Notes to

"*Angular 8 the complete guide*"



©2020 Bernie Snijders – 013 ICT

Table of Contents

[Getting started 6](#_Toc36980063)

[Setting up 6](#_Toc36980064)

[Adapting the default project 6](#_Toc36980065)

[Add the Bootstrap stylesheet 7](#_Toc36980066)

[The Basics 8](#_Toc36980067)

[Components 8](#_Toc36980068)

[selector 9](#_Toc36980069)

[templateUrl vs template 9](#_Toc36980070)

[styleUrls vs styles 10](#_Toc36980071)

[Data binding 10](#_Toc36980072)

[Directives 11](#_Toc36980073)

[Debugging 13](#_Toc36980074)

[Developer Tool 13](#_Toc36980075)

[Typescript files 13](#_Toc36980076)

[Augury Chrome Extension 13](#_Toc36980077)

[Components & Databinding Deep Dive 14](#_Toc36980078)

[Custom Property Binding 14](#_Toc36980079)

[Alias 14](#_Toc36980080)

[Custom Event Binding 14](#_Toc36980081)

[Alias 15](#_Toc36980082)

[Local reference in templates 15](#_Toc36980083)

[Getting access via ViewChild decorator 15](#_Toc36980084)

[ng-content 16](#_Toc36980085)

[Lifecycle hooks 16](#_Toc36980086)

[Getting access via ContentChild decorator 17](#_Toc36980087)

[Directives Deep Dive 18](#_Toc36980088)

[Creating a basic attribute directive 18](#_Toc36980089)

[Using the renderer to build a better attribute directive 18](#_Toc36980090)

[Using Host Listeners 19](#_Toc36980091)

[Using Host Binding 19](#_Toc36980092)

[Binding to Directive properties 19](#_Toc36980093)

[Structural Directives behind the scene 20](#_Toc36980094)

[Building a structural directive 20](#_Toc36980095)

[ngSwitch 21](#_Toc36980096)

[Using Services & Dependency Injection 22](#_Toc36980097)

[Injecting a service into a service 22](#_Toc36980098)

[Using services for cross-component communication 23](#_Toc36980099)

[Changing Pages with Routing 24](#_Toc36980100)

[Styling active router links 24](#_Toc36980101)

[Navigating Programmatically 25](#_Toc36980102)

[Using relative paths 25](#_Toc36980103)

[Route parameters 25](#_Toc36980104)

[Query parameters and fragments 26](#_Toc36980105)

[Retrieving Query parameters and Fragments 26](#_Toc36980106)

[Setting up Child (Nested) Routes 26](#_Toc36980107)

[Handling Query parameters 27](#_Toc36980108)

[Redirecting and Wilcard Routes 27](#_Toc36980109)

[Outsourcing the Routing Configuration 27](#_Toc36980110)

[Protecting Routes with Guards 28](#_Toc36980111)

[Protecting Child Routes with Guards 29](#_Toc36980112)

[Controlling navigation with canDeactivate 29](#_Toc36980113)

[Passing static data to a route 30](#_Toc36980114)

[Using Resolvers 31](#_Toc36980115)

[Location Strategies 32](#_Toc36980116)

[Understanding Observables 33](#_Toc36980117)

[Building a custom Observable 33](#_Toc36980118)

[Errors and Complete 33](#_Toc36980119)

[Operators 34](#_Toc36980120)

[Subjects 34](#_Toc36980121)

[Handling Forms in Angular Apps 35](#_Toc36980122)

[Template Driven Forms 35](#_Toc36980123)

[Validation 35](#_Toc36980124)

[Styling of controls based on the Form State 36](#_Toc36980125)

[Outputting validation error messages 36](#_Toc36980126)

[Default values using property binding 36](#_Toc36980127)

[Three different techniques using ngModel 36](#_Toc36980128)

[Grouping form controls 36](#_Toc36980129)

[Handling radio buttons 37](#_Toc36980130)

[SetValue and PatchValue 37](#_Toc36980131)

[Resetting Forms 37](#_Toc36980132)

[Reactive Forms 38](#_Toc36980133)

[Creating a form in code 38](#_Toc36980134)

[Syncing HTML and Form 38](#_Toc36980135)

[Submitting a form 39](#_Toc36980136)

[Adding Validation 39](#_Toc36980137)

[Getting access to controls 39](#_Toc36980138)

[Grouping Controls 39](#_Toc36980139)

[Arrays of Form Controls 40](#_Toc36980140)

[Creating custom validators 40](#_Toc36980141)

[Using Error Codes 41](#_Toc36980142)

[Creating Custom Async Validators 41](#_Toc36980143)

[Reacting to Status or Value changes 41](#_Toc36980144)

[Setting and Patching Values 41](#_Toc36980145)

[Using Pipes to transform the output 42](#_Toc36980146)

[Parameterize pipes 42](#_Toc36980147)

[Chaining pipes 42](#_Toc36980148)

[Build your own pipe 42](#_Toc36980149)

[Parameterize Custom Pipes 43](#_Toc36980150)

[Creating a Filter Pipe 43](#_Toc36980151)

[Pure and Impure Pipes 43](#_Toc36980152)

[The Async Pipe 44](#_Toc36980153)

[Making HTTP Requests 44](#_Toc36980154)

[Sending an HTTP request 44](#_Toc36980155)

[Using Rxjs operators to transform the response 44](#_Toc36980156)

[Using Types with the HttpClient 45](#_Toc36980157)

[Using a Service for HTTP Requests 45](#_Toc36980158)

[Handling Errors 46](#_Toc36980159)

[Using Subject to handle Errors 46](#_Toc36980160)

[Using the CatchError operator 47](#_Toc36980161)

[Creating HTTP Headers 47](#_Toc36980162)

[Using Query Params 47](#_Toc36980163)

[Observing different type of Responses 47](#_Toc36980164)

[Changing the Response Body Type 48](#_Toc36980165)

[Using Interceptors 48](#_Toc36980166)

[Running Multiple Interceptors 49](#_Toc36980167)

[Authentication and Route Protection 50](#_Toc36980168)

[Dynamic Components 51](#_Toc36980169)

[Splitting Modules 53](#_Toc36980170)

[Routing in Feature Modules 53](#_Toc36980171)

[Core Module 54](#_Toc36980172)

[Implementing Lazy Loading 55](#_Toc36980173)

[Preloading Lazy-Loaded Code 56](#_Toc36980174)

[Deployment 57](#_Toc36980175)

[JIT vs AOT 57](#_Toc36980176)

[Environment Variables 57](#_Toc36980177)

[Unit Testing 58](#_Toc36980178)

[Test dependencies and Services 58](#_Toc36980179)

[Simulating Asynchronous tasks 59](#_Toc36980180)

[Using fakeAsync and Tick 59](#_Toc36980181)

[Isolated vs non-Isolated testing 60](#_Toc36980182)

# Getting started

## Setting up

1. Install NodeJS (server side javascript) – this contains the node package manager (npm)
2. Install the latest Angular command line interface (cli):

$ npm install -g @angular/cli@latest

1. Create a new project:

$ ng new my-project

1. Goto the root folder of your new project:

cd my-project

1. Start an Angular Live Development Server which host your project:

ng serve

1. Start Google Chrome (this browser has the best development tools).
2. Type in the address bar: localhost:4200
3. You will see the default starting screen.

## Adapting the default project

In the folder my-project/src/app there is a file called app.component.html. This file contains the html code of your component. To insert a text input field, do the following:

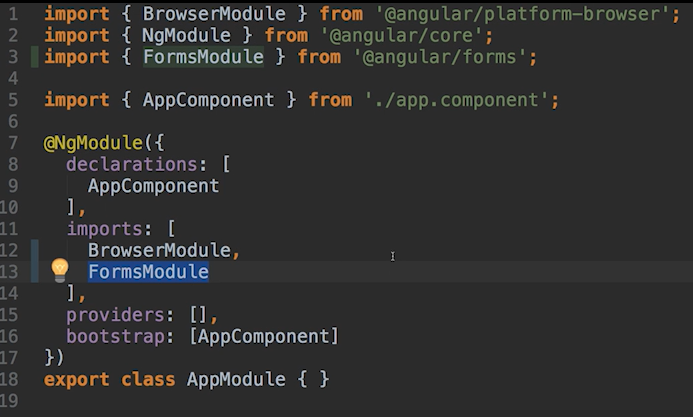
1. Insert in the file app.component.html the following code:

*ngModel is a directive which, in this case, binds the input to a variable and vice versa (two-way-binding).*

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

<p>{{ name }}</p>

1. Add a dependency to ngModel:
2. Open the app.module.ts and add the lines 3 and 13



*The imports on the top are for Typescript. The imports in the @NgModel annotation are for Angular.*

Figure 1: app.module.ts file

## Add the Bootstrap stylesheet

To make sure that we do not have to bother about the look and feel of our project, a cascading stylesheet (css) can be installed that will format the elements in our project.

1. Install the Bootstrap stylesheet:

npm install --save bootstrap@3

1. Open the file angular.json and add line 26.

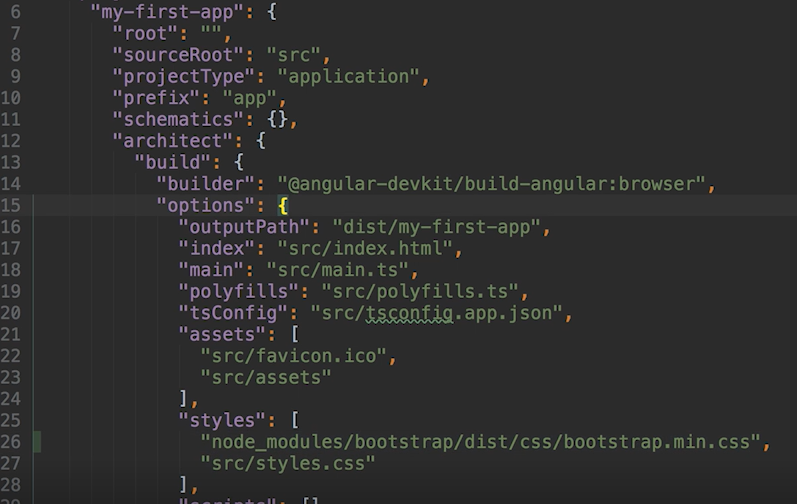


Figure 2: angular.json file

# The Basics

How is Angular know what to load? The key to this, is the main.ts file.

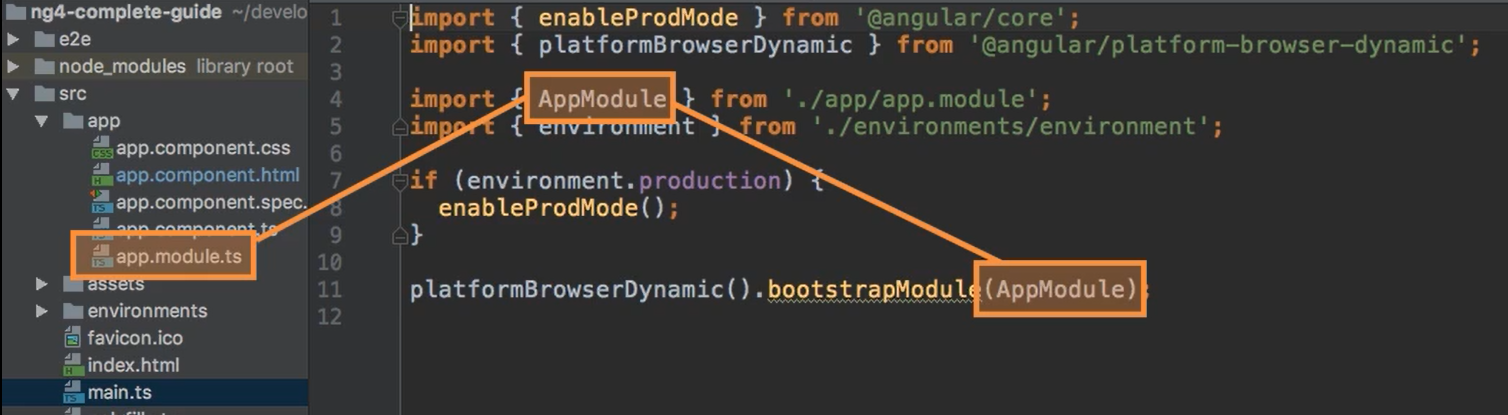


Figure 3: main.ts file

In the main.ts file is specified which Module has to be loaded at startup. In the example above this is the AppModule. The import specifies where to find the Typescript file of this module 🡪 ./app/app.module.ts.

In the file ./app/app.module.ts is specified how the AppModule looks like. In the bootstrap element is specified which Component has to be loaded at startup (see line 16 in the figure below). The import specifies where to find the Typescript file of this component 🡪 ./app/app.component.ts.

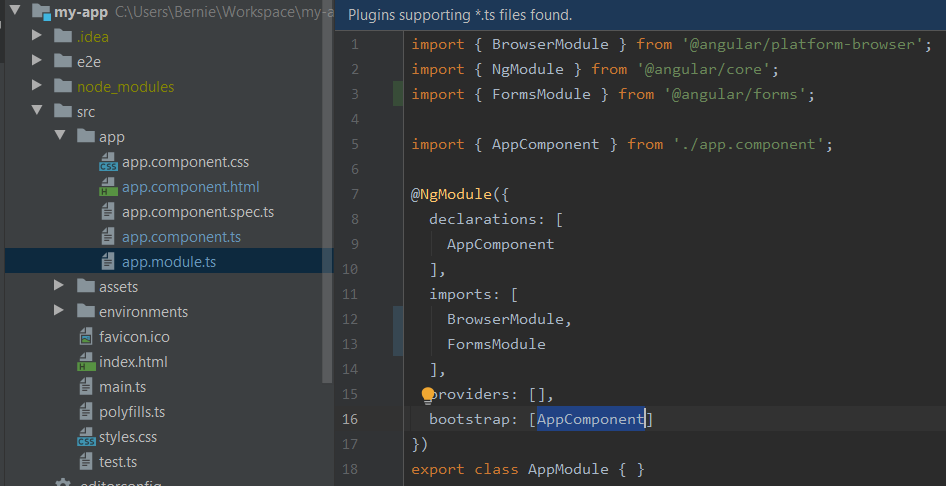


Figure 4: ./app/app.module.ts file

## Components

Adding a component can be done by performing:

ng generate component servers or   
ng g c servers

This will create a folder inside your app with the name you specified. Inside this component folder, several files will be created:

servers.component.html 🡪 The html template.

servers.component.css 🡪 The cascading style sheet.

servers.component.ts 🡪 The definition of the component.

servers.component.spec.ts 🡪 This file is used for testing.

In the app.component.ts file, a declaration is added for this new component.

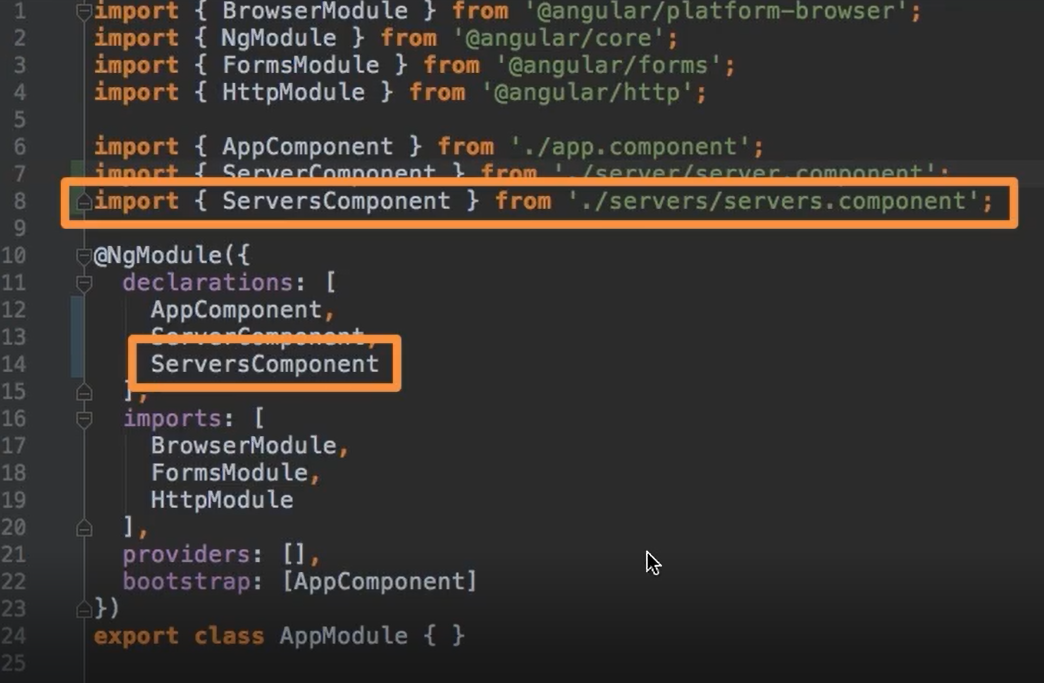


Figure 5: ./app/app.component.ts file

The generated servers.component.ts file looks like below:

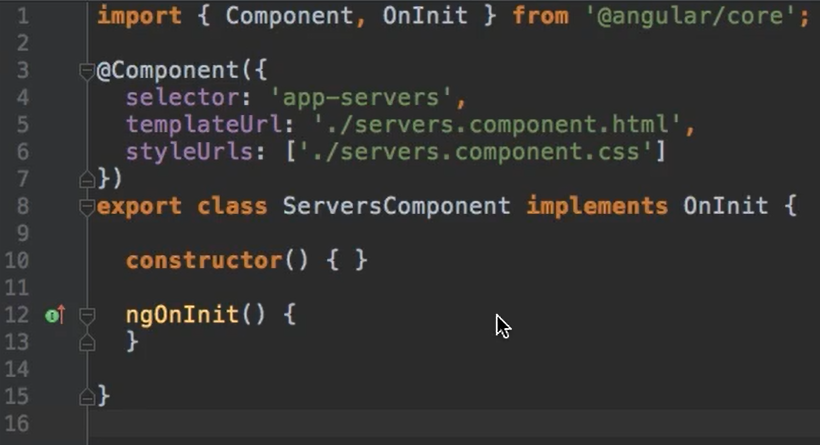


Figure 6: ./app/servers/servers.component.ts file

### selector

With the selector you specify how the template can be used.

selector: ‘app-servers’ 🡪 template can be used as an element;

e.g.<app-servers><app-servers>

selector: [app-servers] 🡪 template can be used as an attribute;

<div app-servers></div>

selector: ‘.app-servers’🡪 template can be used as a class;

<div class="app-servers"></div>

### templateUrl vs template

The templateUrl specifies where to find the HTML template file. Instead of defining an external template, the template can also be included in the servers.component.ts file. Replace templateUrl with template. One of these keywords needs to be present in your component.ts file. Keep in mind that you have to use backticks (` and ‘) to enable multilines in the template definition.



Figure 7: ./app/servers/servers.component.ts file

The preferred way to provide a template, is using an external file.

### styleUrls vs styles

Styles can be defined in the same way as the template. Both with an external file or inline in the component.ts file. Use the keyword styleUrls to provide the name of the external file. Or use the keyword styles to define it in the component.ts file. One difference with using the styles is that you can define multiple. Therefor the definition is done in a string array.

The preferred way to provide styles, is using an external file.

### Data binding

Data binding is the communication between the business logic (the Typescript files) and the template (the HTML files). The data typed in the template, will be connected with the variables in the business logic. There are different ways of data binding:

1. String interpolation
2. Property binding
3. Event binding
4. Two-way binding

#### String interpolation

There are different ways of string interpolation:

{{ variable-name }}

{{ 'string' }}

{{ methodname() }}

The only restriction for string interpolation is that the expression between the double accolades will result in a string or can be automatically converted to a string (e.g. a number or boolean can be automatically converted to a string).

#### Property binding

With property binding you bind a native (HTML) element property in your template with a Typescript property. This can be done in the following way:

<button class="btn btn-primary"   
[disabled]= "!isUserAllowedToPressButton">My Button</button>

In the above example, the button is disabled when the Typescript property isUserAllowedToPressButton is false. Because of the exclamation mark, the value is negated and will result in true. This will disable the button. The marked text is Typescript code and NOT HTML code. Therefor you have to use Typescript code in the double quotation marks. Angular will interpret this line and replace it in the Document Object Model (DOM/HTML) with disabled=true or disabled=false.

#### Event binding

With event binding you bind a native (HTML) event to a Typescript method. This can be done in the following way:

<button class="btn btn-primary"   
(click)= "onLogin()">Login</button>

You can basically bind to any HTML event. Remove the 'on' from the eventname and place the event name in parentheses. Again, the marked text is NOT HTML code but is Typescript code that is interpreted by Angular. The marked text will be replaced by Angular in de the DOM with the corresponding HTML code.

#### Two-way binding

With two-way binding you combine the property and event binding. When an event is triggered, the value of a property is changed. Suppose a user types in a firstname, the firstname property will be updated with the value the user typed. On the other hand, when the firstname property is changed, the input field is also changed. The easiest syntax to use two-way binding is by making use of the ngModel directive:

<input type="text" class="form-control"   
[(ngModel)]= "firstname">

To use ngModel in your template, make sure that FormsModule from @angular/core is imported in your app.module.ts file.

### Directives

#### ngIf

Directives are instructions inside the Document Object Model (DOM). There are custom and built-in directives. An example of a built-in directive is ngIf. This is also a called structural directive because it changes the structure of the DOM. It can be used to add or leave out HTML elements. Structural directives must be preceded with an '\*'.

<p \*ngIf="isShowMessage()">{{ getMsg() }}</p>

The code above will only output the paragraph in the DOM when the expression after \*ngIf is evaluated to true. In this case when the Typescript method isShowMessage() return true. You can use ngIf also with an else statement:

<p \*ngIf="isShowMessage(); else noMessage">{{ getMsg() }}</p>

<ng-template #noMessage><p>No message</p></ng-template>

The notation of #noMessage is called a local reference. And can be used throughout the template where it is defined.

An alternative of the above code is to just add 2 \*ngIf statements like this:

<p \*ngIf="isShowMessage()">{{ getMsg() }}</p>

<p \*ngIf="!isShowMessage()">No message</p>

#### ngStyle

Besides structural directives, there are also attribute directives. These will only change the look of an HTML element. An example is ngStyle.

<p [ngStyle]="{backgroundColor: getColor()}">Text</p>

The above paragraph is dynamically styled with the ngStyle directive. It will set the background color depending on the return value of the method getColor();

You can also use the ngStyle directive with a conditional clause.

<p [ngStyle]="{backgroundColor: number % 2 !==0 ? 'yellow' : 'green'}">Text</p>

This will style the background of the paragraph yellow for odd numbers and green for the even numbers.

#### ngClass

With ngClass it is possible to dynamically add or remove CSS classes to a HTML element. Below you apply the online style to a paragraph when the server status property is 'online'.

<p [ngClass]="{online: serverStatus === 'online'}">Text in paragraph</p>

#### ngFor

With ngFor it is possible to dynamically loop over the values of a certain property and create elements in the HTML for each value of the property.

<app-server \*ngFor="let server of servers"></app-server>

The above example outputs for each value in the property servers, an app-server component.

**Note**: use a '\*' before the ngFor directive because it is a structural directive.

If you want to have access to the index of the for loop, you can do so by adding a variable and give it the value of index.

<app-server \*ngFor="let server of servers;   
let i = index">Server number {{ i }}.</app-server>

# Debugging

## Developer Tool

The first thing to do when your application is not responding the way you want, is to open the Developer Tool window in the Google Chrome browser. Just press F12 to open it. Have a look at the Console if there are any errors. Line numbers often do not tell the exact problem because the Typescript code is compiled into JavaScript.

## Typescript files

When you have opened the Developer Tool window, click on the Sources tab. Remember that the browser runs Javascript files (which can be found in the localhost:4200 folder). But it can be hard to find the line of code you want to debug. It would be much easier when you can put a breakpoint on your Typescript code. To view this, goto webpack://. Here you can find the Typescripts files organized in the package structure you’ve created.

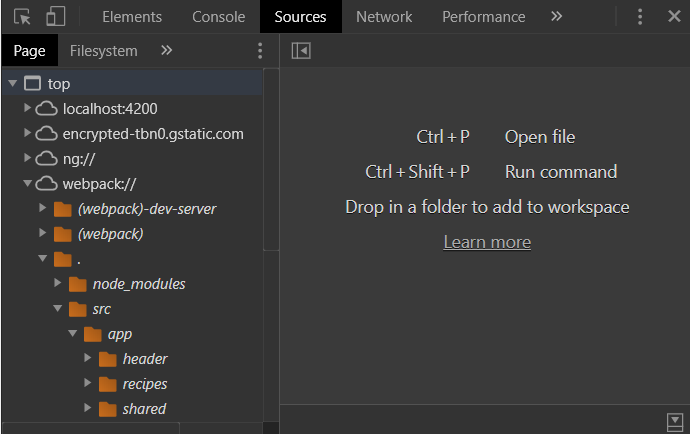


Figure 8: Developer Tool window for debugging Typescript code

## Augury Chrome Extension

There are some Chrome extensions available which helps you even more with debugging your application. For example Augury. Goto to the webpage: <https://augury.rangle.io/> and install the Chrome extension. In the Developer Tools window you can now click on *>>* and select *Augury*. This will open up a window where you get more information about your app components and variables.

# Components & Databinding Deep Dive

## Custom Property Binding

If you want to have access to a property that belongs to another component, you can put the decorator @Input() before the property:

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

@Component({

selector: 'app-server-element',

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

styleURls : ['./server-element.component.css']

})

export class ServerElementComponent implements OnInit {

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

}

Now you can bind to this property:

<app-server-element

\*ngFor="let serverElement of serverElements"

[element]="serverElement"></app-server-element>

### Alias

If you do not want the use the name of the property outside the component, you can provide Input with an alias parameter:

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

You can now bind to the component property element with the name srvElement:

<app-server-element

\*ngFor="let serverElement of serverElements"

[srvElement]="serverElement"></app-server-element>

## Custom Event Binding

If you want to inform a component that something has happened within your component, you can use event binding. You can define your own Events and bind to this event.

cockpit.compontent.html

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

Add Server</button>

cockpit.component.ts

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

@Component({..})

When the user pushes the button AddServer in the Cockpit component, event serverCreated is emitted. This event is caught in the app.component.html file and the method onServerAdded is called with the event data.

export class CockpitComponent implements OnInit {

serverName: string;

serverContent: string;

@Output() serverCreated =

new EventEmitter<{name:string, content:string}>();

onAddServer () {

this.serverCreated.emit({

name: this.serverName,

content: this.serverContent})

}

app.component.html

<app-cockpit (serverCreated) = "onAddServer($event)">

</app-cockpit>

app.component.ts

The method onServerAdded is called and gets the event data passed as arguments. This data can then be used to insert into the serverElements array.

export class AppComponent implements OnInit {

serverElements {

type: string,

name: string,

content: string}: []

onServerAdded(serverData: {name: string, content: string}) {

this.serverElements.push(

type: 'server',

name: serverData.name,

content: serverData.content)

}

### Alias

If you do not want the use the name of your custom event outside the component, you can provide Output with an alias parameter:

@Output('srvCreated') serverCreated =

new EventEmitter<{name:string, content:string}>();

You can now bind to the component event with the name srvCreated:

<app-cockpit (srvCreated)="onAddServer($event)"></app-cockpit>

## Local reference in templates

Instead of two-way databinding, you can also use local references. The local reference can be used, as the name already says, only local in your template. It works like this:

cockpit.component.html

<input "text" class="form-control" #name>

<input "text" class="form-control" #content>

<button class="btn btn-primary"

(click)="onAddServer(name, content)">

cockpit.component.ts

onAddServer(name: HTMLInputElement, content: HTMLInputElement){

this.serverCreated.emit({

name: name.value,

content: content.value})

}

## Getting access via ViewChild decorator

With a local reference you get access, within your template, to an HTML element. With the ViewChild decorator you can get access to an HTML element within your Typescript code. This is how to use the ViewChild decorator.

import { .., .., ViewChild, ElementRef } from '@angular/core';

@ViewChild('nameInput', {static: true}) nameInput: ElementRef;

## ng-content

With ng-content you can output HTML elements in your template between your component selectors. Normally anything between the selectors will be ignored. For instance:

app.compontent.html

<server-element>Loading the application…</server-element>

will only include the app-component Component in your DOM. The text between the selector, in this case "Loading the application…", will be omitted. This is the default behavior of Angular. If you want that the text (or any other HTML elements) to be shown, you can use ng-content.

server-element.component.html

<ng-content></ng-content>

app.component.html

<server-element>Loading the application…</server-element>

Now the text "Loading the application…" is shown in the DOM. So you have to place the ng-content directive in the component where you want to include the HTML elements that are put between the component selectors (in this case server-element).

## Lifecycle hooks

The Angular framework has a couple of hooks during the lifecycle of each component. These hooks can be used to perform some logic. Below there is a list of the hooks in order they occur. All hooks can be imported via '@angular/core'.

Table 1 Life cycle hooks

|  |  |  |
| --- | --- | --- |
| Event | Hook / Interface method | Description |
| ngOnChanges | onChanges(changes: SimpleChanges) | After a bound input property changes. |
| ngOnInit | onInit() | Once the component is initialized. |
| ngDoCheck | onDoCheck() | During every change detection run. |
| ngAfterContentInit | onAfterContentInit() | After content (ng-content) has been projected in the view. |
| ngAfterContentChecked | onAfterContentChecked() | Every time the projected content has been checked. |
| ngAfterViewInit | onAfterViewInit() | After the component's view (and child views) has been initialized. |
| ngAfterViewChecked | onAfterViewChecked() | Every time the component's view (and child views) has been checked. |
| ngOnDestroy | onDestroy() | Once the component is about to be destroyed. |

The hook onChanges is the only hook with a parameter. This parameter is of type SimpleChanges and holds both the current value and the previous value.

**NOTE:** When accessing your HTML elements in your lifecycle hook, be aware that the elements of your component are not yet rendered, and therefore do not have a value. Only after the onAfterViewInit hook, these values are available.

## Getting access via ContentChild decorator

To access the content (not the value!!) of ng-content in your component, you can use the ContentChild decorator.

app-component-element.html

<app-server-element>

<p #content>{{ serverElement.content }}</p>

</app-server-element>

server-compontent.ts

import { .., .., ContentChild, ElementRef } from '@angular/core';

@ContentChild(content, {static: true}) paragraph: ElementRef;

ngAfterContentInit() {

console.log(

"Content: " + this.paragraph.NativeElement.textContent());

}

# Directives Deep Dive

## Creating a basic attribute directive

Directives are placed in a separate file/class. Below you will find an example:

By putting private in the argument list of the constructor, it will create a class member elRef and assign the value of the argument to it.

**NOTE:** It is not a good practice to directly access your DOM elements like this.

basic-highlight.directive.ts

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

@Directive({

selector: [appBasicHighlight]

})

export class BasicHighlightDirective implements OnInit {

constructor(private elRef: ElementRef) { }

onInit() {

this.elRef.nativeElement.style.background='green'

}

}

Add the new directive to the declarations part in the app.module.ts file. Or make sure you create the directive via the CLI: ng generate directive basic-highlight. This will add automatically the import and declarations part below.

import { BasicHighlightDirective } from './basic-highlight/basic-highlight.directive';

@NgModule({

declarations: [

AppCompents,

BasicHighlightDirective],

imports: [

BrowserModule

FormsModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule{ }

Now you can use this attribute directive in your template like this:

<p appBasicHighlight>This paragraph has a green background></p>

## Using the renderer to build a better attribute directive

In the previous paragraph we read that it is not a good practice to directly access your DOM elements. The reasons for this is that Angular can not only be run in a browser. This means that it doesn't have access to the DOM. Although most of the times your app is running in a browser it is still a better approach to use the renderer for this purpose.

export class BetterHighlightDirective implements OnInit {

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

onInit() {

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

}

}

## Using Host Listeners

In the previous paragraph we read about how to create a static attribute directive. To create a dynamic attribute directive, we can make use of a HostListener.

import { .., Renderer2, HostListener } from '@angular/core';

@Directive({

selector: '[appDynamicHighLight]'

})

export class DynamicHighlightDirective {

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

@HostListener('mouseover') mouseover(eventData: event) {

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

}

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

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

}

}

## Using Host Binding

When the only thing we want to do in our attribute directive is setting the background color, you can also use the HostBinding directive. This is a much easier way to use.

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

@Directive({

selector: '[appDynamicHighLight]'

})

export class DynamicHighlightDirective {

constructor() {}

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

@HostListener('mouseover') mouseover(eventData: event) {

this.color = 'blue';

}

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

this.color = 'transparant';

}

}

## Binding to Directive properties

In the previous paragraph we have hardcoded the color values when hovering with the mouse over the HTML element. Suppose we want to make this configurable from outside the component. Then we could use property binding.

import { Directive, HostListener, Input, OnInit } from '@angular/core';

@Directive({

selector: '[appDynamicHighLight]'

})

export class DynamicHighlightDirective implements OnInit{

constructor() {}

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

@Input() highlichtColor: string = 'yellow';

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

onInit() {

this.defaultColor = this.defautColor;

}

@HostListener('mouseover') mouseover(eventData: event) {

this.color = this.highlightColor;

}

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

this.color = this.defaultColor;

}

}

In the template you can now set the colors of the appBetterHighLight directive.

<p appDynamicHighLight

[defaultColor]="'yellow'" [highlightColor]="'red'"> Style me</p>

When you use property binding with a string, you can also write it like this (without square brackets and single quotes):

<p appBetterHighlight

defaultColor="yellow" highlightColor="red"> Style me</p>

## Structural Directives behind the scene

The \*ngIf structural directive can be used to conditionally output an HTML element into your DOM.

<div \*ngIf="isShowMessage()">

<p>This is the message</p>

</div>

Behind the scenes, Angular is transforming this statements in the following structure:

<ng-template [ngIf]="isShowMessage()">

<div>

<p>This is the message</p>

</div>

</ng-template>

## Building a structural directive

You can build your own structural directive. Below you will find the opposite directive of \*ngIf. It will show the element unless the condition is met. Important to mention is that the name of the property to set, must be equal to the name of the selector.

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

@Directive({selector: '[appUnless]'})

export class UnlessDirective {

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

if (!condition) {

this.vcRef.createEmbeddedView(this.templateRef);

} else {

this.vcRef.clear();

}

}

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

}

In your template, use the structural directive as follows:

<p \*appUnless="notShowMessage()">This is the message</p>

**NOTE**: use the '\*' because it is a structural directive.

## ngSwitch

There is a handy structural directive in case you have to check on multiple values and depending on the value you want to show a corresponding HTML element. It is called ngSwitch.

Below you will see a simple example of it:

<div [ngSwitch]="value">

<p \*ngSwitchCase="5">The value is 5</p>

<p \*ngSwitchCase="10">The value is 10</p>

<p \*ngSwitchCase="100">The value is 100</p>

<p \*ngSwitchDefault="5">The value is default</p>

</div>

**NOTE**: value is a property of the component.

# Using Services & Dependency Injection

Services can be used to provide a certain function to the application. For instance, logging. To create a service, you just create a class.

logging.service.ts

export class LoggingService {

statusChanged(status: string) {

console.log('The status is changed: " ' + status);

}

}

You can use this service in any component by making use of dependency injection. This will let Angular instantiate the LoggingService and inject it into your component. Below you will see how to implement this.

account.component.ts

import { LoggingService } from '../services/LoggingService';

Component({

selector: 'app-account'

providers: [LoggingService]

})

export class AccountComponent {

constructor(private logService: LoggingService) {}

}

onSetTo(status: string) {

logService.statusChanged(status)

}

First, add an import statement to the component. Then add a provider in the Component decorator and create a constructor with as argument the service to inject. Now you can use the service throughout your component by simply calling the method on the logService class attribute. If child components need the same service, you do not have to mention it in the providers parameter of the Component decorator. Keep in mind that the child components get a reference to the service. So each child component uses the same service. If your child component need a new reference to the service, you have to add it to the providers.

import { LoggingService, AccountService }

@Compontent({

selector: 'app-account'

providers: [ LoggingService ]

})

export class NewAccountComponent {

constructor(private logService: LoggingService,

private accountService: AccountService) {}

onCreateAccount(name: string, status: string) {

this.accountService.addAccount(name, status);

this.logService.statusChanged(status);

}

}

## Injecting a service into a service

Suppose you want to log inside the AccountService. Therefor you have to inject the LoggingService into the AccountService. To achieve this, there are a couple of things to do. First thing: add the Service to the providers section of the app.module.ts file. Then add a constructor with a private parameter of type LoggingService to the AccountService. Finally, add the @Injectable decorator to the AccountService. Now you can use the class attribute logService.

app.module.ts

@NgModule({

declarations: [AppCompents, AccountComponent, ..],

imports: [BrowserModule, FormsModule],

providers: [AccountService, LoggingService],

bootstrap: [AppComponent]

})

export class AppModule{ }

account.service.ts

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

@Injectable

export class AccountService {

accounts: {name: string, status: string}[] = [];

statusUpdated = new EventEmitter<string>;

constructor(private logService: LoggingService) {}

addAccount(name: string, status: string) {

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

this.logService.statusChanged(status);

}

}

## Using services for cross-component communication

When you define an event emitter in your service (see above statusUpdated), other components can emit and subscribe to this event. This is a very quick way of passing information between components.

account.component.ts

import { LoggingService } from '../services/LoggingService';

Component({

selector: 'app-account'

providers: [LoggingService]

})

export class AccountComponent {

constructor(private logService: LoggingService,

private accountService: AccountService) {}

}

onSetTo(status: string) {

this.accountService.updateStatus(status)

this.accountService.emit(status);

}

new-account.component.ts

export class NewAccountComponent {

constructor(private accountService: AccountService) {

this.accountService.statusChanged.subscribe(

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

}

}

# Changing Pages with Routing

With routing our Angular application is still a single page application but it looks like we created multiple pages because the URL changes when selecting different components. How do we set up a route? Look at the example below.

app.module.ts

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

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

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

{ path: 'servers', component: ServersComponent }

]

@ngModel({

declarations: [HomeComponent, UsersComponent, ServersComponent]

imports: [RouterModule.forRoot(appRoutes)]

})

This is the configuration of our routes. We now have to define where to render the components. You can simply do this by using the router-outlet directive.

<div class="row"><router-outlet><router-outlet></div>

Angular will now replace the router-outlet directive with the component defined in the router definition. To navigate to a route, you can use the following code in your template:

<ul class="nav nav-tabs">

The first routerlink is the homepage of the application.

The third routerlink consists of a multi-level route: localhost:4200/servers/offline

<li role="presentation" class="active">

<a routerlink="/">Home</a>

</li>

<li role="presentation">

<a routerlink="/servers">Servers</a>

</li>

<li role="presentation">

<a [routerlink]="['/servers', 'offline']">

Offline servers</a>

</li>

</ul>

## Styling active router links

When using router links in your application, you can style the active router link to indicate to the user which is the active one. To do this you can use the routerlinkActive directive.

The style you set to the routerlinkActive directive, will only be applied when the routerlink is active. Add the routerlinkActive Options to ensure that the homepage is always styled (because it is always in the path).

<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>

</ul>

## Navigating Programmatically

In the previous sections we saw that we can navigate, using the routerlink directive. It is also possible to navigate writing Typescript code.

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

export class HomeComponent {

constructor(private router: Router) {}

onLoadServers() {

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

}

## Using relative paths

When you want to navigate to a route within a component, you can navigate using a relative path. Inject the current route with the ActivatedRoute class. Pass the value of this ActivatedRoute to the navigate method of the Router. See the example below:

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

export class HomeComponent {

constructor(

private router: Router,

private activeRoute: ActivatedRoute) {}

onLoadServers() {

this.router.navigate(['offline'], {relativeTo: this.activeRoute});

}

## Route parameters

Within in a route you can include dynamic parameters. For instance, a user-id.

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users:id/:name', component: UsersComponent } ]

To fetch this user-id in your component, use the following piece of code:

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

constructor(private activeRoute: ActivatedRoute) {}

ngOnInit() {

user = {

this.user.id = this.activeRoute.snapshot.param['id'],

this.user.name = this.activeRoute.snapshot.param['name']

};

}

When there is a way that the route is changing within your component, you have to make sure that the user object is reflecting this. Subscribe on the change of the active route parameters. Add the following code to the ngOnInit() method.

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

this.activeRoute.params.subscribe(

(params: Params) => {

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

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

}

)

Route observables are destroyed by Angular as soon as the component is destroyed. This is **NOT** the case when you create your own Observable. Then you have to unsubscribe to the Observable in the onDestroy method.

## Query parameters and fragments

A route can contain query parameters and/or fragments. Think of the following route URL: localhost:4200/users/2/edit?name='John'#loading.

This route can be defined in the app.model.ts:

const appRoutes : Routes = [

{path: '/users/:id/edit', component: 'EditUserComponent'}

]

In your template you can navigate to this route.

<a

[routerLink]="['users', user.id, 'edit']"

[queryParams]="{name: 'John'}"

[fragment]="'loading'"

href="#" class="list-group-item"

\*ngFor="let user of users">

{{ user.name }}

</a>

Besides navigating to a route in your template, it is also possible to do this programmatically.

onEditUser(number: id) {

this.router.navigate(['/users', this.user.id, 'edit'],

{queryParams: {name: this.user.name},

fragment: 'loading'});

}

## Retrieving Query parameters and Fragments

In the previous section we learned how to navigate to a route. This section is about how to retrieve the query parameters and/or the fragments in your Typescript file.

ngOnInit() {

// this can be used when the url is not changing in your component

this.activeRoute.snapshot.queryParams;

this.activeRoute.snapshot.fragment;

// use this when the url can change in your component

this.activeRoute.queryParams.subscribe();

this.activeRoute.fragement.subscribe();

}

## Setting up Child (Nested) Routes

When you have multiple routes with the same root, you can group them. For instance, the routes: users and users/id can be grouped because they share the same route: users.

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

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

[{path: ':id', component: UserComponent},

{path: ':id/edit', component: EditUserComponent]

}

]

In the template of your UsersComponent, you can now place a router-oulet directive. Angular will then render the corresponding component at the place of the directive, depending on the active route.

users.component.html

<router-outlet></route-outlet>

## Handling Query parameters

Suppose you pass the query parameter user.id from the UsersComponent to the UserComponent. The UserComponent contains a button to Edit the user. You want to pass the user.id to the EditUserComponent so this component can update the user.

When you want to preserve the query parameters of the active route. You can set the queryParamsHandling property of the router.navigate() method. Choose 'preserve' when you want to keep the query parameters provided. If you want to add additional query parameters, choose 'merge'.

onEdit() {

this.router.navigate(

['edit'],

{ activeRoute.relativeTo: this.activeRoute,

queryParamsHandling: 'preserve'}

}

## Redirecting and Wilcard Routes

Suppose a user types a URL that doesn't exist. You may want to redirect the user to a 'Not Found' page. Below you see an example of redirecting.

The '\*\*' means all routes that are not recognized. Make sure that this route is always at the bottom of the routes definition. All routes that are defined after it, will be redirected.

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

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

[{path: ':id', component: UserComponent},

{path: ':id/edit', component: EditUserComponent] },

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

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

]

## Outsourcing the Routing Configuration

It is a good practice to store all routing configuration in a separate Module. Create a file in the root of your application and give it the content as shown below.

*app-routing.module.ts*

const appRoutes: Routes = [

// your routes configuration here !!!

]

@ngModule({

imports: [RouterModule.forRoot(appRoutes)],

exports: [RouterModule]

})

export class AppRoutingModule {

}

app.model.ts

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

@ngModel({

imports: [AppRoutingModule]

providers: [UserService]

bootstrap: [AppComponent]

})

## Protecting Routes with Guards

Routes can be protected by being accessed by the user. These are so called Guards. Add the guard to the route configuration.

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', canActivate: [AuthGuard], component: UsersComponent, childeren:

[{path: ':id', component: UserComponent},

{path: ':id/edit', component: EditUserComponent] }

The users route and all his children are now protected by the AuthGuard. The AuthGuard can be implemented as a service.

auth-guard.service.ts

@Injectable()

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService,

private router: Router) {}

canActivate(

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot) :

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

this.authService.isUserAuthenticated()

.then(

(authenticated: boolean) => {

if (autenticated) {

return true;

} else {

router.navigate(['/']);

}

}

)

}

}

The AuthService is here returning a Promise of type boolean. Because the authentication of a user is an asynchronous task, it can take a while, the boolean is wrapped in a Promise.

auth.service.ts

isUserAuthenticated() {

const promise = new Promise(

(resolve, reject) => {

setTimeout(()=> {resolve(this.loggedIn);

}, 800);

}

);

return promise;

}

Don’t forget to add the CanActivateGuard and the other services to the app.module.ts file.

@ngModel({

declarations:

import:

providers: [.., AuthService, AuthGuard, CanActivateGuard]

})

## Protecting Child Routes with Guards

In the previous section we saw how to protect a route with a Guard. It is also possible to protect only the children of a route.

export class AuthGuard implements CanActivate, CanActivateChild {

canActivateChild(

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot) :

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

return this.canActive(route, state);

}

In your routing configuration you have to change the following:

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', canActivateChild: [AuthGuard], component: UsersComponent, childeren:

[{path: ':id', component: UserComponent},

{path: ':id/edit', component: EditUserComponent] }

## Controlling navigation with canDeactivate

With the CanDeactivate you can control whether the user is allowed to leave a route. For instance, preventing users by accidentally navigating away from a component, without saving data. You can do this with a Guard. In your route configuration you have to define this in the way shown below:

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

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

[{path: ':id', component: UserComponent},

{path: ':id/edit', canDeActivate: [DeactivateGuard], component: EditUserComponent] }

The Guard can be implemented like this:

can-deactivate-guard.service.ts

import { Observable } from 'rxjs/Observable';

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

The CanDeactivateGuard calls the canDeactive method on the component which the Guard is placed. This component must therefore implement the interface CanComponent Deactivate.

export interface CanComponentDeactive {

canDeactivate: () =>

Observable<boolean> | Promise<boolean> | boolean;

}

export class CanDeactivateGuard implements

CanDeactive<CanComponentDeactivate> {

canDeActivate(

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot,

nextState: RouterStateSnapshot) :

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

return component.canDeactivate();

}

We have the /users/:id/edit route configured and the Guard implemented. We now have to implement the method canDeactivate in the Typescript file of the EditUserComponent.

export class EditUserComponent implements CanDeactivateComponent {

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

if (!this.allowEdit) {

return true;

}

if (isUserDataChanged()) {

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

} else {

return true;

}

}

The method isUserDataChanged() in the above example is a convenient method to determine whether the values of the User have been changed. Don’t forget to add the CanDeactivateGuard to the app.module.ts file.

@ngModel({

declarations:

import:

providers: [.., CanDeactivateGuard]

})

## Passing static data to a route

There is a way to pass data to a component in your route. Suppose you have a ErrorPageComponent which prints a message. You may want to pass a specific message to this component via the route configuration.

error-page.component.html

<h4>{{ errorMessage }}</h4>

error-page.component.ts

@Injectable()

export class ErrorPageComponent implements OnInit {

errorMessage: string;

constructor(private route: ActivateRoute {}

ngOnInit() {

// use this when message doesn't change on this component

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

// use this when message can change on this component

this.route.data.subscribe(

(data: Data) => {

this.errorMessage = data['message'];

}

}

}

## Using Resolvers

With a resolver you can run some code before a route is rendered. It can be used to fetch some data from a backend server. Just as Guards, Resolvers are implemented as a service. The difference between the CanActivate and CanDeactivate is that Resolvers do not decide whether the route should be rendered, they will run just before the route is rendered.

server-resolver.service.ts

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

import { Observable } from 'rxjs/Observable';

interface Server {

id: number;

name: string;

status: string;

}

@Injectable()

export class ServerResolver implements Resolve<Server> {

constructor(private serverService: ServerService) {}

resolve(

Route.params['id'] will return a string, while the method getServer expects a number. You can cast a string to a number by putting a '+' before the statement.

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot) :

Observable<Server> |

Promise<Server> |

Server {

this.serverService.getServer(

+route.params['id']);

}

}

The getServer will return a Server object. But the ServerService could also return an Observable<Server> or a Promise<Server>. The code of this resolver will not change. Now the Observer is created, you have to add it to the route configuration:

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

[

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

resolve: {server: ServerResolver}}

]

Don’t forget to add the Resolver to the app.module.ts file.

@ngModel({

declarations:

import:

providers: [.., ServerResolver]

})

In the ServerComponent you can now use the server object which the ServerResolver has created.

ngOnInit() {

this.route.data.subscribe(

{(data: Data) => this.server = data['server']})

}

## Location Strategies

When using routes, your application server or browser could not handle the routes. The application server could look for a file named users.html for route host:4200/users. When this is the case, you could use the hashtag in the routes. Add {useHash: true}, to the forRoot method of RouterModule.

imports: [RouterModule.forRoot(appRoutes, {useHash: true})],

# Understanding Observables

When your application uses asynchronous communication, Observables can be used. Several data sources in Angular have Observables implemented. You can subscribe to these Observables. With every change of the data source, the subscriber gets notified.

this.activeRoute.params.subscribe((params: Params) => {

this.id = +params.id;

})

The built-in Observables are automatically terminated. When you create your own Observable, you have to clean it yourself.

private subscription: Subscription;

ngOnInit() {

this.subscription = interval(1000).subscribe(count=> {

console.log(count);

});

}

ngOnDestroy() {

this.subsciption.unsubscribe();

}

## Building a custom Observable

Call observer.next() to emit a new value and let the subscriber know a new value is available.

Call observer.error() to pass an error to the subscriber.

Call the method observer.complete() to let the subscriber know that the Observable is done.

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

private subscription: Subscription

ngOnInit() {

const customInterval = Observable.create(observer => {

let count = 0;

setInterval() => {

observer.next(count);

count++;

}, 1000);

});

this.subscription = customInterval.subscribe(data => {

console.log(data);

}

}

ngOnDestroy() {

this.subsciption.unsubscribe();

}

## Errors and Complete

With an Observer you can emit values using the next() method. In case an error occurs in the Observer, you can call the method error(). This will cancel the Observer. When the Observer has performed its task, you can call the complete() method. In both cases the Observer stops emitting values.

The subscriber of the Observable can call the subscribe() method to listen to new emitted values. The subscribe() method can take a second argument which is the method to perform in case of an error. It also can take a third argument which is the method to perform in case the Observer is completed.

this.subscription = customInterval.subscribe(data => {

console.log(data);

}, error => {

console.log(error);

alert(error.message);

}, () => {

console.log('complete');

});

)

## Operators

With operators you can modify the data that is emitted by an Observable. This allows you to modify the data before the data is send to the subscriber. See this [website](https://academind.com/learn/javascript/understanding-rxjs/) for more information.

## Subjects

Previously we learned how to use EventEmitters. It’s a way to communicate across components. If your component is submitting an event using @Ouput, you still have to use the EventEmitter. In all other cases, it's better to use Subjects. A Subject is a kind of Observable.

user.service.ts

@Injectable({providedIn: 'root'})

export class UserService {

activateEmitter = new Subject<boolean>();

}

app.component.ts

export class AppComponent implements OnInit {

userActivated = false;

subscription: Subscription;

constructor(private userService: UserService){}

ngOnInit() {

this.subsciption =

this.userService.subscribe(didActivate => {

this.userActivated = didActivate;

});

}

onDestroy() {

this.subscription.unsubscribe();

}

}

app.component.html

<p \*ngIf "userActivated">Active !</p>

user.component.ts

onActivate() {

userService.activateEmitter.emit(true);

}

user.component.html

<button class="btn btn-primary" (click)="onActivate()">Activate</button>

As with all observers that you created, make sure to unsubscribe to it in the onDestroy() method.

# Handling Forms in Angular Apps

## Template Driven Forms

When using the template driven approach, make sure you import the FormsModule into the app.module.ts file.

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

@ngModule({

imports: [

BrowserModule,

FormsModule,

HttpModule

]

})

In your HTML file you define a form (without the HTML action). It can contain a one or more input controls. For instance:

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

<div class="form-group" >

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

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

ngModel name="email">

</div>

<button type="button">Submit</button>

</form>

Angular will scan your HTML file for the form selector and creates a Typescript object of the form using the ngModel directive and the name (in the above example email). As soon as the user clicks the button, Angular will then call the onSubmit() method in your Typescript file.

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

onSubmit(form: NgForm) {

console.log(form);

}

The form variable contains the Typescript object of the form which is defined in your template. To get access to the input fields, you can access the form.value property to retrieve them by the name specified.

When you want to have access to the form before it is submitted, use the ViewChild decorator.

@ViewChild('f') form: ngForm;

### Validation

Often you want the validate the input of the user. Angular has a lot of validation directives you can add to your html elements.

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

ngModel name="email" required email>

The Typescript object will dynamically change the valid property of the input element and also the valid property of the form.

### Styling of controls based on the Form State

To prevent the user from submitting the form when not all the input elements are valid, you can add the disabled directive to the button.

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

To show the user visually that the input is not correct, you can give the invalid controls a red border. Just add the following code to the CSS file of the component. This snippet will give the input element in the HTML form a red border as soon as the user has touched the field and did not type a valid value.

.input.ng-touched.ng-invalid {

border: 1px solid red;

}

### Outputting validation error messages

When you only color the input element on your HTML page red, in case of an error, this will often not give the user enough information what he has done wrong. So outputting an error message is quite common. To show such an error message, add the following to the input element.

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

ngModel name="email" required email #email="ngModel">

<span class="help-block"

\*ngIf="!email.valid && email.touched">Please enter

a valid email address.

</span>

By adding a local reference to the input field and bind it with the ngModel directive, you can access it in the \*ngIf construction.

### Default values using property binding

To give an input element a default value, use property binding with the ngModel directive like below.

my.component.html

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

[ngModel]="defaultEmail" name="email" required email #email="ngModel">

my.component.ts

defaultEmail = "foo@yahoo.com";

### Three different techniques using ngModel

There are 3 different ways of using the ngModel directive.

1. ngModel 🡪 to inform Angular that it is an input element of a form.
2. [ngModel]="defaultValue" 🡪 using property binding to provide the element with a (default) value.
3. [(ngModel)]="answer" 🡪 Two-way-binding, to provide the input element with a value and to keep track of the value when it changes.

### Grouping form controls

When forms are large, it is a good idea to group some elements together. Each group can have its own validation. To define groups, use the ngModelGroup directive.

<div id="user-data" ngModelGroup="userData" #userData="ngModelGroup"> input elements here </div>

The ngModelGroup="userData" is telling Angular to create in a group of input elements with the name userData. To get access to this Javascript representation of this group, you use a local reference: #userData="ngModelGroup".

### Handling radio buttons

This paragraph will show how to use radio buttons in your form. Below you'll see an example.

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

<label>

<input

type="radio"

name="gender"

ngModel

[value]="gender"

required>

{{ gender }}

</label>

</div>

### SetValue and PatchValue

With the SetValue or PatchValue method, you can give the input controls in your form a value. The SetValue can be used to populate **all** input controls in your form.

@ViewChild('f') myForm: ngForm;

populateForm() {

const emailAddress = "foo@yahoo.com";

this.myForm.setValue({

userData: {

username: "foo"

email: emailAddress

},

gender: "male"

});

}

The PatchValue can be used to set the value of **one** or **more** input controls.

@ViewChild('f') myForm: ngForm;

populateForm() {

const emailAddress = "foo@yahoo.com";

this.myForm.form.patchValue({

userData: {

email: emailAddress

}

});

### Resetting Forms

To reset the form, you can call the reset() method. This will not only clear all values in the input controls, it will also reset the state. For instance, the touched and valid properties of the form. If you want to provide some initial data, you can pass this to the reset() method, just like you can with the setValue() method.

this.myForm.reset();

## Reactive Forms

It is important to mention that is not allowed to use template driven and reactive forms at the same time. You will get an error when you do so.

Remove the FormsModule import in the app.module.ts file and add the ReactiveFormsModule. Also make sure to remove all the ngModel directives in your templates.

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

@NgModule({

declaration: [ AppCompontent ],

imports : [

BrowserModule,

HttpModule,

ReactiveFormsModule, ..

]

})

### Creating a form in code

A userform in Angular is a Typescript object of type FormGroup. The input controls on the form are of type FormControl. Below you see an example of a signup form.

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

signupForm: FormGroup;

ngOnInit() {

this.signupForm = new FormGroup({

'username': new FormControl(null),

'email': new FormControl(null),

'gender': new FormControl('male')

});

}

### Syncing HTML and Form

To couple the Typescript form to your template, you must set some directives in the template. Below you will see how.

<form [formGroup]="signupForm">

<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="email" 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>

</form>

### Submitting a form

With a reactive form, you also have to add the ngSubmit directive to your form. Just as you did with the template driven approach. But you do not have to use a local reference to the form. You already have access to the form in the typescript code. Namely the FormGroup property signupForm.

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

onSubmit() {

console.log(this.signupForm);

}

### Adding Validation

To add validation to your input controls, you can provide Validators to the FormControl constructor. Make sure not to call the validator, like required(). But just provide the reference to it by only provide the name 🡪 Validator.required.

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

ngOnInit() {

this.signupForm = new FormGroup({

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

'email': new FormControl(null,

[Validators.required, Validators.email]),

'gender': new FormControl('male')

});

### Getting access to controls

Often you will show the user a message when he did not enter a valid value in a input control. See the example below how to do this with reactive forms.

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

signupForm.get('email').touched" class="help-block">

Please enter a valid email!</span>

You can style the input control just as with the template driven approach. Angular will add the style classes touched and valid, so you can make use of this.

.input.ng-touched.ng-invalid {

border: 1px solid red;

}

### Grouping Controls

The reactive way to group input controls, is to make use of the FormGroup component. In the example below we group the username and email in a group named userData.

ngOnInit() {

this.signupForm = new FormGroup({

'userData': new FormGroup(

{

'username': new FormControl(null),

'email': new FormControl(null),

})

'gender': new FormControl('male')

});

In our template we have to change the path to the grouped input controls.

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

signupForm.get('userData.email').touched" class="help-block">

### Arrays of Form Controls

In a form you can use an array of form controls. Suppose you want the user the provide his hobbies. You can then create a button which add an input control dynamically each time the user clicks this button.

ngOnInit() {

this.signupForm = new FormGroup({

'userData': new FormGroup(

{

'username': new FormControl(null),

'email': new FormControl(null),

})

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

'hobbies': new ArrayControl([])

});

onAddHobby() {

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

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

}

<div class="form-group"

\*ngFor="let control of signupForm.get('hobbies').controls;

let i of index">

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

</div>

### Creating custom validators

Normally the built-in validators are sufficient. But suppose you want a special validator, you can create your own. Suppose you want to forbid that the user to enter certain user names (for instance, ones that are already occupied).

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

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

return {'userAlreadyExist': true};

}

return null;

The notation [s: string]: boolean means that it is a key of type string which is part of a key-value pair. The value is of type boolean.

It is important that a validator return null in case the value in the control is valid. Do not return {'userAlreadyExist': false}, for instance.

Now the validator is ready, we can add it to the FormControl username:

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

ngOnInit() {

this.signupForm = new FormGroup({

'username': new FormControl(null,

[Validators.required, this.existingUserName.bind(this)]),

'email': new FormControl(null,

[Validators.required, Validators.email]),

'gender': new FormControl('male')

});

You have to bind 'this' to the validator, otherwise you will get an error:

EXCEPTION: Error in :0:0 caused by: Cannot read property 'existingUserName' of undefined

The reason is that Angular is that the method is not called from within the class. It is passed as a reference to the FormControl constructor. Therefore, Angular needs to know what is meant with this.

### Using Error Codes

In the previous paragraph we use a key-value pair as an error object. This error object is added to our form by Angular in case the input does not comply the validator. We can use the error objects to provide the user with the proper message.

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

This name is invalid!

</span>

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

This field is required!

</span>

### Creating Custom Async Validators

Sometimes a validation is asynchronous. For instance, to check if a user is valid, we need to access a database. We then need a Synchronous validator which returns a Promise or an Observable.

forbiddenEmail(controls: 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

When the user starts filling in the form, both the value and status is changing. For instance, the status of a field with an asynchronous validator can change from INVALID to PENDING to VALID. You can subscribe to both.

this.signupForm.valueChanges.subscribe( (value) => {

console.log(value);

});

this.sigupForm.statusChanges.subscribe( (status) => {

console.log(status);

});

### Setting and Patching Values

With the Reactive approach we can use the setValue, patchValue and reset methods on the form, just as with the Template Driven approach. For example, this.signupForm.setValue({}).

# Using Pipes to transform the output

Pipes are used to transform data in the output, i.e. the template. There are a lot of built-in pipes. An example is the uppercase pipe.

firstname='Bernie'

{{ firstname | uppercase }}

BERNIE

## Parameterize pipes

Some pipes take a parameter to configure it. For instance, the DatePipe.

{{ date || date: 'fullDate' }}

To see more on the built-in pipes, see the documentation on the internet: <https://angular.io/api?query=pipe>

## Chaining pipes

All pipes can be chained together. The output of a pipe can serve as the input for another pipe. Be aware that the order might be important.

{{ date || uppercase | date: 'fullDate' }}

The above example will result in an error because the date cannot be put in uppercase because it also holds numbers and special characters. The make it work, change the order of the pipes.

{{ date || date: 'fullDate' | uppercase }}

## Build your own pipe

Creating your own pipe, is just creating a new file and give it a name like shorten.pipe.ts. Below you see an example of a pipe that shortens the input to 10 characters.

import { Pipe, PipeTransform } 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;

}

}

{{ servername | shorten }}

It’s important that you also add the pipe to the app.module.ts file.

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

@NgModule({

declarations: [ AppComponent, ShortenPipe ]

})

## Parameterize Custom Pipes

In the above example, the length of the string returned by the pipe is hardcoded. The pipe should be more useful when you could provide the length as a parameter. The transform of this pipe is shown below.

transform(value: any, limit: number) {

if (value.length > limit) {

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

}

return value;

}

## Creating a Filter Pipe

When you have a list of servers with a certain status, and you want them to filter on this status, you can create a FilterPipe. Below you see such a pipe which takes three arguments. The first parameter is the array which holds the servers. The second parameter is the value to filter on. The third and last parameter is the property name to filter on.

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

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

return value;

}

const resultArray = [];

for (const item of value) {

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

resultArray.push(item);

}

}

return resultArray;

}

In your template you can use this filter pipe as follows:

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

<ul class="list-group">

<li

class="list-group-item"

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

[ngClass]="getStatusClasses(server)">

<span class="badge"> {{ server.status }}</span>

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

</ul>

## Pure and Impure Pipes

By default, a pipe only gets triggered when the input of the pipe changes. In the above example, when the filteredStatus is changing. When the list of servers is changing, the Pipe will not be triggered. The reason for this is that it could hit performance. If you want the pipe to be activated as soon as the server list is changing, you could add a property to the Pipe configuration.

@Pipe({

name: 'shorten'

pure: false

})

## The Async Pipe

The async Pipe can be used with Observables or Promises. Both are used for asynchronous communication. For instance, when you want to display the server status, which takes a couple of seconds. Use the async pipe.

<h3>{{ getServerStatus() | async }}</h3>

The method getServerStatus() will return a Promise<string>. When you will away the async pipe, it will be outputted in your template like: [object Object].

# Making HTTP Requests

To fetch or post data in Angular, you do this by sending HTTP requests. You never access a database directly but you always use a REST API of achieve this. This is because it is not safe to store your credentials in Typescript. In the Angular course they used Firebase from Google. This provide you with a nice collection of tools, such as a database wrapped in a REST API. When you create a database, you will get an URL of the REST API which you can use as URL to send HTTP requests.

## Sending an HTTP request

To be able to send an HTTP request, you must import the HttpClient from '@angular/common/http' in the app.module.ts file.

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

this.httpClient.post(

'https://url/posts.json', postData).subscribe(

(responseData) => {

console.log(responseData);

});

}

**Important note:** the HTTP request will only be send when you subscribe on it. The reason for this is that when no one is interested in the result of the HTTP request, it does not make any sense to send the request.

The extension .json in the URL is not REST or Angular specific but is the way to let Firebase know what to return in the REST API.

## Using Rxjs operators to transform the response

When sending an HTTP GET request, you'll get a response with data. This response needs to be converted into Typescript objects, so we can use them throughout the code.

private fetchPosts() {

The rxjs map() function allow us to transform the response before subscribing. The spread operator (...) copies all properties of the object.

this.httpClient.get('http://url/posts.json').pipe(

map(response => {

const posts = [];

for (const key in response) {

if (response.hasOwnProperty(key)) {

posts.push({...response[key], id: key});

}

}

return posts;

})).subscribe(posts => { this.loadedPosts = posts; })

}

## Using Types with the HttpClient

In the example of the previous paragraph we have the problem that the response is of type any[]. This can be overcome by specifying the type in the generic get method.

export interface Post {

This is the interface of the post entity. The question mark after id, means that this attribute is optional.

title: string;

content: string;

id?: string;

}

this.httpClient.get< {[key: string]}: Post}>('http://url/posts.json')

The response of the GET method, looks like this:

{ -L\_vioAfz8SsDnjJm3dec:

{

Content: "This is a test!"

Title: "Test"

}

}

The [key: string] refers to the key (-L\_vioAfz8SsDnjJm3dec), generated by Firebase.

## Using a Service for HTTP Requests

You must keep your components as lean as possible. It is a good practice to put the code that sends the HTTP request and does the transformations, in a service. Let the service return an Observable were the component can subscribe on.

private fetchPosts() {

return this.httpClient.get('http://url/posts.json')

.pipe(map(response => {

const posts: Post[] = [];

for (const key in response) {

if (response.hasOwnProperty(key)) {

posts.push({...response[key], id: key});

}

}

return posts;

})

);

}

In your component you can now call the service and subscribe on it. Remember, not subscribing on a HTTP request, does result in not sending it.

ngOnInit() {

this.isFetching = true;

this.postService.fetchPosts().subscribe(posts => {

this.loadedPosts = posts;

this.isFetching = false;

})

}

**NOTE:** if your component is not interested in the response/data, for instance when adding a Post. You can subscribe in the service.

## Handling Errors

When sending an HTTP request, a dozen of things can happen that will result in an error response. To handle an error response can be done in the same way as handling an error with Observables. So it can look like this:

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

this.postService.post(postData).subscribe(

(responseData) => {

console.log(responseData);

},

error => {

this.error = error.message;

console.log(this.error);

});

}

### Using Subject to handle Errors

In the previous paragraph we saw how to handle errors. It is also possible to use Subject to pass error information to your component. In your service you create a Subject and will call next on it in the error handler. In your component you subscribe on this Subject.

*post.service.ts*

error = new Subject<string>;

post(postData: { title: string, content: string}) {

this.httpClient.post(

'https://url/posts.json', postData).subscribe(

(responseData) => {

console.log(responseData);

},

error {

error.next(error.message);

});

}

*post.component.ts*

private errorSubscription: Subscription<string>;

private errorMsg: string;

ngOnInit() {

this.errorSubscription = this.postService.error.subscribe(errorMsg => {

this.errorMsg = errorMsg;

})

}

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

this.postService.post(postData).subscribe(responseData => {

console.log(responseData);

});

}

ngOnDestroy() {

this.errorSubscription.unsubscribe();

}

### Using the CatchError operator

There is also a possibility to handle errors with the catchError operator. See the example below.

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

private fetchPosts() {

return this.httpClient.get('http://url/posts.json')

.pipe(map(response => {

const posts: Post[] = [];

for (const key in response) {

if (response.hasOwnProperty(key)) {

posts.push({...response[key], id: key});

}

}

return posts;

}),

catchError(errorResponse => {

// Send data to analytics server

return throwError(errorResponse);

})

);

}

## Creating HTTP Headers

To change the HTTP header of your request, use the headers keyword in your HTTP request.

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

const HttpHeader = new HttpHeaders({

'Content-Type': 'application/json',

'Authorization': 'my-auth-token'

});

this.httpClient.get(this.weatherEndppoint, { headers: myHeaders })

.subscribe((response: Response) => { … });

## Using Query Params

Sometimes you need to provide some query parameters with your URL. Below you will find an example.

let queryParams = new HttpParams()

.append('q', searchValue)

.append('units', 'metric')

.append('lang', 'en')

.append('appid', this.apiKey);

this.httpClient.get(this.weatherEndppoint, { params: queryParams })

.subscribe((response: Response) => { … });

## Observing different type of Responses

Sometimes you do not only want the response data. In some cases, you might need the complete response (for instance information in the header). You can get this data by using the observe keyword in the HTTP method. Possible values are: 'body' (default), 'response' and 'events'.

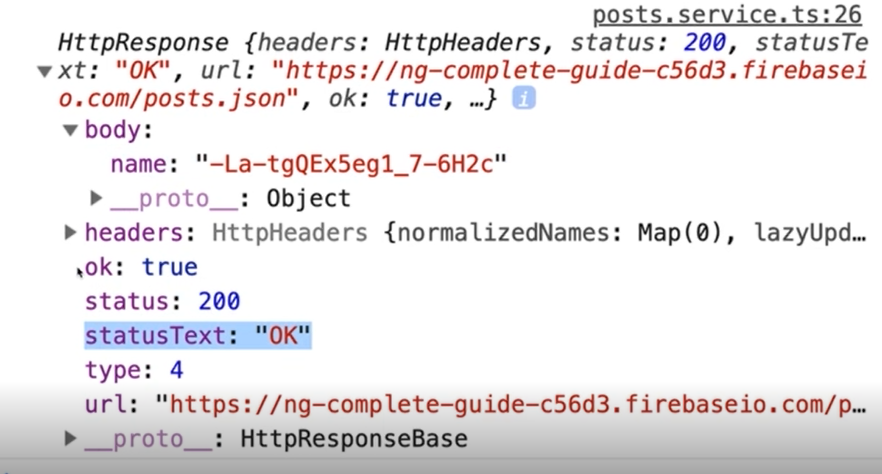
this.httpClient.post<name: string>(

this.url, postData, { observe: 'response' }).subscribe(response => {

console.log(response);

});

);



## Changing the Response Body Type

By default, the response body type is set to json. If the API you are communicating with, only supports text ('text', or it send you a file ('blob'). You can set it with the responseType parameter in the HTTP request.

this.httpClient.post<name: string>(

this.url, postData, { responseType: 'text' }).subscribe(response => {

console.log(response);

});

);

## Using Interceptors

In one of the previous paragraphs we read how to change a request's header. But his was for only one request. When you want to add some information to all requests send by the application, you can use Interceptors. As the name suggests, it will intercept the HTTP request, perform the actions in the intercept() method and continue.

import {

HttpInterceptor,

HttpRequest,

HttttpHandler } from '@angular/common/http';

export class AuthInterceptorService implements HttpInterceptor {

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

const modifiedRequest = req.clone({

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

});

return next.handle(modifiedRequest);

}

}

**IMPORTANT:** The last statement will make sure that the actual request is send. If you forget to call the HttpHandler, the HTTP request will NOT be send.

The request is immutable. So if you want to change something in the request, you have to clone it first. The clone you can edit and pass it to the handle() method of the HttpHandler. If you do not want to change all HTTP requests, you can add an if statement in the intercept method and check the url of the request.

Now the interceptor implementation is ready, we have to tell Angular to use this implementation. This can be done in the app.module.ts file.

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

@NgModule({

providers: [{ provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptorService, mulit: true]}],

})

It is also possible to intercepts the response of the HTTP requests.

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);

}

})

);

## Running Multiple Interceptors

When you want to have multiple interceptors, you can configure this in the app.module.ts file.

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

@NgModule({

providers: [

{

provide: HTTP\_INTERCEPTORS,

useClass: AuthInterceptorService, mulit: true]

},

{

provide: HTTP\_INTERCEPTORS,

useClass: LoggingInterceptorService, mulit: true]

}

],

})

# Authentication and Route Protection

// TODO //

# Dynamic Components

There are two ways of defining dynamic components. The first and preferable way is using the \*ngIf directive. The other way is using Typescript code. Therefor I only will show an example where we adding an Error Modal Alert Box. This is a window that is on top of any other window and must be closed before you continue working in the application.

The template of the Error Modal Alert Box, look like:

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

<div class="alert-box">

<p>{{ message }}</p>

<div class="alert-box-actions">

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

</div>

</div>

The Typescript code:

import { Input, Component, 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();

}

}

Do not forget to add the component to the app.module.ts file.

This Error Modal Alert Box can now be called by using the selector's name. Below you see a fragment of the auth.component.html template file.

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

</app-alert>

In the auth.component.html, you listen on the close event. If this event is emitted (when the user clicked the Close button or somewhere on the backdrop background), the method onHandleError() is executed which will reset the error by setting it to null. The \*ngIf directive will then hide the app-alert component.

The Typescript code:

@Component({

selector: 'app-auth',

templateUrl: './auth.component.html'

})

export class AuthComponent {

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 authObservable: Observable<AuthResponse>;

if (this.isLoginMode) {

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

} else {

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

}

authObservable.subscribe(

authResponse => {

console.log(authResponse);

this.isLoading = false;

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

},

errorMessage => {

console.log(errorMessage);

this.error = errorMessage;

this.isLoading = false;

}

);

form.reset();

}

onHandleError() {

this.error = null;

}

# Splitting Modules

When your application code starts growing, it is a good practice to split up your code in Feature Modules. We already saw this with the RoutingModule. All components, directives, pipes that are used in a Module needs to imported. The exception are services; these only needs to be declared in your app.module.ts file. There is also another exception and that is the BrowserModule. This module only needs to be imported in your app.module.ts file. If your module is using for instance the \*ngIf directive, import CommonModule instead of the BrowserModule.

Below you will see an example of a RecipesModule, which is declared in the recipes.model.ts file. Note, for simplicity the import statements below are not mentioned.

@NgModule({

declarations: [

RecipesComponent,

RecipeDetailComponent,

RecipeListComponent,

RecipeItemComponent,

RecipeStartComponent,

RecipeEditComponent

],

imports: [

CommonModule,

RouterModule,

ReactiveFormsModule,

RecipesRoutingModule

]

})

export class RecipesModule{}

## Routing in Feature Modules

When your Feature Component contains routes, these can be put in a separate file inside the Feature Component. Move the route from the main routing app-routing.component.ts file to the new routing file, for example the recipes-routing.module.ts file.

const routes = [

{

path: 'recipes', component: RecipesComponent,

canActivate: [AuthGuard],

children: [

{path: '', component: RecipeStartComponent},

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

{path: ':id', component: RecipeDetailComponent,

resolve: [RecipeResolverService]},

{path: ':id/edit', component: RecipeEditComponent,

resolve: [RecipeResolverService]}

]

}

];

@NgModule({

imports: [

RouterModule.forChild(routes)

],

exports: [

RouterModule

]

})

export class RecipesRoutingModule{}

## Core Module

When you are **NOT** using the directive @Injectable({ providedIn: 'root' }) in your services, you could remove the provided tag in the app.module.ts file and bundle all the provided services in a core module.

@NgModule({

providers: [

RecipeService, ShoppingListService,

{ provide: HTTP\_INTERCEPTORS,

useClass: AuthInterceptorService,

multi: true }

]

})

export class CoreModule {

}

**NOTE:** It is highly recommended to use the providedIn tag in your services.

# Implementing Lazy Loading

In all previous examples, we used eager loading. This means that all modules are loaded as soon as the application starts. To optimize the application startup time, you could use lazy loading. This will load the module at the time the user navigates to a route. The prerequisites for lazy loading are:

1. The application needs to be split up in Components.
2. The routes in a component needs to be declared in a RouteComponent.

In the app-routing.module.ts file you must specify the root of the component route. See below.

const appRoutes: 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(appRoutes)],

exports: [RouterModule]

})

export class AppRoutingModule {}

The root of the component needs to be modified in an empty (relative) path. This is because in the app-routing.module.ts file the root of the component is already declared.

const routes: Routes = [

{ // the route below was previously 'recipes'

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 {}

**NOTE:** Do not forget to remove the RecipesModule from the import section in the app.module.ts file. Also remove the unused imports, otherwise the component will still be loaded.

All the above changes will result in loading the RecipesModule at the time the user navigates to this route.

## Preloading Lazy-Loaded Code

When you have setup lazy loading, the modules get loaded when the user navigates to it. This has one major downside. When the Module code bundle is large, the application needs to load that bundle at the time it is needed. This can cause in a slow response, depending on the internet connection speed. To avoid this, Angular provide an option to preload the lazy loaded modules. To activate this, adapt the app-routing.module.ts file.

@NgModule({

imports: [RouterModule.forRoot(routes),

{ preloadingStrategy: PreloadAllModules }],

exports: [RouterModule]

})

export class RecipesRoutingModule {}

As soon as the application is ready, it will load all other modules. This will increase the change that the module is already loaded, when needed.

# Deployment

## JIT vs AOT

When developing an application, you use ng serve to compile and run your application. The bundle which will then be deployed contains the Angular compiler. This makes the bundles quite big. This type of compilation is called **Just in Time** Compilation. When deploying your application to a production environment, you want to optimize your bundles. Keep them as small as possible. Therefor you could build your code with the ng build –prod command. This command will bundle your code without the Angular compiler. It will make the bundles much smaller. This type of compilation is called **Ahead of Time** compilation. The result of the compilation is stored in the dist\<MyAppName> folder. To deploy this on a server, wrap the contents of this folder in a zip file and give it the extension .war. For example: RecipeBooksApp.war.

## Environment Variables

Often the API's and servers which you use during development are different than the ones in production. You could use the environment.ts and environment.prod.ts files to define the differences.

*environment.ts*

export const environment = {

production: false,

restApiUrl: 'https://recipes.development.org/recipes'

}

*environment.prod.ts*

export const environment = {

production: true,

restApiUrl: 'https://recipes.production.org/recipes'

}

In your code you can now refer to these variables.

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

return this.httpClient<RecipeResponse>.get(

this.environment.restApiUri);

The Angular compiler will automatically swap the variables between development and production. The ng serve command will use the variables defined in the environment.ts file, the ng build –prod command will use the environment.prod.ts file.

# Unit Testing

As with almost all modern languages, you should write unit test. Angular code can be tested by using the Jasmine test framework and the Karma test runner. Other test frameworks and runners are available.

Below you find a very simple unit test to explain the basics.

A unit test is described with the describe function. It contains a beforeEach function which execute code before each test is run. All the it functions describe a single test. All tests are running independent from each other. These functions are provided by Angular. The expect method is provided by the Jasmine framework. The tests are **NOT** running in a browser. To get the component rendered, call the detectChanges method on the fixture.

import { TestBed, async } from '@angular/core/testing';

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

describe('App: CompleteAngularGuideExample', () => {

beforeEach( () => {

TestBed.configureTestingModule({

declarations: [ AppComponent ]

});

});

it('should create the app', async( () => {

let fixture = TestBed.createComponent(AppComponent);

let app = fixture.debugElement.componentInstance;

expect(app).toBeTruthy();

}));

it('should have a title'), async( () => {

let fixture = TestBed.createComponent(AppComponent);

let app = fixture.debugElement.componentInstance;

expect(app.title).toEqual('app works!');

}));

it('should render title in a h1 tag'), async( () => {

let fixture = TestBed.createComponent(AppComponent);

fixture.detectChanges();

let compiled = fixture.debugElement.nativeElement;

expect(compiled.querySelector('h1').textContent)

.toContain('app works!');

}));

}

);

Unit tests can be started with the ng test Angular CLI command. The output of the above tests should look like below:

10% building modules 1/1 modules 0 active   
...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/   
...INFO [launcher]: Launching browser Chrome ...   
...INFO [launcher]: Starting browser Chrome   
...INFO [Chrome ...]: Connected on socket ... Chrome   
...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)

When you keep this command running, changes in the tests or application will be automatically detected by the Angular framework and it will trigger the unit tests again.

## Test dependencies and Services

In Angular, dependencies and services are injected automatically by the framework. In unit tests, this is **NOT** the case. You will have to inject them yourself. Use the injector function, as shown below.

it('should use the username from the UserService', () => {

let fixture = TestBed.createComponent(AppComponent);

let app = fixture.debugElement.componentInstance;

let userService = fixture.debugElement.injector.get(UserService);

fixture.detectChanges();

expect(userService.user.name).toEqual(app.user.name);

});

Components are rendered automatically in Angular when running in the browser. In unit tests you have to trigger the rendering yourself. As you seen in the previous paragraph, use the detectChanges method on the fixture (the instance of the component under test).

## Simulating Asynchronous tasks

In the previous paragraph we injected a synchronous service. The result of the service method is coming instantly. To test asynchronous services (for instance calls to a database server), you must use the async function.

export public class DataService {

getDetails(): Promise(string)

}

it('should fetch data if called asynchronously', async( () => {

let fixture = TestBed.createComponent(AppComponent);

let app = fixture.debugElement.componentInstance;

let dataService = fixture.debugElement.injector.get(DataService);

let spy = spyOn(dataService, 'getDetails').and.returnValue(

Promise.resolve('Data'));

fixture.detectChanges();

fixture.whenStable().then( () => {

expect(userService.user.name).toEqual(app.user.name);

});

}));

The spyOn method will be fired as soon as the asynchronous call to the service is made. It will then return a value as specified in the returnValue method. With this approach, a stub will be created of the actual call. This prevents that the unit test is actual reaching out to for instance a database server. Which is something you do not want in a unit test.

The whenStable method waits until the async task is performed. Note that in the unit test the task is not really performed in an asynchronous way, we are just faking it. We are returning a string wrapped in a Promise which resolves instantly.

## Using fakeAsync and Tick

The test in the previous paragraph can also be written with fakeAsync. The tick method will finish all (fake) asynchronous tasks. So instead of waiting until they are finished (with the whenStable method), you tell Angular to finish them. Both fakeSync and tick must be imported from '@angular/core/testing'.

it('should fetch data if called asynchronously', fakeAsync( () => {

let fixture = TestBed.createComponent(AppComponent);

let app = fixture.debugElement.componentInstance;

let dataService = fixture.debugElement.injector.get(DataService);

let spy = spyOn(dataService, 'getDetails').and.returnValue(

Promise.resolve('Data'));

fixture.detectChanges();

tick();

expect(userService.user.name).toEqual(app.user.name);

});

}));

## Isolated vs non-Isolated testing

Some components, pipes of services can be tested in isolation. This means that you only test that specific Angular element. For instance, pipes. They just take an input and they return an output. There is no need to use the Angular Jasmine test framework for this.

*reverse.pipe.ts*

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

@Pipe({

name: 'reverse'

})

export class ReversePipe {

transform(value: string) {

return value.split("").reverse().join("");

}

}

*reverse.pipe.spec.ts*

import { ReversePipe } from './reverse.pipe';

describe('Reverse Pipe', () => {

it('reverse hello using the ReversePipe', () => {

let reversePipe = new ReversePipe();

expect(reversePipe.transform('hello')).toEqual('olleh');

});

})

As soon as your component or service is depended on Angular, use the Jasmine Framework (or another testing framework) to create the unit test. This kind of testing is called non-isolated testing.