Angular: framework for building client applications in html/css and javascript/TypeScript

Why we need angular: because with vanilla Js and jQuery is har to maintain and teste in a big project

structure of angular app:

* e2e : end to end test for application
* node\_modules : this is where we store all the third party libraries that application made depend on

when we compile our application parts of this third-party libraries are put in a bundle and deployed with our application

* src : actual source code of the application.
  + app : contain model and component, each application has at list on model and one component.
  + assets : here we store all images file, text files ,icons …
* environments:
  + store configuration setting of environments
* main file

starting point of the application / bootstrapping the main module

* polyfills.ts

import some file required for angular

* styles.css

the global style of our application

* test.ts

setting test of the application

webpack:

* angular CLI use webpack tool to build automatization tool.
* Get all script and stylesheet and combine them in a bundle and then minified them for optimization
* Web pack recompile the application and refresh the bundle / hot module replacement (HMR)
* All stylesheets are compiled in javascript bundel

Angular Histories :

AngularJs - Angular2 - Angular4

# Typescript Fundamentals

* Any java script code is a Typescript /Type script has additional feature than JavaScript.
* Typescript: Strong Typing (Opt) , OOP, Compile-time errors, Great tooling

typescript -- transpile 🡪 JavaScript

* tsc main.ts génère un fichier java script.
* node main.js
* Modules : in type script each file is a module

**export**

export class Point {

**import**

import { Point } from './point';

1. **Angular Fundamentals**

Building Blocks of angular Apps

* Component: Data, Html Template, Logic
* App component is a root component
* **Modules**: is container of group of related components every angular app has at list one module which we call angular Module

Ex: courses model, Messaging model, instructor model, admin model…

To use a component there is tree steps to follow:

* 1. **Create** a component
  2. **Register** it in a module
  3. Add an element in a **HTML markup**
* **String Interpolation**

<h2>{{ getTitle() }}</h2>'

* **Directive:**

 <li \*ngFor="let course of courses"> {{course}} </li>

* **Service:**

ng d s [nom de service]

* **Property Binding**
* Bind a property of the DOM to a filed in a component.
* One-way binding from component to the dom.
* Property binding work only for DOM object NOT HTML element.
* To target a DOM attribute with property binding we use the following syntax

<h1 [textContent]=”title”></h1>

* **Attribute Binding**
* To target html attribute with property binding we use the following syntax

<td [attr.colspan]=”colspan”></td>

* **Bootstrap**

ˆ3.3.7: major.minor.patch

**Download all dependency**: npm install

* **Class Binding**

[class.active]="\_isActive"

* **Style Binding**

<button class="btn btnprimary" [style.background]="\_isActive ? 'blue' : 'white'">btn</button>

* **Event binding**

onSave($event){

    $event.stopPropagation();

    console.log("Button was clicked",$event.clientX);

}

* **Event filtering**

   <input (keyup.enter)="enter()"/>

enter($event){

console.log($event.target.value);

}

* **Template variables**

 <input #email (keyup.enter)="getEmail(email.value)"/>

getEmail(text){

    console.log(text);

}

* **Two-way binding**

     <input [(ngModel)]="\_email" (keyup.enter)="printEmail()"/>

\_email="mohamedamjoud1@gmail.com";

courses;

printEmail(){

console.log(this.\_email);

}

* **Pipes**

{{ course.title | uppercase}} <br/>

{{ course.students | number }} <br/>

{{ course.rating | number: '2.1-2' }} <br/>

{{ course.price | currency: 'AUD':true:'3.2-2' }} <br/>

{{ course.releaseDate | date }} <br/>

1. **Building Re-usable Components**

In order to make a component reusable you want add input and output property.

input property: to pass input or state to a component

output property: to raise event from this custom component

the combination between input and output property make up the public API a component

* **Input**

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

@Input() isFavorite :  boolean = false;

* **Aliasing Input Properties (give to a property a nickName)**

@Input('is-Favorite') isFavorite :  boolean = false;

* **Output**

*component*

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

@Output() change = new EventEmitter();

 onClick(){

    this.isFavorite = !this.isFavorite;

    this.change.emit();

  }

*Call the component*

<course [isFavorite]="post.isFavorite" (change)="onFavoriteChange()"></course>

* **Output - Passing event data**

*Component*

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

@Output() change = new EventEmitter();

onClick(){

  this.isFavorite = !this.isFavorite;

  this.change.emit({ newValue : this.isFavorite });

}

export interface FavoriteChangedEventArgs{

  newValue : boolean;

}

*Call the component in html page*

<course [isFavorite]="post.isFavorite" (change)="onFavoriteChange($event)">

</course>

*Definition of* ***onFavoriteChange***

onFavoriteChange(eventArgs : FavoriteChangedEventArgs){

  console.log("Favorite changed" + eventArgs.newValue);

}

* **Styles**

We define style of component in three ways:

By using styleUrls,By using styles Or by using style markup in html file

@Component({

  selector: 'course',

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

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

  styles:[

  `

  .fa-star{

    color : red;

    font-size : 50px;

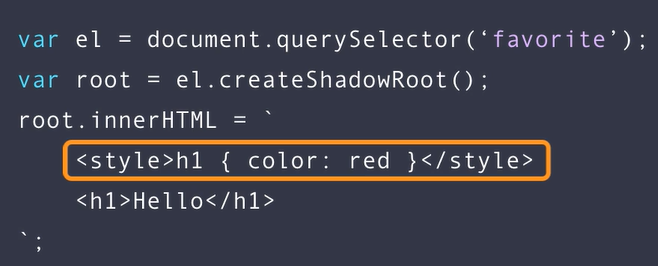
  }

  `]

})

* **Shadow DOM**

Allows us to apply scoped styles to elements without bleeding out to the outer world.



*Apply shadow behavior*

 encapsulation: ViewEncapsulation.Emulated

 encapsulation: ViewEncapsulation.Non

*to avoid*

 encapsulation: ViewEncapsulation.Native

* **ng-content**

Allow to consumer of the component to provide custom content

Definition of component

<div class="card">

    <div class="card-header">

        <ng-content select=".heading"></ng-content>

    </div>

    <div class="card-body">

        <ng-content select=".body"></ng-content>

    </div>

</div>

Call component and custom content

<bootstrap-panel>

    <div class="heading">Heading</div>

    <div class="body">

        <h1>Body</h1>

        <p>Some content here ...</p>

    </div>

</bootstrap-panel>

* **ng-container**

we use it to costume a component when we don’t need to add extra markup

(div , span …)

    <ng-container class="heading">Heading</ng-container>

1. **Directives**

* Structural: Modify the structure of the DOM
* Attribute: Modify the attribute of DOM element
* **Ng-if (structural)**

**We use it if we use a large tree**

<div \*ngIf="courses.length > 0 ; then coursesList else noCourses"></div>

<ng-template #coursesList>

    List of courses

</ng-template>

<ng-template #noCourses>

    There is no courses

</ng-template>

* **Hidden Property (attribute)**

**For small element tree**

<div [hidden]="courses.length == 0">

    List of courses

</div>

* **ngSwitchCase**

<ul class="nav nav-pills">

    <li>

<a (click)="viewMode = 'map'" class="nav-link" [class.active]="viewMode == 'map'">Map View</a></li>

    <li >

<a (click)="viewMode = 'list'" class="nav-link" [class.active]="viewMode == 'list'">List Views</a></li>

</ul>

<div [ngSwitch]="viewMode">

    <div \*ngSwitchCase="'map'">Map View Content</div>

    <div \*ngSwitchCase="'list'">List View Content</div>

    <div \*ngSwitchDefault>Otherwise</div>

</div>

* **ngFor**

<ul>

    <li \*ngFor="let team of teams; index as i">

        {{ i }} - {{team.name}}

    </li>

</ul>

* **Change Detection**

<ul>

    <li \*ngFor="let team of teams">

        {{team.name}} <button (click)="onRemove(team)" >Rmove</button>

    </li>

</ul>

When the object is removed from the model the view is notified.

Change Detection track object by reference in memory or we can costume how it will track object

By tracking object, we avoid updating DOM elements every time and by this we optimize performances.

* **Costume change detection**

<li \*ngFor="let team of teams; trackBy: trackCourse">

        {{team.name}} <button(click)="onRemove(team)" >Rmove</button>

    </li>

trackCourse(index,course){

    return course ? course.id : undefined;

 }

loadObject(){

  this.teams = [

    {id:1,name:'Fcb'},

    {id:2,name:'Real Madrid'},

    {id:3,name:'Liverpol'},

    {id:4,name:'Man City'},

  ];

}

* **ngClass**

<span class="fa"

[ngClass]="{

    'fa-star-o' : isSelected,

    'fa-star' : !isSelected

}"

(click)="onClickStar()">

</span>

* **ngStyle**

<button

 [ngStyle]="{

     'backgroundColor': isSelected ? 'blue' : 'gray',

     'color' : isSelected ? 'white' : 'black'

 }"

>

Save

</button>

* **Safe Traversal Operator**

<span>{{document.signatorie?.name}}</span>

* **Creating custom directive**

**Command:** ng g d input-format

**Registered The directive into app.module.ts file**

**HostListener:** allows to subscribe to event raised by the DOM element.

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

**Example:**

1 : @Input('appInputFormat') format;

2 :  constructor(private el:ElementRef) { }

  @HostListener('blur') onBlur(){

    if(this.format == 'uppercase')

    {

      console.log(this.format);

      let val : string =this.el.nativeElement.value;

      this.el.nativeElement.value = val.toUpperCase();

    }

    else if (this.format == 'lowercase')

    {

      console.log(this.format);

      let val : string =this.el.nativeElement.value;

      this.el.nativeElement.value = val.toLowerCase();

    }

**1 :** Use @Input(‘format’) directive to get parameter from view

**2 :** Inject an element reference object. To get the element using this directive.

**Call the directive in the view:**

<input type="text" [appInputFormat]="'lowercase'"/>

1. **Template driven forms**

## Building a bootstrap form

## Types of forms

* **FormControl** class, each control filed in form we need to create an instance of the control class.
* **FormGroup**: represent group of control in a form, each form is essentially a control Group.

In complex application it must have a form with multiple control groups.

**Property of FormControl and FormGroup :** Value, touched, untouched, dirty, pristine, valid, errors.

**We can create controls:**

**Using Directives: Template-Driven**

* Good for simple forms
* Simple validation
* Easier to create
* Less code

**Using Code: Reactive**

* More Control over validation logic
* Good for complex forms
* Unit testable

## NgModel

<input id="firstName" name="firstName" ngModel #firstName="ngModel"  (change)="log(firstName)"  type="text" class="form-control">

We use ngModel the bind the form control with the model, to pass the ngmodel to the log method in parameter in the change event we create a template variable #firstName

## Adding validation

<div class="alert alert-danger"\*ngIf="firstName.touched && firstName.invalid"> First Name is required </div>

## Specific validation errors

<div class="alert alert-danger" \*ngIf="firstName.touched && !firstName.valid">

     <div \*ngIf="firstName.errors.required">First Name is required</div>

     <div \*ngIf="firstName.errors.minlength">

First Name shoud be more than {{firstName.errors.minlength.requiredLength}}

</div>

     <div \*ngIf="firstName.errors.pattern">

First Name don't matche to expected values

</div>

</div>

## ngForm:

<form #contactForm="ngForm" (ngSubmit)="submit(contactForm)">

## ngFormGroup:

we use it to organize and modify the structure of NgForm.value property.

<div ngModelGroup="contact">

<input…>

<label…>

<div>

# Reactive Forms

## Creating controls programmatically

form = new FormGroup ({

    username : new FormControl(),

    password : new FormControl()

});

<form [formGroup] = "form">

<input formControlName = "username" type="text" name="Username" id="Username"

class="form-control">

## Adding validation

form = new FormGroup({

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

    password : new FormControl()

  });

get username(){

    return this.form.get('username');

  }

<div \*ngIf="username.touched && username.invalid" class="alert alert danger">The user name is required</div>

## Implementing Custom validation

Class: UsernameValidators

export class UsernameValidators{

static cannotContainSpaces(control : AbstractControl): ValidationErrors | null{

         if((control.value as string).indexOf(' ') >= 0)

            return {cannotContainSpaces : true};

         return null;

}

Class: ReactiveFormsComponent

export class ReactiveFormsComponent implements OnInit {

form = new FormGroup({

    username : new FormControl('',[

      Validators.required,

      UsernameValidators.cannotContainSpaces

    ]),

    password : new FormControl()

});

Class: component

<div \*ngIf="username.errors.cannotContainSpaces">The user name cannot contain spaces</div>

## Implementing Custom validation asynchronous

Class: UsernameValidators

static shouldBeUnique(control: AbstractControl): Promise<ValidationErrors | null>{

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

            setTimeout(() => {

                if(control.value === 'mosh')

                    resolve({ shouldBeUnique : true});

                 else

                    resolve(null);

            }, 2000);

        });

}

Class :ReactiveFormsComponent

form = new FormGroup({

    username : new FormControl('',[

      Validators.required,

      UsernameValidators.cannotContainSpaces,

    ],

    UsernameValidators.shouldBeUnique),

    password : new FormControl()

  });

<div \*ngIf="username.errors.shouldBeUnique">The user name are alerady taken</div>

## Showing loader

<div \*ngIf="username.pending">Checking for Uniqueness</div>

## NestedFormGroups

## FormArray

## FormBuilder

# Consuming HTTP Services

## Get

constructor(http:HttpClient) {

      http.get('https://jsonplaceholder.typicode.com/posts')

      .subscribe(Response=>{

        this.posts = Response as string;

          // console.log(Response);

      });

  }

## Post :

public addPost(input : HTMLInputElement){

 let post = { title : input.value };

 this.http.post("https://jsonplaceholder.typicode.com/posts", post).subscribe

(respense=> {

      this.posts.splice(0,0,post);

      input.value = "";

     });

}

## ngOnInit

## Handling Errors

|  |  |
| --- | --- |
| Unexcepted | Expected |
| Server is offline | “Not found” errors (404) |
| Network is down | “Bad request” errors (400) |
| Unhandled exception |  |

## Handling Unexpected Errors:

 delatePost(post){

    this.postService.deletePost(post)

    .subscribe(

      respense=> {

        console.log(respense);

        },

      error=>{

        alert('An unexpected error occurred.');

        console.log(error);

      });

  }

## Handling Expected Errors:

  delatePost(post){

    this.postService.deletePost(post.id)

    .subscribe(

      respense=> {

        let index = this.posts.indexOf(post);

        this.posts.splice(index,1);

      },

      (error : Response)=>{

        if(error.status === 404)

          alert('This post has already been deleted.');

        else {

          alert('An unexpected error occurred.');

          console.log(error);

        }

      });

  }

## Throwing Application-specific Errors

**Post.service.ts**

 deletePost(post){

    return this.http.delete(this.url + '/' + post.id)

    .pipe(

      catchError((error : Response) => {

        if(error.status === 404)

          return Observable.throw(new NotFoundError());

        return Observable.throw(new AppError(error));

      })

    );

  }

Poste.component.ts

  addPost(input : HTMLInputElement){

    let post = { title : input.value };

    this.postService.addPost(post)

    .subscribe(

      respense=> {

        this.posts.splice(0,0,post);

        input.value = "";

      },

      (error : AppError) => {

        if(error instanceof BadRequestError){

           this.form.setErrors(error.originalError);

        }else{

          this.notFoundError(error);

        }

      });

  }

  notFoundError (error : AppError){

    if(error instanceof NotFoundError)

      alert('An unexpected error occurred.');

      console.log(error);

  }

## Optimistic vs Pessimistic Approche

You can always convert observables to promises

Prefer observables

They allow reactive programming

They provide a bunch of useful operators

Observables are lazy

# Routing and navigation:

## Routing in a Nutshell

**Steps :**

1.Configure the routes

2.Add a router outlet

3.Add links

## Configuration

**Import RouterModule**

import { RouterModule } from '@angular/router'

in the app.module.ts in the import section we add :

RouterModule.forRoot([

{

path: '',

  component: HomeComponent

},

{

path: 'followers/:username',

  component: GitHubProfileComponent

}

* **RouterOutlet**

This directives allow to display the component that matche url entered,

<router-outlet></router-outlet>

## RouterLink

For static link we use directive as a attribute:

<a class="nav-link" routerLink="/posts">Posts</a>

For dynamic link we use property Binding:

<a [routerLink]="['/followers', follower.id ]">{{ follower.name }}</a>

## RouterLinkActive

By using this directive routerLinkActive angular manage the navbar css style

<li class="nav-item" routerLinkActive="active current">

## Getting the route parameter

ActivatedRoute help to get the param of the active link.

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

constructor(private route: ActivatedRoute) { }

this.route.paramMap

      .subscribe(params => {

         let id = +params.get('id');

         console.log(id);

      });

## Why route parameters are observable?

Because we need to subscribe to route parameters, each time the parameters changed (when link changed) the subscribers will be notified.

we use observable in workflow when we will stay in the same component. Like pagination

we can get the parameters without subscribing in observables, we use this approach when the user will immediately change the component

this.route.snapshot.paramMap.get('id');

## QueryParam:

this.route.queryParamMap

      .subscribe(params => {

         let id = +params.get('userName');

      });

this.route.snapshot.queryParamMap.get('userName');

## Subscription to multiple Observables:

combineLatest(

      this.route.queryParamMap,

      this.route.queryParams,

    ).subscribe(combinedObservable=>{

      let id = combinedObservable[0].get('id');

      this.gitHubFollowersService.get()

      .subscribe(followers=>{

        this.followers = followers

      })

    });

## SwitchMapOperator

We use SwitchMap operator inside pipe to return an observable<[]>. We use it in this example to avoid using subscribe twice.

If we use map we return observable<observable<[]>>.

combineLatest(

     this.route.queryParamMap,

      this.route.paramMap

    ).pipe(

      switchMap(combined => {

        let id = combined[1].get('id');

        return this.gitHubFollowersService.get();

      })

    )

    .subscribe(followers=>{

        console.log(followers);

    });

## Programmatic Navigation

submit(){

    this.router.navigate(['/followers'],{

      queryParams : {page:1, order:'newest'}

    });

  }

# Authentication and authorization

## JWT (JSON WEB TOKEN)

Json web tokens is a long string encoded in base64 with an algorithm of Hach, contain three blocks:

HEADER {“algo”: “HS256”, “typ”: “JWT”}

PAYLOAD: object json contain the data of the user.

VERIFY SIGNATURE : HMACSHA256(

Base64UrlEncoded(header) + “.” +

Base64UrLEncoded(payload),

Secret key)

## HttpInterceptor

We implement this interface to intercept http request.

We can use it to make a fake backend or to better managed the http request

## localStorage

we use localStorage to save the token in the browser

## Http

This example of http request with post method and parameter, we get in the response token

-authService

return this.http.post('/api/authenticate',credentials)

      .pipe(

        map(response => {

           if (response && response.hasOwnProperty('token') ){

            localStorage.setItem('token',response['token']);

            return true;

           }

           else

            return false;

        }),

        catchError(this.handleError)

      );

-loginComponent:

signIn(loginForm : NgForm){

    this.authService.login(loginForm.value)

      .subscribe(response=>{

        let returnUrl = this.route.snapshot.queryParamMap.get('returnUrl');

        if (response)

          this.router.navigate([ returnUrl || '/' ]);

         else

          this.invalidLogin = true;

      });

  }

## ReturnUrl

We use this approach to return the path needed by the user before he is authenticated.

## CanActivate

We use can activate interface to protect and secure our path, path that will open to authenticated user or user with specific roles.

## JwtHelperService

We use this class to manage the token stored in the browser.

return !jwtHelper.isTokenExpired(token);

return new JwtHelperService().decodeToken(token);

## Accessing to protected API Ressources.

getOrder(){

    let token = localStorage.getItem('token');

    let headers = new HttpHeaders().append('Authorization','Bearer'+token);

    return this.http.get('/api/orders',{headers})

      .pipe(

        map(response=> response),

        catchError(this.handleError)

      );

  }

# Deployment

## Preparing for Deployment

To deploy an angular application, the simple and ugly why is to copy and past all project in the server and run it with ng serve method.

The good why is to do Optimization

Minification : minifier les fichiers, condensé le contenue

Uglification : rename the long name in short names

Bundling : assembler les fichiers js dans un seul fichier

Dead code elimination

For all that we use ng build --prod command

## JIT & AOT Compilation

JIT:

* Inefficient for production
* Happens for every user
* More components, slower
* We have to ship angular compiler

AOT:

* Faster startup
* Smaller bundle size
* Catch template errors earlier
* Better security

## Environment

To do a specific configuration to prod or dev environment we uses environment.ts or environment.prod.ts.

To run project like prod environment we uses ng serve --prod=true

## Linting

Tsli nt.json to have single uniform in the whole project

ng lint

ng lint --fix : to fix the error by the angularCli

for an interactive syntax error erasing can use TSlint extension

## Deployment

GitHub Pages: no back-end

1. We create a github repository
2. We install node package for deploy in github pages : npm i -g angular-cli-ghpages
3. Build application using angular cli : ng build - -prod - -base-href=[https://mohamedamjoud.github.io/HelloWorld](https://mohamedamjoud.github.io/helloworld)/
4. Publish with: ngh - -no-silent

Firebase: Firebase as the nack-end

Heroku: Custom back-end