**Mémo Angular**

Utiliser WebStorm, un IDE spécifique pour JS et Angular

Concepts

**Node.js** est une bibliothèque et un environnement d’exécution multi-plateforme pour exécuter des applications JavaScript en dehors du navigateur. Il s’agit d’un outil gratuit et open source utilisé pour créer des applications JavaScript côté serveur.

**Node.JS est un outil :**

- utile pour créer des applications de mise en réseau côté serveur rapides et évolutives

- est un langage idéal pour développer des projets de petite taille

**AngularJS** est un framework de développement d’applications Web open source.

**AngularJS est :**

- mieux adapté pour créer des applications Web côté client.

- est un langage idéal pour créer des applications Web hautement interactives.

Installation

**Pour installer Angular, cela nécessite node et npm**

pour installer NodeJS (choisissez la LTS) : <https://nodejs.org/en/download>

vérifier dans un terminal :

node --version

npm –version

Désinstaller angular : npm uninstall -g @angular/cli

Installer la dernière version d’Angular : **npm install -g @angular/cli**

Installer une version spécifique d’angular : npm install -g @angular/cli@13.3.11

Commandes Angular CLI

**ng version** Afficher la version installée

**ng help** Afficher les commandes de ng

**ng new** nomprojet Créer un nouveau projet Angular

(se mettre dans le repertoire souhaité):

add angular routine: yes

choisir SCSS

**ng serve** Commande dans le dossier projet, Construit et sert votre application

visible sur localhost:4200

**ng generate component** component-name Créé un nouveau composant

**ng directive | pipe | service | class | guard | interface | enum | module** | **test**

**ng build** Build le projet → dossier dist à la racine du projet

**package. json**

est véritablement le coeur de tout projet utilisant node. Il permet de définir :

- les metadatas d'un projet,

- d'installer toutes ses dépendances,

- d'exécuter les différents scripts npm.

"@angular/animations": "**~**13.3.0" ~ : permet d’indiquer de prendre la dernière version 13.x.x sans passer à la 14

**angular.json**

La configuration d’Angular CLI. Permet de définir plusieurs valeurs par défaut et de configurer également les fichiers inclus lors de la création de votre projet.

Directives Angular

**\*ngFor** permet de boucler sur un array et d'injecter les éléments dans le DOM.

**\*ngIf** permet de réaliser une condition sur une expression booléenne.

Code

**Une page « Accueil » contenant un titre, une image et un paragraphe qui apparaît/disparaît à l’appui d’un bouton « + ».**

app.component.html

<app-accueil-paragraph></app-accueil-paragraph>  
<router-outlet></router-outlet>

xxxxxx.component.hmtl

<!-- titre et image de la page -->  
<h1>titre de ma page accueil</h1>  
<img src="assets/zen.jpg" width=30% height=30% alt="zen">  
<p></p>  
  
<!-- bouton + qui appelle la methode affichage sur clic -->  
<button type="button" class="btn" (click)="affichage();">+</button>  
<p></p>  
  
<!-- texte affiche suivant condition (true/false) -->  
<div *\*ngIf*="condition;then ***contenu***"></div>  
<ng-template #***contenu***>texte paragraphe</ng-template>

xxxxxx.component.ts

// declaration condition  
public condition: any;  
  
// methode affichage : alterne condition true/false  
affichage = () : void => {  
 this.condition = !this.condition;  
};

BootStrap

Installation :

- **npm install bootstrap**

- rajouter dans angular.json :

"styles": [  
 "node\_modules/bootstrap/dist/css/bootstrap.min.css"  
 ],  
 "scripts": [  
 "node\_modules/bootstrap/dist/js/bootstrap.bundle.min.js"

]  
 - rajouter dans styles.scss :

@import "~bootstrap/dist/css/bootstrap.css";

Accessibilité

L’accessibilité du web comprend tous les handicaps affectant l’accès au web,

en particulier le handicap :

- auditif

- cognitif

- neurologique

- physique

- de la parole

- visuel

L’accessibilité du web bénéficie également aux personnes sans handicap, comme par exemple :

- les personnes utilisant un téléphone mobile, une montre connectée, une télévision connectée, et autres périphériques ayant des petits écrans, différents modes de saisie, etc.

- les personnes âgées dont les capacités changent avec l’âge

- les personnes ayant « une limitation situationnelle » comme être en plein soleil

Bonnes pratiques de bases

**Alternative textuelle pour les images**

image d'un logo; balisage HTML img alt='Logo de la Web Accessibility Initiative'

**Saisie au clavier sans souris**

Un site web accessible ne dépend pas de la souris ; il rend toutes les fonctionnalités disponibles au clavier.

**Transcriptions de l’audio**

exemple de transcript

**Respecter les balises titre et les sous-titres des pages**

une hiérarchie sans interruption dans les balises titres,

**Prévoir un plan du site**

Un sitemap est un fichier qui contient des informations sur les pages, vidéos et autres fichiers présents sur votre site, et dans lequel vous indiquez les liens entre ces fichiers. La plupart des CMS proposent les outils nécessaires pour créer un sitemap ou utiliser un générateur de sitemap.

**Prévoir des formulaires qui indiquent un contrôle de saisie**

L’utilisateur malvoyant doit pourvoir être alerté si les champs d’un formulaire sont incorrects ou incomplets. Il doit aussi comprendre quels sont les champs obligatoires.

**Ajustez les couleurs et les contrastes de votre site**

Choisissez des couleurs suffisamment différentes pour que le texte soit facile à lire. La combinaison la plus simple et la plus efficace reste indubitablement une couleur sombre sur un fond clair.

**Créez des boutons clairs et assez grands**

Laissez de l’espace autour des boutons : ils seront plus faciles à cliquer et à lire. Évitez également de placer deux boutons importants l’un à côté de l’autre.

**Choisissez des polices lisibles**

Préférez les polices larges, avec ou sans sérif

Sites de validation ou d’audit :

[https://validator.w3.org/#validate\_by\_uri](https://validator.w3.org/" \l "validate_by_uri)

**Google Lighthouse**

Cette extension est comparable à l'outil d'IBM, mais elle prend aussi en compte les problématiques de performances du site. https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk?hl=fr

Gestion de l’internnalisation

**ng add @angular/localize** installe les dépendances nécessaires

Rajouter dans angular.json

"architect": {  
 "build": {  
 "builder": "@angular-devkit/build-angular:browser",  
 "options": {  
 "localize": ["fr-FR"],

"projects": {  
 "mboneAngular": {  
 "i18n": {  
 "sourceLocale": "fr-FR",  
 "locales": {  
 "en": "src/locale/messages.en.xlf"  
 }

Rajouter i18n dans toutes les balises souhaitées dans les .html

<div *\*ngIf*="!conditionConnexion && !okConnexion">  
 <button type="button" class="btn-primary" (click)="afficheFormulaire();" **i18n**>Se connecter</button>  
</div>

**ng extract-i18n** –outpath -path src/locale

cela créé un fichier .xlf avec toutes les balises html qui contiennent i18n

dupliquer ce fichier : le renommer en « en » et traduire tous les contenus

Rajouter dans angular.json

"serve": {  
 "builder": "@angular-devkit/build-angular:dev-server",  
 "configurations": {  
 "en": {  
 "browserTarget":"mboneAngular:build:en"  
 },

Rajouter dans package.json

"scripts": {  
 "start:en": "ng serve --configuration=en",

**ng serve –configuration=**en pour lancer le site en anglais

Pour traduire les fichiers .xlf, on peut utiliser une application dédiée, par exempe <https://poedit.net/>

**Une page avec 2 composants qui affiche une liste d’artistes (nom, image).**

**La liste est initiée et bouclée dans un composant : article-liste.components**

**Le nom et l’image sont affichés dans un composant fille : article-fiche.components**

article-liste.components.html

<h3>Liste des artistes</h3>  
<!-- boucle de balayage de la liste et envoi au composant enfant (artiste-fiche) -->  
<div *\*ngFor*="let ***artiste*** of artistes">  
 <app-artiste-fiche [artiste]="***artiste***"></app-artiste-fiche>  
</div>

article-liste.components.ts

export class ArtisteListeComponent implements OnInit {  
  
 // initialisation liste artistes  
 artistes : Artiste[] = [  
 {  
 nom : "Gauguin",  
 urlImage : "assets/gauguin.jpg"  
 },{  
 nom : "Monet",  
 urlImage : "assets/monet.jpg"  
 }, {  
 nom :"Van Gogh",  
 urlImage :"assets/vangogh.jpg"  
 }  
 ];

Artiste.ts

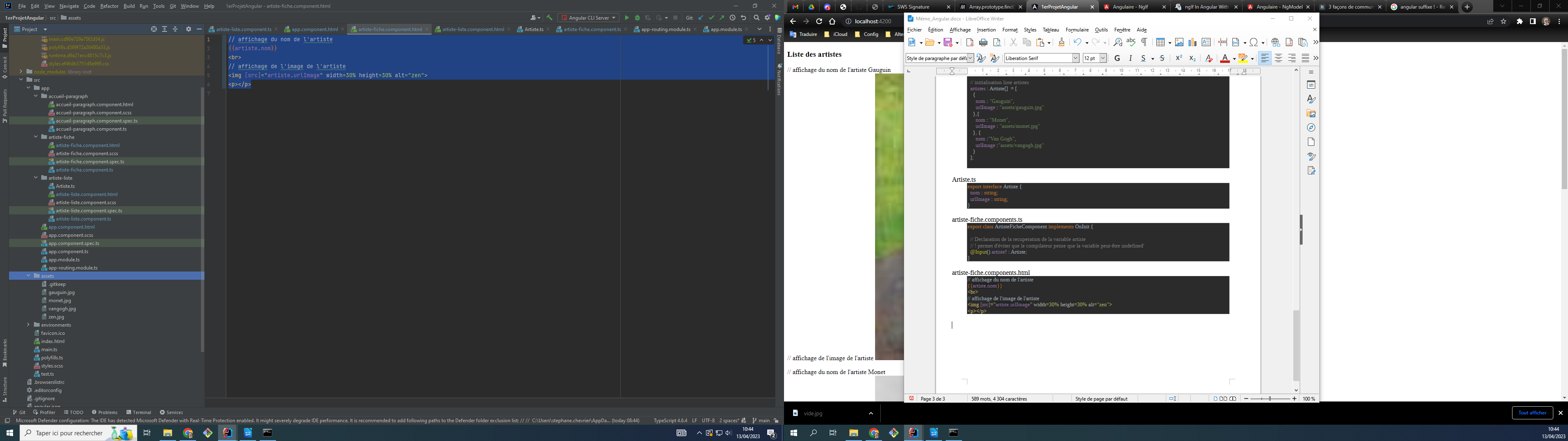
export interface Artiste {  
 nom : string;  
 urlImage : string;  
}

artiste-fiche.components.ts

export class ArtisteFicheComponent implements OnInit {  
  
 // Declaration de la recuperation de la variable artiste  
 // ! permet d'éviter que le compilateur pense que la variable peut-être undefined'  
 @Input() artiste! : Artiste;  
}

Attention dans ce cas il s’agit d’un objet. Si on modifie la valeur de l’objet, Angular ne va pas mettre à jour le composant enfant car la reference de l’objet de change pas. Ce probleme ne va pas se passer quand il s’agit d’une chaine ou d’un nombre.

artiste-fiche.components.html

<!-- affichage du nom de l'artiste -->  
{{artiste.nom}}  
<br>  
<!-- affichage de l'image de l'artiste -->  
<img [src]="artiste.urlImage" width=30% height=30% alt="zen">  
<p></p>

dossier assets (convention angular du dossier où sont stockés les fichiers)

**Une barre de navigation permettant de passer d’une page à l’autre grâce au routing Angular .**

App-routing.module.ts

import { AccueilParagraphComponent} from "./accueil-paragraph/accueil-paragraph.component";  
import { ArtisteListeComponent} from "./artiste-liste/artiste-liste.component";  
  
const routes: Routes = [  
 { path: 'Accueil', component: AccueilParagraphComponent},  
 { path: 'Artistes', component: ArtisteListeComponent}  
];

bar-nav.component.html

<ul class="nav navbar-nav">  
 <li>  
 <a routerLink="Accueil">Accueil</a>  
 </li>  
 <li>  
 <a routerLink="Artistes">Liste Artistes</a>  
 </li>  
</ul>

bar-nav.component.scss

\* {  
 font-family: 'Lucida Sans';  
}  
.nav{  
 display:flex;  
 justify-content: center;  
 background-color: antiquewhite;  
 padding: 20px;  
 margin: 0;  
 color: white;  
}  
li {  
 margin-right: 20px;  
 margin-left: 20px;  
 list-style: none;  
}

**Un bouton « supprimer » à côté de chaque artiste qui permet de supprimer l’artiste de la liste des artistes.**

Artiste-fiche.component.html

<!-- bouton + qui appelle la methode suppressionArtiste sur clic -->  
<button type="button" class="btn" (click)="suppressionArtiste()">supprimer cet artiste</button>

Artiste-fiche.component.ts

// Declaration de l'evenement qui permet d'envoyer l'Artiste du composant enfant (celui-ci) au parent  
@Output() artisteSup : ***EventEmitter***<Artiste> = new ***EventEmitter***<Artiste>();  
  
// Methode d'envoi de l'artiste au composant parent  
public suppressionArtiste() : void {  
 this.artisteSup.emit(this.artiste);  
}

Artiste-liste.component.html

<h3>Liste des artistes</h3>  
<!-- boucle de balayage de la liste et -->  
<div *\*ngFor*="let ***artiste*** of artistes">  
 <!-- envoi au composant enfant (artiste-fiche) -->  
 <!-- et recuperation de l'evenement envoye par l'enfant (artiste a suprimer) -->  
 <app-artiste-fiche [artiste]="***artiste***" (artisteSup)="artisteSup(***$event***)"></app-artiste-fiche>  
</div>

Artiste-liste.component.ts

// Methode de suppression de la liste d'artistes le l'artiste en argument  
artisteSup(artiste:Artiste) : void {  
 this.artistes = this.artistes.filter((artiste1 : Artiste): boolean => artiste1.nom != artiste.nom);  
}

**Le nouvel artiste est ajouté à la liste des artistes au clic d’un bouton « ajouter » en utilisant les reactive forms Angular.**

Artiste-form.component.html

<!-- formulaire de saisie -->  
<div>  
 Name: <input [formControl]="nomArtiste">  
 <label *\*ngIf*="nomArtiste.invalid" [ngClass] = "'error'"> Le nom requiert au moins 1 carractère </label>  
</div>  
  
<!-- bouton enregistrer la saisie -->  
<button type="button" class="btn" (click)="setArtiste()">enregistrer la saisie</button>  
<p></p>

Artiste-form.component.ts

nomArtiste : FormControl = new FormControl('', [Validators.*required*, Validators.*minLength*(1)]);  
  
// Declaration de l'evenement qui permet d'envoyer l'Artiste du composant enfant (celui-ci) au parent  
@Output() artistePlus : ***EventEmitter***<string> = new ***EventEmitter***<string>();  
  
setArtiste() : void {  
 ***console***.log(this.nomArtiste.value)  
 this.artistePlus.emit(this.nomArtiste.value);  
}

artiste-liste.component.html

<!-- affichage du formulaire pour rajouter un artiste -->  
<!-- texte affiche suivant condition (true/false) -->  
<div *\*ngIf*="conditionAjouterArtiste">  
 <app-artiste-form (artistePlus)="ArtistePlus(***$event***)"></app-artiste-form>  
</div>

artiste-liste.component.ts

// Methode pour afficher le formulaire quand on veut rajouter un artiste  
ajouterArtiste() : void {  
 this.conditionAjouterArtiste = !this.conditionAjouterArtiste;  
}  
  
// Methode pour ajouter un element a la liste Artistes  
ArtistePlus(artisteNom : string) : void {  
 // Efface le formulaire  
 this.conditionAjouterArtiste = false;  
 // Rajoute l'element artiste nom  
 let artisteNew : Artiste = {nom:artisteNom,urlImage:""} ;  
 this.artistes.push(artisteNew);  
 ***console***.log(this.artistes)  
}

**Gestion des requetes à une API**

Cela nécessite 2 blocs :

- un component.ts créé avec **ng generate component**

- un service.ts créé avec **ng generate service**

component.ts

import { Component, OnInit } from '@angular/core';  
import {AuthentificationService} from "../service/authentification.service";  
import { User } from "../service/authentification.service";  
  
@Component({  
 selector: 'app-login',  
 templateUrl: './login.component.html',  
 styleUrls: ['./login.component.scss']  
})  
export class LoginComponent implements OnInit {  
  
 // création des propriétés qui seront récupérées du service   
 isConnected!: boolean;  
 user!: User;  
  
 // Créé un lien avec le service  
 constructor(private authenticationService: AuthentificationService) {  
   
 // récupération de l'état de la connexion true/false  
 this.authenticationService.isConnectedObservable  
 .subscribe((value: boolean) => {  
 this.isConnected = value;  
 })  
   
 // récupération du user correspndant au tokken  
 this.authenticationService.userObservable  
 .subscribe(pUser => {  
 this.user = pUser;  
 })  
 }}

service.ts

import { Injectable } from '@angular/core';  
import { HttpClient, HttpHeaders } from "@angular/common/http";  
import { Observable, throwError } from 'rxjs';  
import {Expert, ExpertResearch} from "../expertslist/expertslist.component";  
  
@Injectable({  
 providedIn: 'root'})  
  
// Service  
export class ExpertsListService {  
 expertsList!: Expert[];  
  
 // utilisation de la Class httpClient quand on construit le service  
 constructor(private httpClient: HttpClient) { }  
  
 // récupération de la réponse à la requête et mise dans la liste d'Experts  
 public getList(): Observable<any> {  
 let header = new HttpHeaders().set('Authorization', this.getTokenFromStorage());  
 return this.httpClient.get<Expert[]>('http://localhost:8084/get/listexperts', {headers: header})  
 }  
  
 // recuperation du token stocké  
 getTokenFromStorage(): string {  
 let token = ***localStorage***.getItem('token');  
 return token != null ? token : '';  
 }  
}

**Récupération du tokken**

Cela nécessite 2 blocs :

- un component.ts créé avec **ng generate component**

- un service.ts créé avec **ng generate service**

Un observable est un outils qui permet d’échanger des données ente 2 component.ts

component.ts

import { Component, OnInit } from '@angular/core';  
import {AuthentificationService} from "../service/authentification.service";  
import { User } from "../service/authentification.service";  
  
@Component({  
 selector: 'app-login',  
 templateUrl: './login.component.html',  
 styleUrls: ['./login.component.scss']  
})  
export class LoginComponent implements OnInit {  
  
 // création des propriétés qui seront récupérées du service  
 isConnected!: boolean;  
 user!: User;  
  
 // Créé un lien avec le service  
 constructor(private authenticationService: AuthentificationService) {  
  
 // récupération de l'état de la connexion true/false  
 this.authenticationService.isConnectedObservable  
 .subscribe((value: boolean) => {  
 this.isConnected = value;  
 })  
  
 // récupération du user correspndant au tokken  
 this.authenticationService.userObservable  
 .subscribe(pUser => {  
 this.user = pUser;  
 })  
 }}

service.ts

import { Injectable } from '@angular/core';  
import {HttpClient, HttpResponse} from "@angular/common/http";  
import {BehaviorSubject, firstValueFrom, map, Observable} from "rxjs";  
import {Router} from "@angular/router";  
  
export interface User {  
 // propriété user  
}  
  
@Injectable({  
 providedIn: 'root'  
})  
  
export class AuthentificationService {  
 user: User = {  
 // initialisation à null des propriétés du User  
 }  
 // chemin de la requete  
 private readonly URL: string = 'http://localhost:8084/post/connexion'  
  
 // Création des observables  
 \_user = new BehaviorSubject<User>(this.user);  
 private \_isConnectedSubject = new BehaviorSubject<boolean>(false);  
 token!: string;  
  
 // utilisation de la Class httpClient quand on construit le service  
 constructor(private http: HttpClient, private router: Router) {  
 }  
  
 // Méthode de lecture de l'état de la connection  
 get isConnectedObservable() {  
 return this.\_isConnectedSubject.asObservable();  
 }  
  
 // Méthode de lecture du user connecté  
 get userObservable() {  
 return this.\_user.asObservable();  
 }  
  
 // Méthode pour se connecter à l'API avec lecture et stockage du tokken  
 async attemptConnexion(email: string, password: string): Promise<any> {  
 await this.sendRequest(email, password).subscribe((response: HttpResponse<any>) => {  
 this.\_user.next(response.body);  
 ***localStorage***.setItem('token', response.headers.get("Authorization") || '');  
 this.router.navigate(['/home']);  
 });  
 }  
  
 // Méthode d'envoi d'une requete de connexion à l'API et de renvoi de la réponse (header et body)  
 sendRequest(email: string, password: string) {  
 return this.http.post<any>(  
 this.URL, {  
 username: email,  
 password: password,  
 },  
 {  
 observe: 'response'  
 }  
 )  
 }}

**Tests**

Il est nécessaire de bien dissocier les tests unitaires des tests fonctionnels, par exemple E2E.

Par exemple pour tester le changement d’une valeur par une fonction, le test unitaire est parfait. Mais pour tester l’affichage d’un component en cliquant sur un bouton, il est préférable de le faire dans un test fonctionnel et pas unitaire.

La création de test unitaire de component avec une injection de dépendance (paramètre dans le constructeur qui utilise un autre component.ts) demande une logique particulière pas vu en formation.

La création de test utilisant plusieurs components n’a pas été vu non plus.

J’ai juste appri à créer des tests unitaires d’un component HTML et d’un component JS.

Exemples dans fichier **xxx.component.spec.ts**

Test unitaire d’un component.html

// test sur component HTML : affichage d'un texte  
// declaration  
it('should mdp affiche', () => {  
 fixture.detectChanges();  
 // creation d'un objet de recuperation d'un element HTML  
 const compiled = fixture.nativeElement as HTMLElement;  
 // on vérifie que la balise dont l'id est labelpourtest contient le texte voulu  
 expect(compiled.querySelector('.labelpourtest')?.textContent).toContain('mot de passe :');  
});

Test unitaire d’un component JS

// test sur component JS  
// declaration  
it('should emit false annulerdemande', () => {  
 // creation d'un objet de surveillance sur annulerdemane et la methode emit  
 let spy = spyOn(component.annulerDemande, 'emit');  
 // on execute la methode annuler  
 component.annuler();  
 // on verifie que l'event emitter a emit false  
 expect(spy).toHaveBeenCalledWith(false);  
})