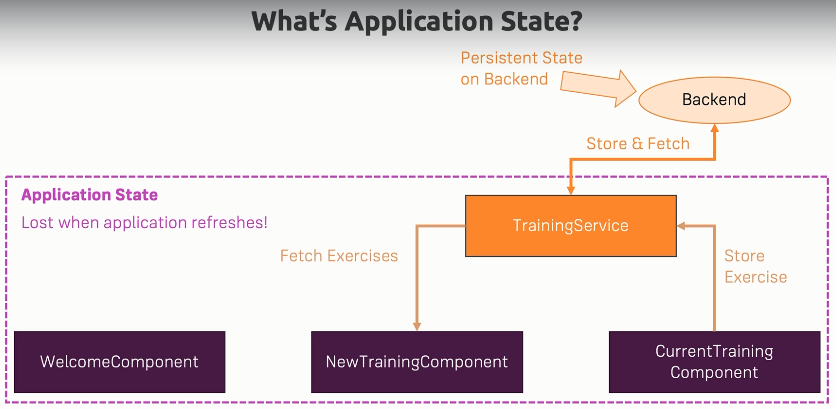
Here's why: <https://academind.com/learn/angular/angular-q-a/#how-to-fix-broken-routes-after-deployment>

# Section 24: Bonus: Working with NgRx in our Project

Now in this course, you already learned a lot about Angular and actually, we already went through all the important fundamentals you've got to know. Now this is a bonus module and it is about a topic, NgRx, which is also maintained or developed by parts of the official Angular team, which can help you with state management in bigger Angular applications.

So let's first of all have a look at which problem it tries to solve and then we'll have a detailed look at how you use NgRx to manage your application state in a maybe more streamlined and elegant way.

## What is Application State?



Of course the obvious first question is, what is application state?

Why do we need a state management solution? Doesn't Angular already have one? What is this state thing?

Well, consider this simple Angular application, it's not the application we worked on in this course but of course this could be just another application you have where you have the welcome component, then a couple of the training components here because maybe you're building like a training application, something where people can do workouts and save them and you have your training service.

Now in this application, the service would be there to manage the core data these different components need to work correctly, for example you could be fetching exercises or once the user is done with the exercise, you could store that exercise result here. Well and all that data which kind of is important for your application because this data is in the end controlling what's visible on the screen, that is your application state and this application state is lost whenever your application refreshes, so whenever you reload your Angular app, your website.

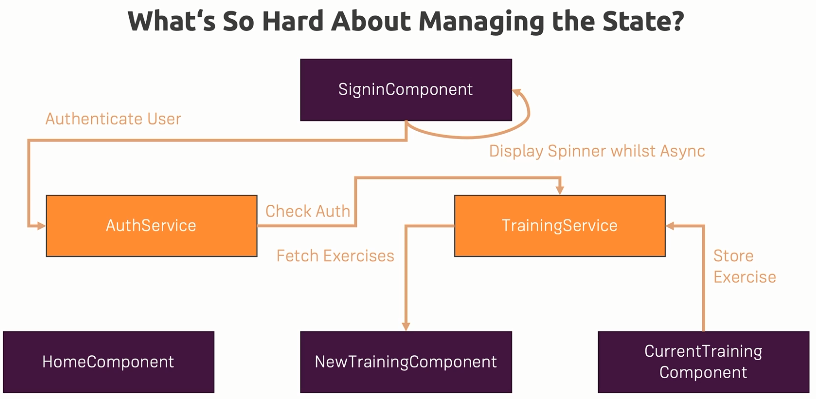
Now this simply is the case because this entire data here is managed in a service and in components but all that together is just a Javascript app in the end, right, your entire Angular application is compiled into a Javascript bundle in the end. So it's all in Javascript and therefore, it's managed in memory by Javascript and therefore whenever your page is reloaded, the Angular app restarts and the other app is basically dumped, it's lost and the memory is cleared, that's why it's refreshed but that's not the problem NgRx helps you deal with because that is not per say a problem, that's just how Javascript works and to solve that refreshing problem, we typically work with a backend.

Like in this course, when we added our recipes and we don't have a database on the backend where we store that data, well then we would lose all our recipes and all the changes we made to them whenever our application reloads and therefore, we typically also interact with such a backend where we store and fetch data too.

This is so-called persistent state and it's persistent because it survives our page refreshes as it is stored on the backend. So state in a nutshell could be translated with data that's important to your application and that influences what's visible on the screen.

State is not just data like stored exercises or recipes, state could also be things like "this app is currently waiting for some data to be fetched", so the app is in a loading state, it's showing a spinner maybe.

So basically, any data, any information that controls what should be visible on the screen, that is state. OK, so that is state and thus far in this course we already managed such state, typically in services but tiny pieces of state are even getting managed inside of components.



Now what's the problem with this approach?

Well simpler apps and that includes our course project to be very honest so I should maybe say smaller apps. Typically are fine with how we already manage state, by using components and services and actually, you can get very far with this approach only, even bigger apps can be managed with just services and components but the bigger your app gets and the more complex your state gets, so maybe an app like this where we have a lot of components and service that interact with each other, if you have an app that becomes bigger and bigger and your state depends on many components at the same time and many components depend on the service and so on, then you could end up in state management nightmare.

So you could end up in an application where it's hard to maintain the app because it's hard for you to tell where exactly am I managing this piece of information and maybe you're even changing a piece of information, like the isLoading state of a component accidentally from another place in the application simply because your code is structured such that this is possible.

Well we already have a remedy to that and that is rxjs, at least partly. RxJS already allows us to create a streamlined state management experience. With RxJS and there specifically with subjects, we can react to user events in the user interface or to application events like some data fetching finishing, we can react to such a state changing event, so to an event where we want to update some data or some information in our app by using observables or as I said, subjects and we can emit or next a new data piece there and maybe use operators to even transform data in the way we want it and then listen to such state changes in other parts of the application where we need it to then update the UI and we're doing this already as well.

If I dive back into the code and back into the recipes service, there indeed, recipes are managed with the help of these recipes changed subject here.

Now keep in mind that this subject in the end is used to emit a new copy of all recipes whenever the recipes did change, so for example when we overwrote our existing recipes or whenever we added a recipe.

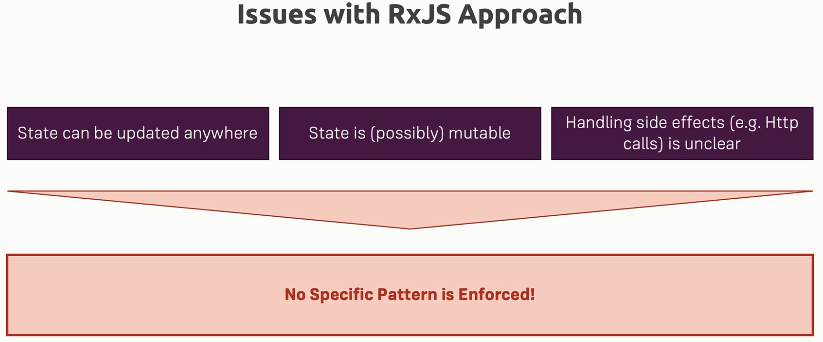
And why does this matter?

Because this then can be used in other components, like the recipe list component where we set up a subscription to that recipes changed subject to update our recipes in that component and therefore indirectly update the user interface of that component whenever our recipes do change.

So here we are already using a quite good pattern of managing our state by using subjects and using that push subscribe mechanism where we have a clear stream of data, a unidirectional data stream. It's for example not really possible or not easily possible to update our recipes outside of that predefined flow of using these methods in the recipes service where we then emit a new event where we then subscribe to.

Yes, you can always hack your way around this but of course if you want to break your app, you can always do that. But what I want to say here is that this approach of managing state using RxJS and subjects is already pretty good, it might be all you need and that includes bigger apps, so even bigger and very large apps, you can definitely work with a setup that only uses services, components and such subjects to manage this state but that can of course be tricky because it really forces you to have an eye on providing a good structure and a clean setup and ensuring that really all services and all pieces of state are implemented in a good way and that is where NgRx steps in.

## What is NgRx?

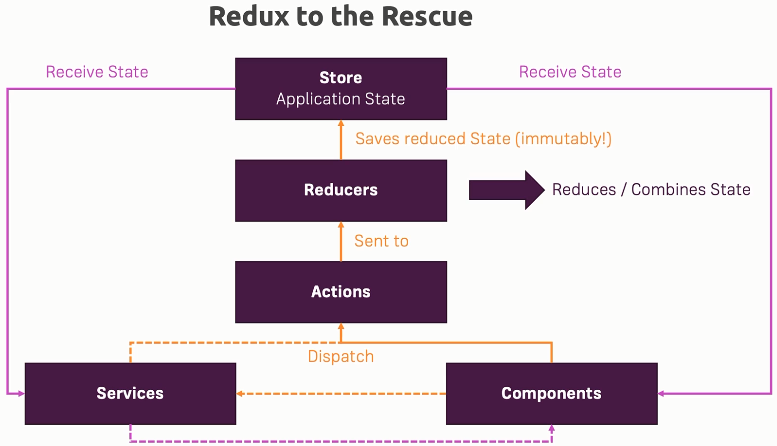


This RxJS driven approach has some issues as I outlined, to be precise you could still have an app where state can be updated from anywhere because you maybe failed to set up a clear flow of data.

Your state might also possibly be mutable, which means that whilst you might have only a couple of places where your state is intended to change, your code in there might not force you to state the old data by overwriting it with new data and that is what I would recommend doing though because otherwise Angular sometimes doesn't pick up some changes to your state because of the reference type nature of objects and arrays in Javascript, where if you only change a property of an object, the overall object didn't change and therefore such a state change might not get picked up.

And in addition, handling side effects and that means things like HTTP requests, it's unclear where this should happen - should you write the code for sending them in a component? Should you do it in a service?

And yes, all these problems can of course be solved and you can have a clear route and a clear approach regarding all these things but especially if you're working on a bigger app with a bigger team, it can be hard to enforce the best practices across all team members here and therefore, enforcing a specific pattern might be hard because Angular by default doesn't enforce one and that is where NgRx may help you or to be precise, Redux may help you.



Now what is Redux, I just introduced NgRx, what is Redux?

Well if you have a ReactJS background, you might already know Redux, otherwise you will learn about it now. Redux is a state management pattern, it's also a library that helps you implement that pattern into any application.

The idea behind Redux, so behind that approach for managing state and always keep in mind, state is just data or information in your app, so the idea behind managing that state with Redux is that you have one central store in your entire application that holds your application state, so think of that as a large Javascript object that contains all the data the different parts of your application need, of course kind of categorized in properties that might then hold nested objects but in the end that's the idea, have one large data store. Your different parts of the application, services and components, can still interact with each other but they receive their state from that store, so that store is the single source of truth that manages the entire application state.

Now of course, state is not just received, you sometimes also need to change the state, for example when you add a new recipe and for that in the Redux world, you dispatch so-called actions.

An action is the end also just a Javascript object with an identifier basically identifying the kind of action you want to perform and optionally, a payload.

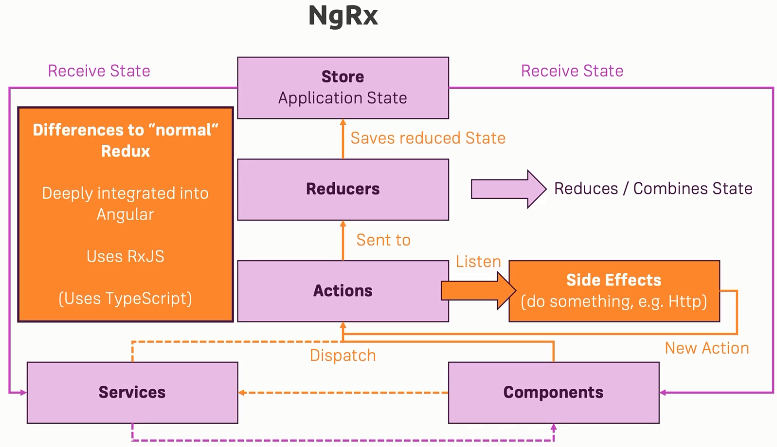
So if that action needs some extra data to complete, for example if you're creating a new recipe, you would need to attach that recipe data to the action, then you can have that data in the action that is supported by Redux.

Now that action doesn't directly reach the store, instead it reaches a so-called reducer. Now a reducer is just a function, a Javascript function that gets the current state which is stored in the store and the action as a input, passed in automatically by the Redux library if you use that and then in the reducer, you can have a look at the action identifier, find out which kind of action it is, for example add a recipe or delete the recipe and then perform code on the state which you also got as an argument to update that state, however in an immutable way, so by copying it and then changing the copy because ultimately, the reducer returns a new state, so it returns a copy of the old state that will have been changed according to the action and that state which is returned by the reducer then is forwarded to the application store where this reduced state, again the state was edited immutably, so without touching the old state instead copying it and then editing the copy, so where this reduced state is then overwriting the old state of the application.

That is the Redux pattern and as you can see, it enforces a pretty clean and clear flow of data.

We only have one place in the app where we store our data and we only edit it through actions and reducers and we received the state through so-called subscriptions which we can setup.

Now if you were using Angular, you could also use the Redux library, it's not restricted to be used with ReactJS only but NgRx is in the end just Angular's implementation of Redux.



Now it comes with some differences or Angular specific things, for example the left orange box here, it's deeply integrated into Angular, so it for example comes with injectable services so that you can easily access your application store in any part of your app by simply injecting it.

It also embraces RxJS and observables, so all the state is managed as one large observable you could say and that also can yield certain advantages, for example you can use all the observable operators to then edit the state you are fetching in the component where you need it in the way you need it there, this will not change the store state of course and it of course also supports TypeScript.

Now in addition, there is one extra part of that state management library which is about side effects. So NgRx also gives you a tool that makes working with side effects easier and side effects, just to bring that back in memory, are things like sending HTTP requests. NgRx also helps with that, traditionally in Redux, that was kind of an open question, it was unclear where you should send such side effects, where you should send such HTTP requests, so typically asynchronous code was always a problem because reducers, I can say that already, only may execute synchronous code, so you must not and you can't send HTTP requests from inside a reducer function for example and therefore it always was kind of an open question in Redux where you should do such asynchronous tasks, NgRx ships with a built-in solution and you will learn about that solution here too.

So that was lots of theory, time to finally get back to the code and start implementing that pattern and that package there step-by-step.

## Getting Started with Reducers

So let's start implementing NgRx in our course project and I chose the course project for that because I think you'll learn the most if you really see how it's implemented in a real application.

Now again, I stressed this on the slides already but it is important to me, you could definitely leave the app as it is.

It's not bad, not a bad practice or anything like that if you manage your state with services and with RxJS subjects, this is absolutely fine and there is no reason to change this if you don't want to or if you see no advantage in NgRx but my recommendation would be that you go through this module where we have a look at NgRx in great detail and you then simply decide for yourself, do you like that pattern?

Do you maybe see its advantages in larger apps and do you want to use it there and then you will already have learned it and that of course is what this module and this course is all about.

So to get started, we first of all need to install NgRx because it is an extra package which you need to install through NPM, as I mentioned, it is maintained by parts of the Angular team and also by the community.

So you install it by running npm install in your project folder, --save to also add an entry to package.json and then it's @ngrx/store.

**npm install --save @ngrx/store**

This is the first package we need and I say first package because we'll later also need some other packages but this is the core of NgRx which you always need and which gives you the core features you need to get started.

Steps to add Redux to our app:

1. We create a file named shipping-list.reducer.ts in the shopping-list folder and inside of the file we create a function that receive two arguments.

That function and that's now important receives two arguments automatically because we will soon add it such that it's automatically managed by the NgRx package and this NgRx package will in the end be the part that calls this function, not right now but soon and the NgRx package will therefore pass these arguments into this function. So we can rely on two arguments being passed into this function and that's the **state**, so the current state before it was changed, it's changed by the reducer so obviously we get it before it was changed and the second argument is the **action** because keep in mind, it's the action that triggers the reducer and in the end, the state update, so these two things are passed in by NgRx. Now inside of that function, we can now find out which kind of action it is and update the state.

shopping-list.reducer.ts

import { Ingredient } from '../shared/ingredient.model';

import { Action } from '@ngrx/store';

const initialState = {

  ingredients: [

    new Ingredient('Apples', 5),

    new Ingredient('Tomatoes', 10)

  ]

}

export function shoppingListReducer(state = initialState, action: Action) {

  switch(action.type) {

    case 'ADD\_INGREDIENT':

      return {

        ...state,

        ingredients: [...state.ingredients, action]

      };

  }

}

in this function we pass the state paramether state = initialState, this is just TypeScript, just in case the paramether were null que set the constants value initialState.

Now to check different kinds of possible actions, you could add multiple if statements and run different codes depending on which action you have or since we have multiple possible conditions, you can use a switch case statement and then on the action, you have to check the type of that action. Now to get better TypeScript support, you can add a type to the action argument and the type would be Action, where action is actually imported not from this package but where action is actually imported from @ngrx/store. This exports an action interface and this simply is an interface that forces any action to have a type property, which is why we can safely check it here.

So now I'm switching on the action type and now we can just register different cases, so different action types we want to handle.

Now it's totally up to you as a developer of this application which action types your application has and here, we could start with a simple one, if we have a look at the shopping service maybe we start with adding an ingredient because that will change our ingredients state, it will add a new one.

So we could start with a simple state which we now give an identifier or type is the identifier but the value of type is a string. So here, we named ADD\_INGREDIENT. Now it's absolutely up to you what the value here is but the convention is to use an all uppercase text which kind of is clear about what this action will do.

So we get the old state and we need to return a new state, now what you could do is you could take the state which has the ingredients property and then simply use push to add a new ingredient and that would be totally wrong because state changes with NgRx always have to be immutable, which means you must not edit the existing or the previous state, it's absolutely forbidden and a bad practice if you do so.

So never touch the existing state, instead return a new object which will replace the old state and to not lose all the old data, copy the old state with the **spread operator**. That is next generation Javascript syntax which is also supported by TypeScript, it essentially pulls out all the properties of the old state and adds these properties to this new object.

So we copied all the old state and now I want to overwrite the ingredients, so I'll add the ingredients property and now we can assign a new value to that.

## Understanding & Adding Actions

We added our first reducer and thus far, NgRx is not doing anything for us but we're getting there. The next step is that we also take care about the actions that we want to have or about the one action which we support right now, adding an ingredient. We want to be able to dispatch this and to dispatch this and to there also reliably dispatch it in the same way by having the right identifier where we rule out typos and by adding the right payload ingredient that can be added here, we should also standardize this action creation process or how such an action looks like because at the moment, I'm just checking for this identifier here but if I now would dispatch this action from anywhere in my code, I would have to make sure that I type this same identifier and as you might imagine, if you're in a hurry, if your application is bigger, it's easy to forget a character and then you have a bug that's super hard to spot, your app does not work correctly and you waste hours looking for that stupid typo. To rule that out, NgRx also kind of at least pushes us into a direction where we have a clear pattern about creating and providing our actions as well.

Now for that, I'll create a new file next to the shopping list reducer and that is my shopping-list.actions.ts file.

Now since the actions and the reducer kind of belong together and belong both to the NgRx setup, I'll add a new subfolder here in the shopping list folder and I'll name it store because it's all related to the @ngrx/store.

So let's move the reducer and the actions file into that store folder now and now let's work on the shopping list actions file.

Now in that file, I first of all want to export my identifiers or my single identifier I have thus far in a way that we can simply import it and we don't have to type it manually, to rule out typos.

So in the shopping list actions file, I'll therefore export a constant which I also name add ingredient, though that name is totally up to you and then I assign a value of add ingredient.

Now this is a convention which I recommend following, you use a constant whose name is the same name as the one of your identifier, which in turn should be descriptive about what this action does and you then simply store your string identifier in that constant and by exporting it here in the actions file, we can import it into our files, like in our shopping list reducer file where we need that action and therefore since we never have to type it again and we just import it and the IDE will yell at us if we mess up the import, it won't yell at us if we mess up a string, therefore for all these reasons we then have a secure way where we rule out typos in the action identifiers.

So now, I will import add ingredient from the shopping list actions file here and instead of using the string like this down there, I just use add ingredient like this and this now refers to that string.

Now that's the first step but of course this step doesn't ensure that we for example have the right payload.

The next step therefore is that we export another thing that describes our action and our action, if we go back to reducer, is of course not that string.

Keep in mind the action is an object which needs to have a type property, so the action actually is an object, not a string, it is of type action and action is imported from @ngrx/store. So it probably doesn't hurt if we grab that action import and we import it here in the shopping-list.actions.ts file as well and that what we export here should be our action and that should be a class therefore, since the action is an object and objects can be created based on classes, that should have a descriptive name and I'll name it add ingredient, now not all uppercase because class names should be pascal case, so using this convention but it still is of course belonging to that action, I think that's pretty clear by the name.

Now it's a class and that class needs to implement the action interface we're importing from @ngrx/store. Action as I mentioned earlier already is an interface, not itself an object and this interface being implemented into add ingredient now forces us to structure the add ingredient class in a certain way. To be precise, we need to have a type property in the add ingredient class and that type is the identifier of this action.

Now this will always be an add ingredient action, so type here should simply be add ingredient, so our string from up there and now an extra annotation which we should add here is readonly, that is a TypeScript feature which indicates to TypeScript that this must never be changed from outside and if we ever try so from anywhere in our code, we'll get an error and this simply enhances type safety and ensures that we don't mess up with our pattern.

Now as I mentioned, this action needs more than just a type though, it also needs the ingredient that should be added and therefore, we can add more. We can add a second property, the payload and that should be of type ingredient and for that, you should also add an import to the ingredient model, like this.

Now with that, we finished our action definition. Now it still might be hard to see all these pieces come together but we're almost there. Let's now use this action in the shopping list reducer, let's extract the payload and let's then finally work on registering the application store and also on selecting our state and dispatching actions.

shopping-list.actions.ts

import { Action } from '@ngrx/store';

import { Ingredient } from 'src/app/shared/ingredient.model';

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

export class AddIngredient implements Action {

  readonly type = ADD\_INGREDIENT;

  payload: Ingredient;

}

shopping-list.reducer.ts

import { Ingredient } from '../../shared/ingredient.model';

import \* as shoppingListActions from './shopping-list.actions';

const initialState = {

  ingredients: [

    new Ingredient('Apples', 5),

    new Ingredient('Tomatoes', 10)

  ]

}

export function shoppingListReducer(state = initialState, action: shoppingListActions.AddIngredient) {

  switch(action.type) {

    case shoppingListActions.ADD\_INGREDIENT:

      return {

        ...state,

        ingredients: [...state.ingredients, action.payload]

      };

    default:

      return state;

  }

## Setting Up the NgRx Store

With the action added, back to the shopping list reducer. Here I'm already handling the add ingredient case which I'm importing from the shopping list actions.

So now with that all done, how can we now use the reducer and our nice setup action to finally setup that application store which it is all about to then also use that? Because thus far, we spend a lot of work on implementing something which has no effect, well that's about to change. To add our application store.

In our app module let's import something from @ngrx/store and that something is the store module, that helps us with setting up that application-wide @ngrx/store. Now store module as the name suggests is an Angular module and therefore we should add it here, maybe below the other Angular modules we add the store module.

We don't just add it like this though, we also need to tell the store module, we need to tell NgRx what makes up our store, which reducers are involved and for that, we call for root here and to for root, we pass a so-called action reducer map. Now that sounds more complicated than it is, in the end it's a Javascript object where you can define any identifier of your choice, so maybe shopping list and then the reducer that belongs to that identifier and in my case that's the shopping list reducer function which needs to be imported from the shopping list folder, there the store folder and there, the shopping list reducer file, so that shopping list reducer function we worked on over the past lectures. This function is now assigned as a value to this shopping list key and that key name is totally up to you but of course it should be descriptive about which feature of the app this reducer belongs to and it is the reducer for our shopping list part of the app, therefore it makes sense to name this shopping list as well.

With that, we told NgRx where to find our reducer and therefore now when the app restarts, even though we don't see it yet, NgRx will take that reducer into account and set up an application store for us where it registers this reducer, so now any actions that are dispatched will reach that reducer and to see that initial application state, we can now start fetching some data from that state.

app.module.ts

import { StoreModule } from "@ngrx/store";

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent

  ],

  imports: [

    BrowserModule,

    HttpClientModule,

    AppRoutingModule,

    StoreModule.forRoot({shoppingList: shoppingListReducer}),

    SharedModule,

    CoreModule,

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

## Selecting State

With the store module registered here in app module and it being connected to our shopping list reducer, as I said in the last lecture, finally NgRx gets going.

Time to use it and I want to use it in my shopping list component. In the shopping list component, the shopping list service now will not be used because instead we going to use NgRx.

In our constructor I will inject something new and that will be our application store. If you remember the earliest slides of this module, I mentioned that we're not using Redux but NgRx because it gives us some extra features and deeper integration into Angular and for example, it gives us an injectable store that makes it easy for us to access our application state which is stored in that store after all. Now the type here which is important for the injection is store and store needs to be imported from @ngrx/store, so this import needs to be added in the shopping list component and therefore, we now have the store but we still have an error here because store turns out to be a generic type, you need to describe how your application store looks like and now that is the structure you set up in the app module.

StoreModule.forRoot({shoppingList: shoppingListReducer})

Here, it's this ActionReducerMap which you set up here, this object where you have a key of your choice and then the reducer, this is in the end the structure of your store. Of course, in your application, you might and in this module you will have at some point a store that is made up of many different pieces, later for example we'll have a shopping list area and a recipes area and that's all merged together into one store and therefore, our type is in the end a description of the different parts we have in the store.

So back in the shopping this component, here the type would be a Javascript object where we have a shopping list key and now this has to be the name you chose here on the left as a key and the type of the data stored in that shopping list area is now not your reducer function but what this reducer function yields and what does our shopping list reducer yield in the end?

Well, a state of this type. It yields a state of a type object with an ingredients key where this ingredients key holds an array of ingredients. So finally the private object in the constructor will be:

private store: Store<{shoppingList: {ingredients: Ingredient[]}}>

Well now, I want to use my store here and get access to the ingredients stored in that store and that will then, as it turns out, handle both these cases.

Now on this store, which is this injected store service provided by NgRx, you have a select method and that helps you select a slice of your state.

Now that slice is identified as a string and here, I automatically get type support, it's shopping list.

this.ingredients = this.store.select('shoppingList');

So here I am selecting that shopping list part of my global store, again right now the shopping this part is the only thing I have in this store but as this will change throughout the app, you of course need to tell NgRx into which part of the store you are interested in at this point.

So here, I select my shopping list and this turns out to give us an observable. So we change our ingredients fields to be and observable with same structure of the Store object.

ingredients: Observable<{ingredients: Ingredient[]}>;

We don't need to unsubscribe from this selection here but now, NgRx and Angular do that for us.

Now ingredients is an observable and not an array of ingredients anymore and that of course will break our template because if we have a look at the shopping list component, we actually do loop through all the ingredients and that of course only works with an array and not with an observable but Angular has us covered there, you might remember the async pipe from the pipes section.

I can wrap ingredients with parentheses and ingredients again is an observable and add the async pipe like this. What this will do is that Angular will subscribe to ingredients for us and therefore it will manage that subscription for us and we don't need to do that on our own and will resolve that and in the end therefore, we can access ingredients on that value that Angular will ultimately give us once it has resolved this and once it got a value from RxJS, from that observable and then ingredients again is an array through which we can loop.

\*ngFor="let ingredient of (ingredients | async).ingredients;

Finally our files changed are:

shopping-list.component.ts

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

import { Ingredient } from '../shared/ingredient.model';

import { ShoppingListService } from './shopping-list.service';

import { Subscription, Observable } from 'rxjs';

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

import { Store } from '@ngrx/store';

@Component({

  selector: 'app-shopping-list',

  templateUrl: './shopping-list.component.html',

  styleUrls: ['./shopping-list.component.css']

})

export class ShoppingListComponent implements OnInit, OnDestroy {

  ingredients: Observable<{ingredients: Ingredient[]}>;

  private subscription: Subscription;

  constructor(private slService: ShoppingListService,

              private loggingService: LoggingService,

              private store: Store<{shoppingList: {ingredients: Ingredient[]}}>)

  {}

  ngOnInit(): void {

    this.ingredients = this.store.select('shoppingList');

    // this.ingredients = this.slService.getIngredients();

    // this.subscription = this.slService.ingredientChanged.subscribe(

    //   (ingredients: Ingredient[]) => {

    //       this.ingredients = ingredients;

    //     }

    // );

    this.loggingService.printLog('Hello from ShoppingListComponent ngOnInit');

  }

  onEditItem(index: number) {

    this.slService.startedEditing.next(index);

  }

  ngOnDestroy(): void {

    // this.subscription.unsubscribe();

  }

}

shopping-list.component.html

<div class="row">

    <div class="col-xs-10">

        <app-shopping-edit></app-shopping-edit>

        <hr>

        <ul class="list-group">

            <a

              class="list-group-item"

              style="cursor: pointer;"

              \*ngFor="let ingredient of (ingredients | async).ingredients; let i = index"

              (click)="onEditItem(i)"

            >

            {{ingredient.name}} ({{ingredient.amount}})

            </a>

        </ul>

    </div>

</div>

## Dispatching Actions

Where do I want to dispatch actions?

This is like send and evento to Angular redux to tell him please store this element.

Well if we're talking about the shopping list part of the app, that would be in a shopping-edit component, in the shopping-edit component, in the end we got our code to dispatch actions whenever we're done, so whenever we're submitting our form to either update or add an ingredient, these are the two places where we change our ingredients that are displayed and therefore the two places where we change our shopping list state and therefore should change the store.

So here we inject in the constructor the angular store field of type shoppingList. And with that we have to call a function named dispatch, that going to receive and object of the same type we declare to store field. In this case we pass the class AddIngredient that store the action and the payload field, that hold our ingredients. E.g.

shopping-list.actions.ts

import { Action } from '@ngrx/store';

import { Ingredient } from 'src/app/shared/ingredient.model';

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

export class AddIngredient implements Action {

  readonly type = ADD\_INGREDIENT;

  constructor(public payload: Ingredient) {}

}

shopping-edit.component.ts

import { Store } from '@ngrx/store';

import \* as shoppingListActions from '../store/shopping-list.actions';

@Component({

  selector: 'app-shopping-edit',

  templateUrl: './shopping-edit.component.html',

  styleUrls: ['./shopping-edit.component.css']

})

export class ShoppingEditComponent implements OnInit, OnDestroy {

  @ViewChild('f') slForm: NgForm;

  subscription: Subscription;

  editMode = false;

  editedItemIndex: number;

  editedItem: Ingredient;

  constructor(

    private slService: ShoppingListService,

    private store: Store<{ shoppingList: { ingredients: Ingredient[] } }>

  ) { }

  /\*\*

   \* Function to Add or Update an Item of the shopping list

   \* @param form

   \*/

  onSubmit(form: NgForm) {

    const value = form.value;

    const newIngredient = new Ingredient(value.name, value.amount);

    if(this.editMode) {

      this.slService.updateIngredient(this.editedItemIndex, newIngredient);

    } else {

      // this.slService.addIngredient(newIngredient);

      this.store.dispatch(new shoppingListActions.AddIngredient(newIngredient))

    }

    this.editMode = false;

    this.slForm.reset();

  }

}

## Multiple Actions

So let's continue working on this and let's continue by having a look at how we can add multiple ingredients because that's also a case we have, so not just one ingredient but multiple ingredients at once.

For that we go to the shopping-list.actions.ts file and we add a new class and a new constant for this new actions, will named AddIngredients.

Also we need to create a generic type that encapsulate both actions. For that we create in type script the type ShoppingListActions that could store an AddIngredient  or AddIngredients object.

E.g. shopping-list.actions.ts

import { Action } from '@ngrx/store';

import { Ingredient } from 'src/app/shared/ingredient.model';

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

export const ADD\_INGREDIENTS = 'ADD\_INGREDIENTS';

export class AddIngredient implements Action {

  readonly type = ADD\_INGREDIENT;

  constructor(public payload: Ingredient) {}

}

export class AddIngredients implements Action {

  readonly type = ADD\_INGREDIENTS;

  constructor(public payload: Ingredient[]) {}

}

export type ShoppingListActions = AddIngredient | AddIngredients;

Now in our shopping list reducer que use the generic type and create a new case that going to store the list of ingredients. E.g.

shopping-list.reducer.ts

import { Ingredient } from '../../shared/ingredient.model';

import \* as shoppingListActions from './shopping-list.actions';

const initialState = {

  ingredients: [

    new Ingredient('Apples', 5),

    new Ingredient('Tomatoes', 10)

  ]

}

export function shoppingListReducer(state = initialState, action: shoppingListActions.ShoppingListActions) {

  switch (action.type) {

    case shoppingListActions.ADD\_INGREDIENT:

      return {

        ...state,

        ingredients: [...state.ingredients, action.payload]

      };

    case shoppingListActions.ADD\_INGREDIENTS:

      return {

        ...state,

        ingredients: [...state.ingredients, ...action.payload]

      };

    default:

      return state;

  }

}

Finally we have to dispatch this event to store the state instead calling a service. E.g.

recipe.service.ts

import { Store } from '@ngrx/store';

import \* as shoppingListActions from '../shopping-list/store/shopping-list.actions';

@Injectable()

export class RecipeService {

  recipesChanged = new Subject<Recipe[]>();

  private recipes: Recipe[] = [];

  constructor(

    private slService: ShoppingListService,

    private store: Store<{ shoppingList: { ingredients: Ingredient[] } }>

  ) {}

  addIngredientsToShoppingList(ingredients: Ingredient[]) {

    //this.slService.addIngredients(ingredients);

    this.store.dispatch(new shoppingListActions.AddIngredients(ingredients))

  }

}

## First Summary

So we started working with NgRx and at this point before we continue and we move more of the functionality we have in our application into the NgRx world, I want to sum up what we did thus far and it's really important to follow along here so that it becomes clear which strengths and weaknesses NgRx has and why you might want to use it or might not want to use it.

So we added NgRx to our application by including the store module and calling for root. For root then needs a map, so basically an object that tells NgRx which reducers we have in our application because all these reducers and the state they generate make up the store of this application, make up the NgRx store.

Now we add a feature for that with an identifier of our choice and then the reducer for that feature. Now that reducer as you learned here is just a function and now the important thing about that reducer function is that NgRx will automatically pass in the current state and an action it received, so it will execute this function whenever a new action is received and then this function executes and gets both the current state and the action that was received. In the reducer, I then use a switch case statement and this is a typical setup you see in the Redux and NgRx world, where we check the type of action we got because action is actually a Javascript object with a type property and depending on the type which we have, we return a new state and that is how a reducer function always works, data in data out, no asynchronous code, it's all synchronous, so we only have synchronous code in here and we always return a new object which will be used to replace the old state for this part of the application, so for the shopping list here and this return state is what NgRx will in the end register for the shopping list slice of the overall AppState, of the App store it manages here. So this is what we added, we added this reducer and regarding the actions, we also added an actions file where we for one defined unique identifiers for all our actions, these are simply strings that identify each action and then the action itself is not just this identifier but it's a complete object based on classes we define in here.

Each action needs to have a type property where we do store the string identifier for the action but in addition, we might also have a payload, so a property which can be set to attach data to that action and we needed that for example for adding an ingredient, though we also have an action, the delete action which has no extra data because it doesn't need any.

Now this entire setup was really a lot of work, we have to be honest there, adding the reducer, adding the actions, injecting the store, dispatching actions, that is a lot of overhead work and using the normal shopping list service definitely was easier there and therefore using a service with subjects is a more than fine alternative to using NgRx.

However, once you've gone through that first initial setup and you got NgRx going, you saw that it's pretty easy to add new actions, new state slices and therefore, it's really easy to grow our state and to manage everything in our app or thus far in this app, in a part of our app with NgRx.

So it's mainly that initial setup that requires a lot of work, once that is set up it's actually quite fast to work with NgRx.

Nonetheless, it's of course always up to you whether you want to add NgRx or whether you say in this application, I'm fine with subjects and normal services and as I mentioned multiple times, that would be fine.

**Finals changes to the app**

We have introduced other changes to be more generic the redux function for the shopping-list component, so he are the final state before transform the others components of the app.

shopping-list.actions.ts

import { Action } from '@ngrx/store';

import { Ingredient } from 'src/app/shared/ingredient.model';

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

export const ADD\_INGREDIENTS = 'ADD\_INGREDIENTS';

export const UPDATE\_INGREDIENT = 'UPDATE\_INGREDIENT';

export const DELETE\_INGREDIENT = 'DELETE\_INGREDIENT';

export const START\_EDIT = 'START\_EDIT';

export const STOP\_EDIT = 'STOP\_EDIT';

export class AddIngredient implements Action {

  readonly type = ADD\_INGREDIENT;

  constructor(public payload: Ingredient) {}

}

export class AddIngredients implements Action {

  readonly type = ADD\_INGREDIENTS;

  constructor(public payload: Ingredient[]) {}

}

export class UpdateIngredients implements Action {

  readonly type = UPDATE\_INGREDIENT;

  constructor(public payload: Ingredient) {}

}

export class DeleteIngredients implements Action {

  readonly type = DELETE\_INGREDIENT;

}

export class StartEdit implements Action {

  readonly type = START\_EDIT;

  constructor(public payload: number) {}

}

export class StopEdit implements Action {

  readonly type = STOP\_EDIT;

}

export type ShoppingListActions = AddIngredient | AddIngredients | UpdateIngredients | DeleteIngredients | StartEdit | StopEdit;

shopping-list.reducer.ts

import { Ingredient } from '../../shared/ingredient.model';

import \* as shoppingListActions from './shopping-list.actions';

export interface State {

  ingredients: Ingredient[],

  editedIngredient: Ingredient,

  editedIngredientIndex: number

}

export interface AppState {

  shoppingList: State

}

const initialState: State = {

  ingredients: [new Ingredient('Apples', 5), new Ingredient('Tomatoes', 10)],

  editedIngredient: null,

  editedIngredientIndex: -1

}

export function shoppingListReducer(state: State = initialState, action: shoppingListActions.ShoppingListActions) {

  switch (action.type) {

    case shoppingListActions.ADD\_INGREDIENT:

      return {

        ...state,

        ingredients: [...state.ingredients, action.payload]

      };

    case shoppingListActions.ADD\_INGREDIENTS:

      return {

        ...state,

        ingredients: [...state.ingredients, ...action.payload]

      };

    case shoppingListActions.UPDATE\_INGREDIENT:

      const ingredient = state.ingredients[state.editedIngredientIndex];

      const updatedIngredient = {

        ...ingredient,

        ...action.payload

      };

      const updatedIngredients = [...state.ingredients];

      updatedIngredients[state.editedIngredientIndex] = updatedIngredient;

      return {

        ...state,

        ingredients: updatedIngredients,

        editedIngredientIndex: -1,

        editedIngredient: null

      };

    case shoppingListActions.DELETE\_INGREDIENT:

      return {

        ...state,

        ingredients: state.ingredients.filter((ig, igIndex) => {

          return igIndex !== state.editedIngredientIndex;

        }),

        editedIngredientIndex: -1,

        editedIngredient: null

      };

    case shoppingListActions.START\_EDIT:

      return {

        ...state,

        editedIngredientIndex: action.payload,

        editedIngredient: {...state.ingredients[action.payload]}

      };

    case shoppingListActions.STOP\_EDIT:

      return {

        ...state,

        editedIngredient: null,

        editedIngredientIndex: -1

      };

    default:

      return state;

  }

}

app.module.ts

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

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

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

import { StoreModule } from "@ngrx/store";

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

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

import { CoreModule } from './core.module';

import { HeaderComponent } from './header/header.component';

import { SharedModule } from './shared/shared.module';

import { shoppingListReducer } from './shopping-list/store/shopping-list.reducer';

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent

  ],

  imports: [

    BrowserModule,

    HttpClientModule,

    AppRoutingModule,

    StoreModule.forRoot({shoppingList: shoppingListReducer}),

    SharedModule,

    CoreModule,

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

shopping-list.component.ts

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

import { Ingredient } from '../shared/ingredient.model';

import { Subscription, Observable } from 'rxjs';

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

import { Store } from '@ngrx/store';

import \* as fromShoppingList from './store/shopping-list.reducer'

import \* as shoppingListActions from './store/shopping-list.actions';

@Component({

  selector: 'app-shopping-list',

  templateUrl: './shopping-list.component.html',

  styleUrls: ['./shopping-list.component.css']

})

export class ShoppingListComponent implements OnInit, OnDestroy {

  ingredients: Observable<{ingredients: Ingredient[]}>;

  private subscription: Subscription;

  constructor(private loggingService: LoggingService,

              private store: Store<fromShoppingList.AppState>)

  {}

  ngOnInit(): void {

    this.ingredients = this.store.select('shoppingList');

    this.loggingService.printLog('Hello from ShoppingListComponent ngOnInit');

  }

  onEditItem(index: number) {

    this.store.dispatch(new shoppingListActions.StartEdit(index));

  }

  ngOnDestroy(): void {

  }

}

shopping-edit.component.ts

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

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

import { Subscription } from 'rxjs';

import { Ingredient } from 'src/app/shared/ingredient.model';

import { Store } from '@ngrx/store';

import \* as shoppingListActions from '../store/shopping-list.actions';

import \* as fromShoppingList from '../store/shopping-list.reducer'

@Component({

  selector: 'app-shopping-edit',

  templateUrl: './shopping-edit.component.html',

  styleUrls: ['./shopping-edit.component.css']

})

export class ShoppingEditComponent implements OnInit, OnDestroy {

  @ViewChild('f') slForm: NgForm;

  subscription: Subscription;

  editMode = false;

  editedItemIndex: number;

  editedItem: Ingredient;

  constructor(private store: Store<fromShoppingList.AppState>) {}

  ngOnInit(): void {

    this.subscription = this.store.select('shoppingList').subscribe(stateData => {

      if(stateData.editedIngredientIndex > -1) {

        this.editMode = true;

        this.editedItem = stateData.editedIngredient;

        this.slForm.setValue({

          name: this.editedItem.name,

          amount: this.editedItem.amount

        });

      } else {

        this.editMode = false;

      }

    });

  }

  ngOnDestroy(): void {

    this.subscription.unsubscribe();

    this.store.dispatch(new shoppingListActions.StopEdit());

  }

  /\*\*

   \* Function to Add or Update an Item of the shopping list

   \* @param form

   \*/

  onSubmit(form: NgForm) {

    const value = form.value;

    const newIngredient = new Ingredient(value.name, value.amount);

    if(this.editMode) {

     this.store.dispatch(new shoppingListActions.UpdateIngredients(newIngredient));

    } else {

     this.store.dispatch(new shoppingListActions.AddIngredient(newIngredient));

    }

    this.editMode = false;

    this.slForm.reset();

  }

  onDelete() {

    this.store.dispatch(new shoppingListActions.DeleteIngredients());

    this.onClear();

  }

  onClear(){

    this.slForm.reset();

    this.editMode = false;

    this.store.dispatch(new shoppingListActions.StopEdit());

  }

}

recipe.service.ts

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

import { Subject } from 'rxjs';

import { Recipe } from './recipe.model';

import { Ingredient } from '../shared/ingredient.model';

import { Store } from '@ngrx/store';

import \* as shoppingListActions from '../shopping-list/store/shopping-list.actions';

import \* as fromShoppingList from '../shopping-list/store/shopping-list.reducer'

@Injectable()

export class RecipeService {

  recipesChanged = new Subject<Recipe[]>();

  private recipes: Recipe[] = [];

  constructor(

    private store: Store<fromShoppingList.AppState>

  ) {}

  addIngredientsToShoppingList(ingredients: Ingredient[]) {

    this.store.dispatch(new shoppingListActions.AddIngredients(ingredients))

  }

….

}

## And Important Note on Actions

Before we tackle side effects and how we handle such effects with NgRx, two important things. First, let's have a look at our reducer real quick and there, at this default return state case.

Thus far, I mentioned that this is important for initializing the state and that is true. When NgRx starts up, thanks to us initializing the store module in the app module, it essentially sends one initial action to all reducers and since this action has an identifier we don't handle anywhere here and we'll soon see which identifier this is, no worries.

Since this is an identifier we don't explicitly handle, we make it into the default case, therefore we return the state and since we have no prior state when this first action is emitted, we therefore take this initial state.

So that's one important note but another thing about what I just said is also important. This initial action dispatched automatically reaches all reducers. Well that is not just a case for the initial action, any action you dispatch, even your own ones, not just that initial one, so **any action you dispatch by calling dispatch always reaches all reducers**.

It does not just reach the auth reducer because you are dispatching it in the auth service, how would NgRx know that this is your intention?

It would be dangerous if that would be the result because that means you could never reach another reducer with actions dispatched here in the auth service and there certainly are scenarios where you might want to reach another reducer. So therefore, any action you dispatch always reaches all reducers.

This has an important implication, it's even more important that you always copy the old state here and it's super important that you return this default state here, so the unchanged state because if a shopping list related action is dispatched, it still reaches the auth reducer. In there of course, we have no case that would handle that action, hence all these code snippets here don't kick in but therefore the default case kicks in and handles this and there, we absolutely have to return our current state, otherwise we would lose that or cause an error and I can simply show that by adding a console log and just print the state in the main function of any reducer of the app, so that we can see what's inside there.

So that's the first important thing, the next important thing is that since our dispatched actions reach all reducers, you really have to ensure that your identifiers here are login across the entire application because these auth related actions here are not just handled by the auth reducer, they also reach the shopping list reducer.

Now of course, we in this application we have identifiers that don't really interfere, add ingredient and login are quite different but for bigger applications, it is of course possible that you have duplicate identifiers across different parts of your app.

Now since they have to be unique for the entire application because actions reach the entire application, I would recommend a technique called prefixing. That means you edit your identifiers here a little bit to have one extra piece of information that really makes them unique. In NgRx world, you typically use square brackets and then you add the name of the feature you are dealing with, so in this case, that would be shopping list and then you could still use the uppercase notation here or you also use a normal text casing here so to say. Now I will convert like this, it's not a must do but it is recommended especially for bigger apps to avoid clashes, now of course any prefix that ensures uniqueness would be fine but here I'll just implement a pattern you'll also find in the official NgRx docs. E.g shopping-list.actions.ts

// export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

// export const ADD\_INGREDIENTS = 'ADD\_INGREDIENTS';

export const ADD\_INGREDIENT = '[Shopping List] ADD\_INGREDIENT';

export const ADD\_INGREDIENTS = '[Shopping List] ADD\_INGREDIENTS';

export const UPDATE\_INGREDIENT = '[Shopping List] UPDATE\_INGREDIENT';

export const DELETE\_INGREDIENT = '[Shopping List] DELETE\_INGREDIENT';

export const START\_EDIT = '[Shopping List] START\_EDIT';

export const STOP\_EDIT = '[Shopping List] STOP\_EDIT';

auth.actions.ts

export const LOGIN  = '[Auth] LOGIN';

export const LOGOUT = '[Auth] LOGOUT';

My Note: I think its is important to know when should be used Redux in an application, because I don’t like that any dispatch event be reached by all reducers of the app. I think this coul be a performance problem. I have found this article that could be ok:

<https://blog.angular-university.io/angular-2-redux-ngrx-rxjs/>

## Exploring NgRx Effects

First of all, let's understand what side effects are and we can best see this in the auth service. Side effects are basically parts in your code where you run some logic that of course is important for your application, otherwise you wouldn't do it but that's not so important for the immediate update of the current state.

So for example here, this HTTP request, of course the result of that matters, it decides whether we did successfully create a new user or not but for this process, where we start the sign-up process, this is not important.

So we could basically split this up into two actions - start sign-up and sign-up success for example or even three actions, start sign-up, sign-up success and sign-up error.

So the start sign-up action would then be an action where the response here doesn't matter and this is just a side effect for us.

The sign-up success action also is one where the response matters but where we don't want to deal with sending this, so the request is kind of in-between the different actions, each action only cares about a part of that this request while it's on its way, it's not really something our state cares about or our store cares about, we don't want to save that, it's just some side effect code we run. The same for local storage, this actually does not happen asynchronously. Accessing local storage happens synchronicity so it's done instantly and we could therefore do it in the reducer, only async code is forbidden in there and it will break it.

This, adding this line to a reducer wouldn't break the reducer but it would be considered a bad practice because again, accessing the local storage and writing something to it or getting something from there is really just a side effect of the application.

Of course the result of that, especially when you're getting data, matters to us but the process itself is not that interesting to us.

So things like that are side effects and whilst we could manage some of them, like local storage in the reducer, others like async coordinate or HTTP requests can't even be managed there and either of the two things would be a bad practice, even if we could technically handle HTTP requests in there.

That's why there is a separate package, also provided and maintained by the NgRx team that helps us with dealing such side effects while still staying in this NgRx world and this package now has to be installed of course. We do that with:

**npm install --save @ngrx/effects**

@ngrx/effects is a package that gives us tools for elegantly working with side effects between actions which we dispatch and receive so that we can keep our reducers clean and still have a good place for managing these side effects.

It is important to me here to point out that of course you could also go for an approach where you still have a service for sending the HTTP request and for working with local storage and then you have your reducer and NgRx for managing your current state, so basically the setup we already have here.

You could also keep that and as so much else with NgRx, it comes down to your personal preference and likings but here I think it is good, especially for learning about NgRx if we move as much as possible into the NgRx world and therefore if we use @ngrx/effects for managing our HTTP requests and local storage access.

### Defining the First Effect

So let's start adding effects and let's start by adding a login effect and of course we'll thereafter also add sign-up and so on. For this, since it's auth related, in the auth store folder, let's add a new file, auth.effects.ts

So from @ngrx/effects, we have to import something and that something is actions. Now we imported action from @ngrx/store before, for creating our actions, actions here is something different, it's a different kind of object.

import { Actions, ofType } from '@ngrx/effects';

export class AuthEffects {

  constructor(private actions$: Actions) { }

}

It allows us to inject actions and the convention here is to write this with a dollar sign at the end but this is totally up to you. Now what is actions? Actions is one big observable that will give you access to all dispatched actions so that you can react to them, you just react differently than in the reducer because in the reducer, you of course also get access to all dispatched actions as you learned, here in the effect class, in the auth effects class, the idea now is that you don't change any state but that you can execute any other code that should happen when such an action is dispatched and that you then can simply dispatch a new action once that code, which also may be asynchronous, is done.

So the actions are injected here, with that injected, we can register our first effect, so our first action handler so to say. You add such an effect as a normal property in your auth effect class and I will name this auth login.

Now actions as I mentioned is just an observable and therefore we can call pipe or subscribe.

  authLogin = this.actions$.pipe(

    ofType(AuthActions.LOGIN\_START)

  );

Now don't call subscribe here, @ngrx/effects will subscribe for you, just call pipe instead and what you need to pipe here is now a special RxJS operator which is not part of RxJS but which is provided by @ngrx/effects, it's the ofType operator. ofType simply allows you to define a filter for which types of effects you want to continue in this observable pipe you're creating, in this observable stream here because you can have multiple effects by adding multiple properties to your class here and you can simply define different types of effects that you want to handle in each chain.

### Effects & Error Handling

… not too important

Tips about effects:

1. The effect have to be mark with the decorator @Effect()
2. The effect operator ofType, e.g:  
   ofType(AuthActions.LOGIN\_START)  
    It is an observable of actions and will be listening until the actions we pass to it has paramethers were ejecuted, so after that action be executed we execute our effect.
3. Normaly an effect finishing returnin an action of the reducer for example:  
   return of(new AuthActions.LoginFail(errorMessage));
4. Some effect could not return an action. In that case we mark the effect with decorator:  
   @Effect({ dispatch: false })
5. In our app.module.ts file in import section we have to import the EffectsModule module from "@ngrx/effects" and pass between brackets out effects classes. E.g.

EffectsModule.forRoot([AuthEffects])

### Finishing the Auth Effects

Obviously, this entire NgRx part was pretty extensive, was pretty much for authentication but keep in mind that authentication also is pretty complex in this application because we manage the token lifetime and so on.

Now it's basically all managed with NgRx. We were using effects for all the side effects like HTTP requests and local storage and redirection, we're using the reducer for our general state, like the authenticated user, if we have an error and if we're loading and the auth component dispatches our different login or sign-up actions and takes advantage of the error and the loading state. The auth service is used for managing our token timer then, we could move that into @ngrx/effects somehow but we would need a lot of observable magic for this and I think this is just a simpler and easy to understand approach. The auth interceptor and auth guard take advantage of our @ngrx/store and therefore now whilst it was a lot of work, now the entire authentication logic lives in NgRx. Again it was a lot of work but now whatever you add is added with relatively little effort, still of course, using this entire setup is optional.

auth.effects.ts

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

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

import { Actions, ofType, Effect } from '@ngrx/effects';

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

import { of } from 'rxjs';

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

import { environment } from '../../../environments/environment';

import \* as AuthActions from './auth.actions';

import { User } from '../user.model';

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

/\*\*

 \* Response of firebase

 \* idToken:      A Firebase Auth ID token for the newly created user or authenticated user.

 \* email:        The email for the newly created user or authenticated user.

 \* refreshToken: A Firebase Auth refresh token for the newly created user or authenticated user.

 \* expiresIn:    The number of seconds in which the ID token expires.

 \* localId:      The uid of the newly created user or authenticated user.

 \* registered:   Whether the email is for an existing account.

 \*/

export interface AuthResponseData {

  idToken: string;

  email: string;

  refreshToken: string;

  expiresIn: string;

  localId: string;

  registered?: boolean

}

const handleAuthentication = (expiresIn: number, email: string, userId: string, token: string) => {

  const expirationDate = new Date(new Date().getTime() + +expiresIn \* 1000);

  //store in browser the user info

  const user = new User(email, userId, token, expirationDate);

  localStorage.setItem('userData', JSON.stringify(user));

  //returning the action

  return new AuthActions.AuthenticateSuccess({

    email: email,

    userId: userId,

    token: token,

    expirationDate: expirationDate

  });

};

const handleError = (errorRes: any) => {

  let errorMessage = 'An unknown error occurred!';

  if (!errorRes.error || !errorRes.error.error) {

    return of(new AuthActions.AuthenticateFail(errorMessage));

  }

  switch (errorRes.error.error.message) {

    case 'EMAIL\_EXISTS':

      errorMessage = 'This email exists already';

      break;

    case 'EMAIL\_NOT\_FOUND':

      errorMessage = 'This email does not exist.';

      break;

    case 'INVALID\_PASSWORD':

      errorMessage = 'This password is not correct.';

      break;

  }

  return of(new AuthActions.AuthenticateFail(errorMessage));

};

@Injectable()

export class AuthEffects {

  constructor(

    private actions$: Actions,

    private http: HttpClient,

    private router: Router,

    private authService: AuthService

  ) { }

  @Effect()

  authSignup = this.actions$.pipe(

    ofType(AuthActions.SIGNUP\_START),

    switchMap((signupAction: AuthActions.SignupStart) => {

      return this.http.post<AuthResponseData>(

        'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=' + environment.FIREBASE\_PROJECT\_KEY,

        {

          email: signupAction.payload.email,

          password: signupAction.payload.password,

          returnSecureToken: true

        }

      )

      .pipe(

        tap( resData => {

          this.authService.setLogoutTimer(+resData.expiresIn \* 1000);

        }),

        map((resData: AuthResponseData) => {

          return handleAuthentication(+resData.expiresIn, resData.email, resData.localId, resData.idToken);

        }),

        catchError(errorRes => {

          return handleError(errorRes);

        })

      );

    })

  );

  @Effect()

  authLogin = this.actions$.pipe(

    ofType(AuthActions.LOGIN\_START),

    switchMap((authData: AuthActions.LoginStart) => {

      return this.http.post<AuthResponseData>(

        'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=' + environment.FIREBASE\_PROJECT\_KEY,

        {

          email: authData.payload.email,

          password: authData.payload.password,

          returnSecureToken: true

        }

      )

      .pipe(

        tap( resData => {

          this.authService.setLogoutTimer(+resData.expiresIn \* 1000);

        }),

        map((resData: AuthResponseData) => {

          return handleAuthentication(+resData.expiresIn, resData.email, resData.localId, resData.idToken);

        }),

        catchError(errorRes => {

          return handleError(errorRes);

        })

      );

    })

  );

  @Effect({ dispatch: false })

  authRedirect = this.actions$.pipe(

    ofType(AuthActions.AUTHENTICATE\_SUCCESS),

    tap(() => {

      this.router.navigate(['/']);

    })

  );

  @Effect()

  autoLogin = this.actions$.pipe(

    ofType(AuthActions.AUTO\_LOGIN),

    map(() => {

      const userData: {

        email: string,

        id: string,

        \_token: string,

        \_tokenExpirationDate: string

      } = JSON.parse(localStorage.getItem('userData'));

      if(!userData) {

        return {type: 'DUMMY'}

      } else {

        const loadedUser = new User(userData.email, userData.id, userData.\_token,

          new Date(userData.\_tokenExpirationDate));

        if(loadedUser.token){

          const expirationDuration =

            new Date(userData.\_tokenExpirationDate).getTime() - new Date().getTime();

          this.authService.setLogoutTimer(expirationDuration); //setting the timer

          //return the action

          return new AuthActions.AuthenticateSuccess({email: loadedUser.email, userId: loadedUser.id,

            token: loadedUser.token, expirationDate: new Date(userData.\_tokenExpirationDate)

          });

        }

        return {type: 'DUMMY'}

      }

    })

  );

  @Effect({ dispatch: false })

  authLogout = this.actions$.pipe(

    ofType(AuthActions.LOGOUT),

    tap(() => {

      this.authService.clearLogoutTimer(); //clear timer

      //delete from local storage of browser the user info

      localStorage.removeItem('userData');

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

    })

  );

}

auth.actions.ts

import { Action } from '@ngrx/store';

export const LOGIN\_START = '[Auth] LOGIN\_START';

export const AUTHENTICATE\_SUCCESS = '[Auth] AUTHENTICATE\_SUCCESS';

export const AUTHENTICATE\_FAIL = '[Auth] AUTHENTICATE\_FAIL';

export const SIGNUP\_START = '[Auth] SIGNUP\_START';

export const CLEAR\_ERROR = '[Auth] CLEAR\_ERROR';

export const AUTO\_LOGIN = '[Auth] AUTO\_LOGIN';

export const LOGOUT = '[Auth] LOGOUT';

export class AuthenticateSuccess implements Action {

  readonly type = AUTHENTICATE\_SUCCESS;

  constructor(public payload: {

    email: string,

    userId: string,

    token: string,

    expirationDate: Date

  }) {}

}

export class Logout implements Action {

  readonly type = LOGOUT;

}

export class LoginStart implements Action {

  readonly type = LOGIN\_START;

  constructor(public payload: { email: string, password: string }) {}

}

export class AuthenticateFail implements Action {

  readonly type = AUTHENTICATE\_FAIL;

  constructor(public payload: string) {}

}

export class SignupStart implements Action {

  readonly type = SIGNUP\_START;

  constructor(public payload: { email: string, password: string }) {}

}

export class ClearError implements Action {

  readonly type = CLEAR\_ERROR;

}

export class AutoLogin implements Action {

  readonly type = AUTO\_LOGIN;

}

export type AuthActions = AuthenticateSuccess | Logout | LoginStart | AuthenticateFail | SignupStart | ClearError | AutoLogin;

auth.reducer.ts

import { User } from '../user.model';

import \* as AuthActions from './auth.actions';

export interface State {

  user: User;

  authError: string;

  loading: boolean;

}

const initialState: State = {

  user: null,

  authError: null,

  loading: false

};

export function authReducer(

  state = initialState,

  action: AuthActions.AuthActions

) {

  switch (action.type) {

    case AuthActions.AUTHENTICATE\_SUCCESS:

      const user = new User(

        action.payload.email,

        action.payload.userId,

        action.payload.token,

        action.payload.expirationDate

      );

      return {

        ...state,

        authError: null,

        user: user,

        loading: false

      };

    case AuthActions.LOGOUT:

      return {

        ...state,

        user: null

      };

    case AuthActions.LOGIN\_START:

    case AuthActions.SIGNUP\_START:

      return {

        ...state,

        authError: null,

        loading: true

      };

    case AuthActions.AUTHENTICATE\_FAIL:

      return {

        ...state,

        user: null,

        authError: action.payload,

        loading: false

      };

    case AuthActions.CLEAR\_ERROR:

      return {

        ...state,

        authError: null

      };

    default:

      return state;

  }

}

auth.component.ts

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

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

import { Store } from '@ngrx/store';

import { Subscription } from 'rxjs';

import { AlertComponent } from '../shared/alert/alert.component';

import { PlaceholderDirective } from '../shared/placeholder/placeholder.directive';

import \* as fromApp from '../store/app.reducer';

import \* as AuthActions from './store/auth.actions';

@Component({

  selector: 'app-auth',

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

  styleUrls: ['./auth.component.css']

})

export class AuthComponent implements OnInit, OnDestroy {

  //this variable control if we are in login mode or signup mode

  isLoginMode = true;

  isLoading = false;

  error: string = null;

  @ViewChild(PlaceholderDirective) alertHost: PlaceholderDirective;

  private closeSub: Subscription;

  private storeSub: Subscription;

  constructor(

    private componentFactoryResolver: ComponentFactoryResolver,

    private store: Store<fromApp.AppState>

  ) { }

  ngOnInit(): void {

    this.storeSub = this.store.select('auth').subscribe(authState => {

      this.isLoading = authState.loading;

      this.error = authState.authError;

      if(this.error) {

        this.showErrorAlert(this.error);

      }

    });

  }

  onSwitchMode() {

    this.isLoginMode = !this.isLoginMode;

  }

  onSubmit(form: NgForm) {

    if (!form.valid) {

      return;

    }

    const email = form.value.email;

    const password = form.value.password;

    if (this.isLoginMode) {

      this.store.dispatch(new AuthActions.LoginStart({ email: email, password: password }));

    } else {

      this.store.dispatch(new AuthActions.SignupStart({ email: email, password: password }));

    }

    form.reset();

  }

  onHandleError() {

    this.store.dispatch(new AuthActions.ClearError());

  }

  ngOnDestroy() {

    if (this.closeSub) {

      this.closeSub.unsubscribe();

    }

    if (this.storeSub) {

      this.storeSub.unsubscribe();

    }

  }

  private showErrorAlert(message: string) {

    const alertCmpFactory =

      this.componentFactoryResolver.resolveComponentFactory(AlertComponent);

    const hostViewContainerRef = this.alertHost.viewContainerRef;

    hostViewContainerRef.clear();

    //creating the component

    const cmpRef = hostViewContainerRef.createComponent(alertCmpFactory);

    cmpRef.instance.message = message;

    this.closeSub = cmpRef.instance.close.subscribe(() => {

      this.closeSub.unsubscribe();

      hostViewContainerRef.clear();

    })

  }

}

app.module.ts

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

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

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

import { StoreModule } from "@ngrx/store";

import { EffectsModule } from "@ngrx/effects";

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

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

import { CoreModule } from './core.module';

import { HeaderComponent } from './header/header.component';

import { SharedModule } from './shared/shared.module';

import \* as fromApp from './store/app.reducer';

import { AuthEffects } from './auth/store/auth.effects';

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent

  ],

  imports: [

    BrowserModule,

    HttpClientModule,

    AppRoutingModule,

    StoreModule.forRoot(fromApp.appReducer),

    EffectsModule.forRoot([AuthEffects]),

    SharedModule,

    CoreModule,

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

Other files that also change we the auth actions and effect are these:

* auth.service.ts
* auth.guard.ts
* auth-interceptor.service.ts
* app.component.ts

However the important ones related with the effects are above.

# 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/>