Mémento

Angular

Version 1.0 (créé le 10/12/2024, modifié le 10/12/2024)

Angular est un Framework TypeScript open source développé et maintenu par l'entreprise Google. Il est orienté Composants et permet de faire des sites webs de type Single Page Application, Il intègre la possibilité d'utiliser un routeur, un gestionnaire d'états centralisé, des tests... sans avoir à gérer des librairies supplémentaires.

Loric Informatique

Table des matières

1. Prise en main

1.1. Outils nécessaires

- Un logiciel de codage (ex : Visual Studio Code avec l'extension Angular Language Service et l'option Auto Save activée)
- Un navigateur internet avec extension Angular DevTools
- NodeJS (LTS suffit)
- Angular CLI (version 18 ou supérieure)
- Connaissances en HTML, CSS et TypeScript

Le mémento HTML est disponible en ligne sur le site : https://loricaudin.github.io/loric-informatique/mementos/html/memento_html.html

Le mémento CSS est disponible en ligne sur le site : https://loricaudin.github.io/loric-informatique/mementos/css/memento_css.html

Le mémento TS est disponible en ligne sur le site : https://loricaudin.github.io/loric-informatique/mementos/js/memento_ts.html

1.2. Installer Angular

Dans le dossier où se trouvera le projet Angular, entrer la commande : `npm install -g @angular/cli@18.2.2`

Puis dans le dossier du projet :

```
npm install
```

Si les prochaines commandes ne fonctionnent pas ou sont introuvables, essayez d'ajouter « `npx` » devant chaque commande « `ng` ».

1.3. Créer un projet Angular

Commande : `ng new monPremierProjet --skip-tests --defaults`

1.4. Démarrer le serveur Angular

Commande : `ng serve`

2. Bases du Framework

2.1. Les fichiers de démarrage du Framework

2.1.1. Le fichier main.ts

Ce fichier permet de charger le composant racine de l'application.

Syntaxe :

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
    .catch((err) => console.error(err));
```

2.1.2. Le fichier index.html

Ce fichier est celui appelé dans l'application et pour tous les chemins.

Syntaxe :

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon titre</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <link
href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;5
00&display=swap" rel="stylesheet">
    <link
href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
</head>
<body>
    <app-root></app-root>
```

```
</body>
</html>
```

Ici, le fichier app.component.ts ainsi que le fichier app.component.html sont appelés via la classe `AppComponent` du fichier main.ts, et via la balise `app-root` du fichier index.html (puisque le sélecteur vaut `app-root` dans app.component.ts)

2.2. Les composants

Commande de création d'un composant :

```
ng generate component components/MonComposant
```

ou

```
ng g c MonComposant
```

2.2.1. Syntaxe

```
import {Component} from '@angular/core';
import {RouterOutlet} from '@angular/router';

@Component({
    selector: 'app-mon-composant',
    standalone: true,
    imports: [RouterOutlet, …],
    templateUrl: './app.component.html',
    styleUrl: './app.component.css'
})
export class MonComposant {
    …
}
```

2.2.2. Envoi des données du composant parent vers le composant fils

2.2.2.1. Récupérer des valeurs provenant du composant parent (côté script)

Dans la classe du composant fils, écrire les variables sous la forme :

```
@Input() attribut: type = valeurParDefaut;
```

ou (pour forcer le passage d'une valeur dans l'attribut) :

```
@Input({required: true}) attribut!: type;
```

2.2.2.2. Écrire des valeurs dans le composant fils (côté HTML)

```
<app-composant-fils attribut="valeur"/>
```

2.2.3. Envoi des données du composant fils vers le composant parent

2.2.3.1. Émettre un événement au composant parent (côté script)

Dans la classe du composant fils, écrire les variables sous la forme :

```
@Output() monEvenement = new EventEmitter<void>();
```

ou

```
@Output() monEvenement = new EventEmitter<type>();
```

Pour émettre l'événement :

```
this.monEvenement.emit()
```

ou

```
this.monEvenement.emit(valeur);
```

2.2.3.2. Écouter un événement provenant du composant fils (côté HTML)

```
<app-composant-fils (monEvenement)="onMaFonction()" />
```

ou

```
<app-composant-fils (monEvenement)="onMaFonction($event)" />
```

2.3. Les classes

Commande de création d'une classe :

```
ng generate class class/MaClasse
```

2.4. Les interfaces

Commande de création d'une interface :

```
ng generate interface models/MonModele
```

2.4.1. Interfaces prédéfinis pour les composants

Interface	Exécution	Méthode à implémenter
`OnInit`	À l'initialisation du composant	`ngOnInit`
`OnDestroy`	À la destruction du composant	`ngOnDestroy`
`OnChanges`	Au changement d'une valeur de l'input	`ngOnChanges`
`AfterViewInit`	À fin de l'initialisation de la vue	`ngAfterViewInit`

2.5. Les énumérations

Commande de création d'une énumération :

```
ng generate enum enum/MonEnumeration
```

2.6. Les services (connexion à une API REST)

Un service est une classe qui permet de fournir des données aux composants et de faire des interactions avec des APIs.

Toutes les méthodes publiques de cette classe sont utilisables partout dans l'application.

Commande de création d'un service (pas besoin de mettre Service à la fin du nom) :

```
ng generate service services/monService
```

ou

```
ng g s services/monService
```

2.6.1. Syntaxe

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

@Injectable({
    providedIn: 'root'
})
export class MonService {

    constructor(private variable1: type1, private variable2: type2…)
{
        …
    }
}
```

2.6.2. Paramètre du service (dans @Injectable)

Paramètre	Description
`providedIn: 'root'`	Ajouter le service au registre des composants injectables n'importe où dans l'application (le service démarrera automatiquement quand il est appelé, s'il n'est pas renseigné, il faudra ajouter la

	classe dans le tableau des providers dans app.config.ts)

2.6.3. Les observers et observables

Importations : `import {Observable, Subscription} from 'rxjs';`

2.6.3.1. Les observables

2.6.3.1.1. Créer un observable

```
monObservable$ = new Observable<type>((observer) => {
    instructions;
});
```

Par convention, le nom de l'observable doit toujours se terminer par $

2.6.3.1.2. Émettre une notification aux observers abonnés

Instruction	Description
`observer.next(maVariable);`	Émettre une nouvelle valeur (peut se répéter plusieurs fois)
`observer.error(monErreur);`	Émettre une erreur
`observer.complete();`	Arrêter l'émission de nouvelles valeurs

Les instructions peuvent s'enchaîner et même se répéter plusieurs fois.

2.6.3.2. Les observers

2.6.3.2.1. Créer un observer

```
monObserver = {
    next: (value: string) => {
        instructions;
    },
    error: (error: Error) => {
        instructions;
    },
    complete: () => {
        instructions;
    }
};
```

Les événements `error` et `complete` sont facultatifs. Si seul le next est utilisé, il est possible de simplifier l'écriture de l'observer avec :
`monObserver = (value: string) => instructions;`

2.6.3.2.1. Abonner un observer à un observable

```
maSouscription: Subscription =
this.monObservable$.subscribe(monObserver);
```

Il est conseillé de faire cette instruction dans une méthode `ngOnInit` afin de l'exécuter à l'initialisation du composant (nécessite d'implémenter `OnInit`)

2.6.3.2.2. Désabonner un observer d'un observable

```
maSouscription.unsubscribe();
```

Il est conseillé de faire cette instruction dans une méthode `ngOnDestroy` afin de l'exécuter à la suppression du composant (nécessite d'implémenter `OnDestroy`)

2.6.3.3. Opérateurs de RxJS

Observable	Description
`interval(n);`	Émettre une nouvelle valeur incrémentée tous les *n* millisecondes (renvoie une valeur entière incrémentée à chaque appel de `next`)
`interval(n).pipe(map(x) => x * 10), filter(x) => x < 100);`	Émettre une nouvelle valeur incrémentée de 10 tous les *n* millisecondes, tant que *x* est inférieure à 100 (renvoie une valeur entière incrémentée à chaque appel de `next`)
`of(valeur1, valeur2…).pipe(map(x) => x * 10);`	Émettre une liste de valeurs et les transformer

`monObservable$.pipe(`
`  takeUntilDestroyed()`
`);`
	Permettre le désabonnement automatique quand le composant est détruit (l'abonnement pourra donc se faire dans le constructeur)

`monObservable$.pipe(`
`  takeUntilDestroyed(this.destroyRef)`
`);`
	Permettre le désabonnement automatique quand le composant est détruit (version avec abonnement dans `ngOnInit`, mais nécessitant d'écrire dans la classe du composant : `destroyRef = inject(DestroyRef)`)

`monObservable$.pipe(`
`  tap(console.log(valeur))`
`);`
	Exécuter des instructions (ici affichage dans la console) en plus d'émettre la valeur

2.6.4. Le service de connexion à une API

2.6.4.1. Enregistrer un provider au démarrage de l'application (pour utiliser le service HttpClient)

Dans le fichier app.config.ts, ajouter dans la liste des providers : `provideHttpClient()`

Importation :

```
import {provideHttpClient} from '@angular/common/http';
```

2.6.4.2. Injecter le service HttpClient

Syntaxe :

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

@Injectable({
    providedIn: 'root'
})
export class MonService {

    donnees?: MonModele;

    constructor(private http: HttpClient) {
        …
    }
}
```

Note : Pour renforcer la sécurité des données en empêchant leur modification depuis les composants, il est possible de restreindre la modification en ajoutant les lignes suivantes dans le service :

```
private monSujet = new BehaviorSudject<MonModele>(valeurParDefaut);
public currentDonnees = this.monSujet.asObservable();
```

Dans le service, il n'y aura juste à faire `this.monSujet.next(donneeRecue);` dans l'événement `next`, et à utiliser le pipe `async` sur *currentDonnees* dans le composant.

2.6.4.3. Échanger des données JSON avec une API

Instruction (renvoyant un observable)	Description
`this.http.get<MonModele>("urlAPI");`	Récupérer des données avec la méthode GET
`this.http.post<MonModele>("urlAPI", donnees);`	Envoyer des données avec la méthode POST
`this.http.head<MonModele>("urlAPI", donnees);`	Récupérer l'en-tête de la ressource avec la méthode HEAD
`this.http.put<MonModele>("urlAPI", donnees);`	Ajouter une donnée avec la méthode PUT
`this.http.patch<MonModele>("urlAPI", donnees);`	Modifier une donnée avec la méthode PATCH
`this.http.delete<MonModele>("urlAPI");`	Supprimer une donnée avec la méthode DELETE

2.7. Les pipes

Un pipe est un opérateur spécial dans les templates qui permet de formatter une donnée pour l'affichage sans la modifier.

Commande de création d'un pipe (pas besoin de mettre Pipe à la fin du nom) :

```
ng generate pipe pipes/monPipe
```

2.7.1. Syntaxe

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

@Pipe({
    name: 'monPipe',
    standalone: true
})
export class MonPipe implements PipeTransform {
```

```
    transform(value: string): string {
        instructions;
    }
}
```

2.7.2. Paramètres du pipe (dans @Pipe)

Paramètre	Description
`type: false;`	Permettre au pipe de s'actualiser automatiquement, ce qui n'est pas le cas par défaut si la valeur ne change pas

2.7.3. Pipes prédéfinis

Importation : `import {MaPipe} from '@angular/common';`

2.7.3.1. Les pipes pour les chaînes de caractères

Pipe	Appel du pipe dans le code HTML	Description
`UpperCasePipe`	`uppercase`	Mettre une chaîne de caractères en majuscule
`LowerCasePipe`	`lowercase`	Mettre une chaîne de caractères en minuscule
`TitleCasePipe`	`titlecase`	Mettre les premières lettres de chaque mot dans une chaîne de caractères en majuscule

2.7.3.2. Les pipes pour les nombres

Pipe à importer : `DecimalPipe`

Appel du pipe dans le code HTML :

`number : '{nbChiffresMinAvV}.{nbChiffresMinApV}-{nbChiffresMaxApV}'`

ou (pour écrire en français les nombres) :

`number : '{nbChiffresMinAvV}.{nbChiffresMinApV}-{nbChiffresMaxApV}' : 'fr'`

Paramètre	Description
nbChiffresMinAvV	Nombre minimal de chiffres entiers avant la virgule
nbChiffresMinApV	Nombre minimal de chiffres après la virgule
nbChiffresMaxApV	Nombre maximal de chiffres après la virgule

2.7.3.3. Les pipes pour les dates

Pipe à importer : `DatePipe`

Appel du pipe dans le code HTML : `date : 'masque1caractere1masque2…'`

Le caractère peut être inséré comme une chaîne de caractères dans la chaîne de caractère (également entre guillemets simples ou double)

Masque	Description
`'YYYY'` ou `'yyyy'` ou `'Y'` ou `'y'`	Année (ex : 2024)
`'w'`	Numéro de la semaine dans l'année
`'M'` ou `'L'`	Numéro du mois (ex : 5)
`'MM'` ou `'LL'`	Numéro du mois (ex : 05)
`'d'`	Numéro du jour dans le mois (ex : 8)
`'dd'`	Numéro du jour dans le mois (ex : 08)

`'h'`	Heure sur 12 heures (ex : 9)
`'hh'`	Heure sur 12 heures (ex : 09)
`'H'`	Heure sur 24 heures (ex : 9, 21)
`'HH'`	Heure sur 24 heures (ex : 09, 21)
`'m'`	Minutes (ex : 5)
`'mm'`	Minutes (ex : 05)
`'s'`	Secondes (ex : 3)
`'ss'`	Secondes (ex : 3)
`'MMMM'` ou `'LLLL'`	Mois en lettres
`'MMM'` ou `'LLL'`	Mois abrégé (3 lettres)
`'EEEE'` ou `'cccc'`	Jour en lettres
`'EEE'` ou `'E'` ou `'ccc'`	Jour abrégé (3 lettres)
`'a'` ou `'aa'`	AM ou PM
`'short'`	Date sous la forme : `'M/d/yy, h:mm a'`
`'medium'`	Date sous la forme : `'MMM d, y, h:mm:ss a'`
`'long'`	Date sous la forme : `'MMMM d, y, h:mm:ss a z'`
`'full'`	Date sous la forme : `'EEEE, MMMM d, y, h:mm:ss a zzzz'`
`'shortDate'`	Date sous la forme : `'M/d/yy'`
`'mediumDate'`	Date sous la forme : `'MMM d, y'`
`'longDate'`	Date sous la forme : `'MMMM d, y'`
`'fullDate'`	Date sous la forme : `'EEEE, MMMM d, y'`
`'shortTime'`	Date sous la forme : `'h:mm a'`
`'mediumTime'`	Date sous la forme : `'h:mm:ss a'`
`'longTime'`	Date sous la forme : `'h:mm:ss a z'`
`'fullTime'`	Date sous la forme : `'h:mm:ss a zzzz'`

2.7.4. Autres pipes

Pipe	**Appel du pipe dans le code HTML**	**Description**
`AsyncPipe`	`async`	Afficher les données d'un

		observable de manière asynchrone et gérer l'abonnement et le désabonnement à un observable

2.7.5. Utiliser un pipe dans le HTML

Dans la classe où le pipe peut être utilisé, ajouter l'utilisation du pipe avec :

`imports: [MonPipe]`

Et dans le code HTML, faire le binding de variable sous la forme :

`{{maVariable | appeldupipe}}`

Ou avec des paramètres :

`{{maVariable | appeldupipe : parametre1 : parametre2…}}`

2.8. Les routes

2.8.1. Créer un tableau de routes

Créer un fichier app.routes.ts et insérer un tableau de routes sous la forme :

```
import {Routes} from "@angular/router";
export const routes: Routes = [
    route1,
    route2…
]
```

Dans app.config.ts, `provideRouter(routes)` doit être ajouté dans la liste des providers pour utiliser les routes (avec `routes` correspondant au tableau de routes).

2.8.2. Associer des chemins à des composants avec des routes

Route simple :

```
{
    path: 'mon-chemin',
    component: MonComposant,
    title: 'Titre de l'onglet'
}
```

Route avec paramètres personnalisées :

```
{
    path: 'mon-chemin/:param-1',
    component: MonComposant,
    title: 'Titre de l'onglet'
}
```

2.8.3. Rediriger une route vers une autre route

```
{
    path: 'mon-chemin',
    redirectTo: 'chemin-a-rediriger',
    pathMatch: 'full'
}
```

Si `pathMatch` est égal à `full`, alors les autres chemins commençant par le chemin configuré ne seront pas traités dans ce cas.

2.8.4. Afficher le composant dans la page HTML selon la route actuelle

```
<router-outlet />
```

La directive RouterOutlet est à importer dans le tableau des imports du comosant qui l'utilise.

2.8.5. Créer un lien vers une autre route dans une page HTML

2.8.5.1. Navigation depuis une page HTML

```
<a routerLink="mon-chemin">contenu</a>
```

ou

```
<a [routerLink]="['partie1-du-chemin', 'partie2-du-chemin']">
```

La directive RouterLink est à importer dans le tableau des imports du comosant qui l'utilise.

2.8.5.2. Navigation manuelle depuis le code TypeScript

```
this.router.navigateByUrl("monChemin");
```

ou

```
this.router.navigate(["partie1-du-chemin", "partie2-du-chemin"]);
```

Pour l'utiliser, injectez dans le constructeur : `private router: Router`

2.8.6. Récupération du paramètre passé dans la route

2.8.6.1. Méthode simple

```
this.route.snapshot.params["param-1"];
```

Pour l'utiliser, injectez dans le constructeur : `private route: ActivatedRoute`

2.8.6.2. Méthode avec un observable

```
this.route.paramMap.subscribe((params : ParamMap) => {
    params.get("param-1") ;
    instructions;
}) ;
```

2.8.6.3. Méthode avec un input relié (pour Angular 16 ou supérieur)

```
@Input({required: true})
param-1!: string;
```

`withComponentInputBinding(routes)` doit être ajouté dans la liste des providers pour utiliser les routes (avec routes correspondant au tableau de routes).

2.9. Les formulaires

2.9.1. Contrôler un formulaire

Instruction	Description
`private fb = inject(NonNullableFormBuilder)`	Injecter une classe pour créer des FormControl et FormGroup.
`champFC = this.fb.control(valeurParDefaut);`	Ajouter un contrôleur sur un champ (la valeur par défaut n'est pas obligatoire)
`champFC = this.fb.control(valeurParDefaut, validateur);`	Ajouter un contrôleur sur un champ avec une vérification via un validateur
`champFC = this.fb.control(valeurParDefaut, [validateur1, validateur2…]);`	Ajouter un contrôleur sur un champ avec une vérification via plusieurs validateurs
`monGroupeFormulaire = this.fb.group({ champ1: champ1FC, champ2: champ2FC… });`	Créer un contrôleur de formulaires avec les champs à contrôler

2.9.2. Valider les champs d'un formulaire

2.9.2.1. Créer des validateurs

Validateur	Description
`Validators.required`	Valeur obligatoire
`Validators.minLength(`*`n`*`)`	Longueur minimale à respecter
`Validators.maxLength(`*`n`*`)`	Longueur maximale à respecter
`Validators.min(`*`n`*`)`	Valeur minimale à respecter
`Validators.max(`*`n`*`)`	Valeur maximale à respecter
`Validators.pattern(`*`regex`*`)`	Valeur devant respecter une expression régulière
`Validators.email`	Valeur devant être un email

2.9.2.2. Vérifier la validité du formulaire

Instruction	Description
`monGroupeFormulaire``.valid`	Vérifier que le formulaire est validé
`monGroupeFormulaire``.invalid`	Vérifier que le formulaire n'est pas validé
`monGroupeFormulaire``.errors`	Récupérer la liste des erreurs du formulaire
`monGroupeFormulaire``.value`	Récupérer toutes les valeurs du formulaire
`monGroupeFormulaire.champ1``.valid`	Vérifier que le formulaire est validé
`monGroupeFormulaire.champ1``.invalid`	Vérifier que le formulaire n'est pas validé
`monGroupeFormulaire.champ1``.errors`	Récupérer la liste des erreurs d'un champ du formulaire
`monGroupeFormulaire.champ1``.value`	Récupérer la valeur du champ
`monGF.champ1``.hasError("`*`nomErreur`*`")`	Vérifier si une condition du champ est respectée (ex : "minlength")

2.9.3. Créer un formulaire

Syntaxe :

```
<form (ngSubmit)="maFonction()" [formGroup]="monGroupeFormulaire">
    <input formControlName="champ1"/>
    <input formControlName="champ2"/>
    …
    <button type="submit"
[disabled]="monGroupeFormulaire.invalid">Valider</button>
</form>
```

3. Le Framework côté HTML

3.1. Le binding dans les balises

Instruction	Description
`[maPropriete]="maVariable"`	Faire un binding de propriétés (par exemple `class`, `href`…)
`(click)="onClickMaFonction()"`	Faire un binding d'événements
`[class.maClasse]="condition"`	Faire un binding de classes (si *condition* est vraie, alors la classe est appliquée)
`[class]=["maClasse1", "maClasse2"]`	Faire un binding de classes sous forme de liste (facultatif)
`[class]={"maClasse1": condition1, "maClasse2": condition2}`	Faire un binding de classes sous forme d'objet littéral (si *condition* est vraie, alors la classe est appliquée)

3.2. Conditions

Une condition renvoie `true` si elle est respectée et `false` sinon.

3.2.1. Opérateurs de comparaison

Condition	Description de ce que vérifie la condition
`a == b`	*a* égal à *b*
`a < b`	*a* strictement inférieur à *b*
`a > b`	*a* strictement supérieur à *b*
`a <= b`	*a* supérieur ou égal à *b*
`a != b`	*a* n'est pas égal à *b*
`a == null`	Tester si une variable est nulle

`||`	À mettre entre deux conditions, permet d'avoir une des deux conditions qui doit être vraie
`&&`	À mettre entre deux conditions, permet d'avoir deux conditions qui doivent être vraie
`!condition`	Ne doit pas respecter la condition

3.2.2. Tests de conditions

Instruction	Description

`@if (condition1) {`
`  <balise1>contenu1</balise1>`
`}`
	Si *condition1* est vraie, alors on affiche *contenu1*

`@if (condition1) {`
`  <balise1>contenu1</balise1>`
`} @else {`
`  <balise2>contenu2</balise2>`
`}`
	Si *condition1* est vraie, alors on affiche *contenu1*, sinon, on affiche *contenu2*

`@if (condition1) {`
`  <balise1>contenu1</balise1>`
`} @else if (condition1) {`
`  <balise2>contenu2</balise2>`
`} @else {`
`  <balise3>contenu3</balise3>`
`}`
	Si *condition1* est vraie, alors on affiche *contenu1*, sinon, si *condition2* est vraie, on affiche *contenu2*, sinon, on affiche *contenu3*

3.3. Boucles

3.3.1. Les différentes boucles

Instruction	Description

`@for (elt of tableau; track item.id) {`
`  <balise1>contenu1</balise1>`
`}`
	On parcourt le tableau pour *elt* prenant toutes les valeurs du tableau

`@for (elt of tableau; track elt.id) {`
`  <balise1>contenu1</balise1>`
`} @empty {`
	On parcourt le tableau pour *elt* prenant toutes les valeurs du tableau, ou on affiche *contenu2* si

`<balise2>contenu2</balise2>` `}`	le tableau est vide (`elt.id` peut être remplacé par `$index`)

3.3.2. Les variables contextuelles de la boucle for

Variable	Description
`$count`	Nombre d'items
`$index`	Indice de l'item
`$first`	Si c'est le premier item
`$last`	Si c'est le dernier item
`$odd`	Si l'item a un indice impair
`$even`	Si l'item a un indice pair

L'appel par interpolation fonctionne également avec les variables contextuelles.

3.4. Les instructions de base

Instruction	Description
`{{maVariableTS}}`	Appeler une variable TypeScript (interpolation)

4. Le Framework côté CSS

4.1. Appliquer manuellement du style à une balise

Syntaxe :

```
<balise [ngStyle]="{attribut1: valeur1; attribut2: valeur2;…}"></balise>
```

4.2. Angular Material

Angular Material permet d'implémenter des spécifications du Material Design de Google. Celui-ci est un ensemble de règles de design pour l'interface graphique des applications web et mobiles, ayant pour but d'améliorer l'expérience utilisateur.

Plus d'info sur ce site : https://material.angular.io/

5. Les fichiers de configurations

5.1. Le fichier angular.json

Champ	Description

`"monProjet": {`
`  "i18n": {`
`  "sourceLocale": "fr"`
`  }`
`}`
	Configurer le site en français