Chapitre 1 : Introduction à Angular

1. Définition d'Angular

Définition simple

Angular est un **framework frontend** basé sur **TypeScript** qui permet de créer des applications web **dynamiques, rapides et bien structurées**.

Il est développé et maintenu par Google.

Sans Angular :

- On code en HTML + CSS + JavaScript
- Le code devient vite désorganisé

Avec Angular :

- Le code est structuré
- On découpe l'application en **composants**
- On peut gérer des applications complexes

Les concepts clés d'Angular

- **Composants** → blocs d'interface
- **Templates** → HTML dynamique
- **Services** → logique métier / données
- **Routing** → navigation

Angular permet de :

- structurer le projet
- séparer les responsabilités
- faciliter la maintenance

2. Différence avec React et Vue.js

Comparaison simplifiée

Critère	**Angular**	**React**	**Vue.js**
Type	Framework complet	Librairie	Framework léger
Structure	Très structurée	Flexible	Simple
Langage	TypeScript	JavaScript	JavaScript
Usage	Applications complexes	UI dynamiques	Projets simples/moyens

Angular et React sont souvent utilisés pour créer des applications web modernes, mais leur philosophie est différente. Angular est un framework complet, ce qui signifie qu'il fournit directement la majorité des outils nécessaires au développement d'une application professionnelle. Par exemple, Angular possède par défaut un système de routing pour la navigation entre pages, un module de formulaires (FormsModule, ReactiveFormsModule), un client HTTP (HttpClient) pour communiquer avec les API, l'injection de dépendances, un système de tests, ainsi qu'une architecture claire basée sur les composants et les services. Le développeur travaille donc dans un environnement déjà structuré et organisé.

À l'inverse, React est principalement une librairie dédiée à la création d'interfaces utilisateur. React ne fournit pas directement certaines fonctionnalités essentielles : pour le routing, il faut installer react-router-dom, pour la gestion globale des données on ajoute souvent Redux ou Zustand, pour les requêtes HTTP on utilise souvent Axios ou Fetch API, et pour les formulaires on peut ajouter React Hook Form ou Formik. React offre donc beaucoup plus de liberté, mais cela signifie aussi que le développeur doit choisir lui-même les outils et l'architecture du projet.

En résumé, Angular propose un environnement complet "tout-en-un" prêt à l'emploi avec des fonctionnalités intégrées par défaut, tandis que React fournit principalement la partie interface utilisateur et nécessite l'ajout de plusieurs librairies externes pour construire une application complète.

Angular est souvent utilisé dans :

- grandes entreprises
- projets long terme
- applications complexes

3. Architecture SPA (Single Page Application)

Définition

Une **SPA (Single Page Application)** est une application web qui :
- charge une seule page HTML
- met à jour le contenu sans recharger la page

Comparaison

Site classique (MPA)

- Chaque clic recharge la page
- Lent
- Mauvaise expérience utilisateur

SPA (Angular)

- Pas de rechargement
- Navigation fluide
- Meilleure performance

Exemple réel

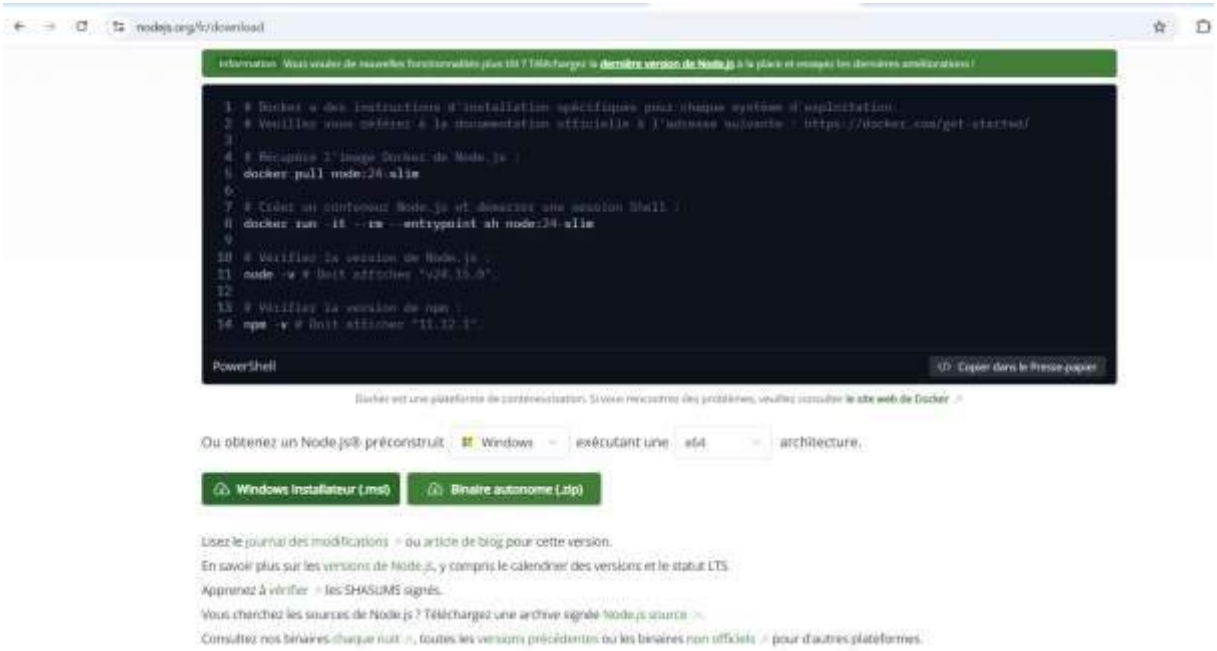
- Gmail
- Facebook
- Applications web modernes

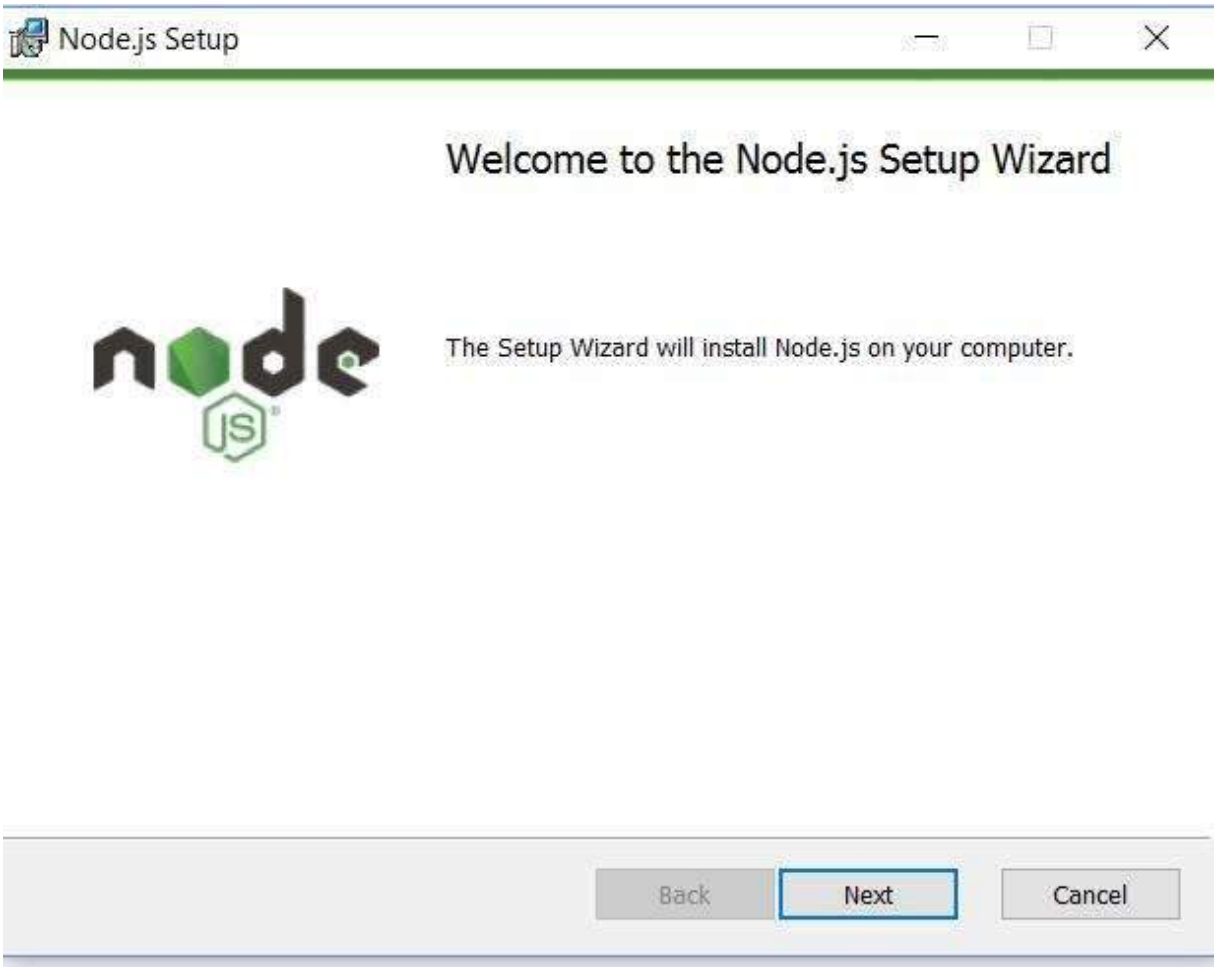
Chapitre 2 : Installation & Environnement

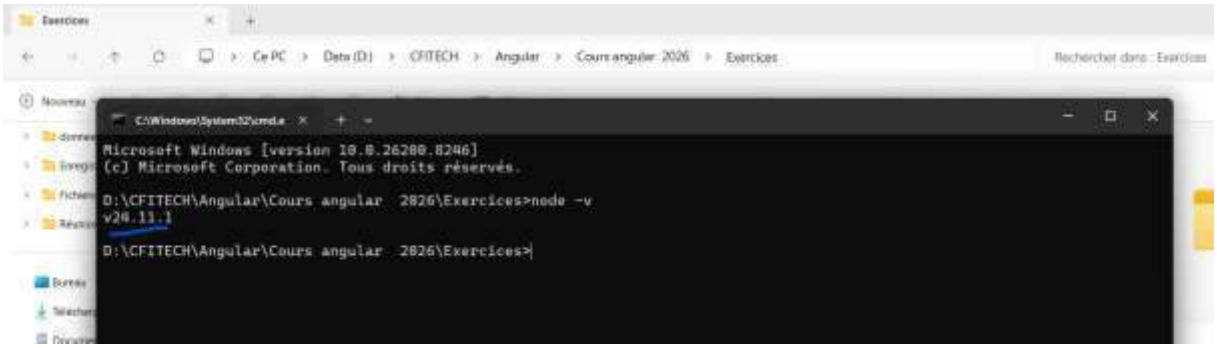
Objectifs

- Installer correctement l'environnement Angular
- Créer un projet Angular moderne (standalone)
- Lancer et comprendre une application Angular
- Naviguer dans la structure générée

1. Installation de Node.js







Angular utilise Node.js pour :

- exécuter des commandes
- installer des packages
- lancer le serveur de développement

Installation

1. Aller sur le site officiel
2. Télécharger la version **LTS (recommandée)**
3. Installer comme un logiciel classique

Vérification (IMPORTANT)

Dans le terminal :

node -v

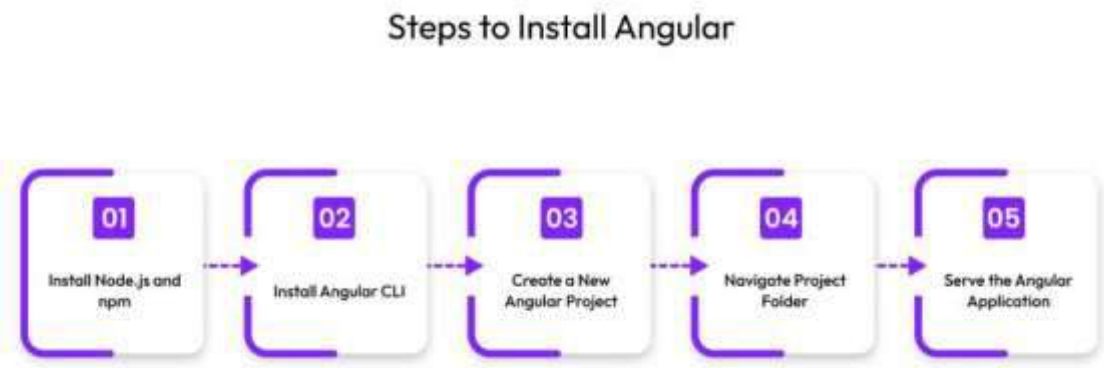
npm -v

Si des versions s'affichent → installation OK

Erreurs fréquentes

- Node non reconnu → problème PATH
- Version trop ancienne → mettre à jour

2. Installation de Angular CLI



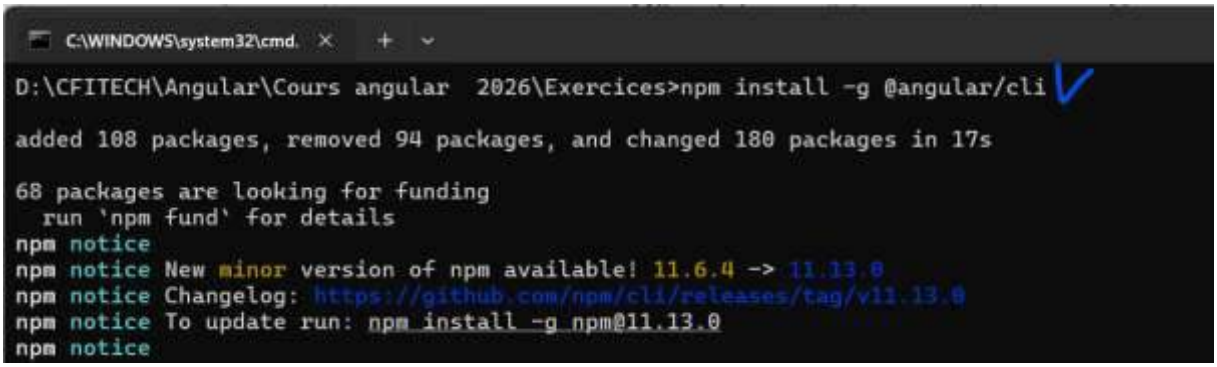
Angular CLI , c'est:

Un outil qui permet de :

- créer des projets
- générer des composants , ...
- lancer l'application

Installation

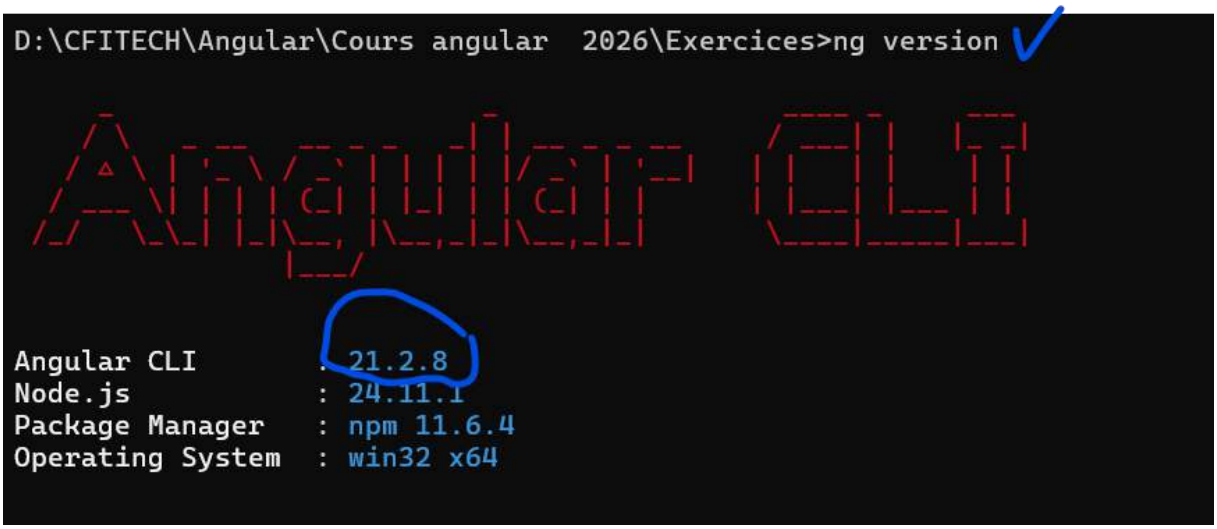
npm install -g @angular/cli



Vérification

ng version

Doit afficher la version Angular



4. **Création d'un projet Angular (moderne)**

Commande

ng new cfitech-app

Questions pendant la création

- Routing ? → YES
- CSS ou SCSS ? → au choix

Définition simple

SSR = le serveur construit la page HTML avant de l'envoyer au navigateur

Fonctionnement

X **Sans SSR (classique)**

- Le navigateur reçoit une page vide
- Angular (JavaScript) construit la page
- Affichage après un délai

V **Avec SSR**

- Le serveur génère la page complète
- Le navigateur affiche immédiatement
- Angular prend ensuite le relais

Pourquoi utiliser SSR ?

- Affichage plus rapide
- Meilleur SEO (Google)
- Meilleure performance sur mobile

On utilise SSR par ex pour:

- Site public
- E-commerce
- Blog
- Landing page

x **Non**

- Application interne
- Dashboard
- Projet débutant

"SSR permet d'afficher la page immédiatement, au lieu de laisser le navigateur tout construire."

En Conclusion

SSR améliore :

- la vitesse
- la visibilité

MAIS :

- ce n'est pas obligatoire
- on l'utilise surtout en niveau avancé

Pour les débutants en Angular

v **Choix recommandé :**

None

Pour :

- Eviter la confusion
- apprendre **vraiment Angular**
- Pas de dépendance à l'IA
- Meilleure compréhension des bases

Si on veut utiliser un outil plus tard

Avis pro sur chaque option

GitHub Copilot

Le meilleur pour développeurs

- Complétion automatique
- Aide au code
- Très utilisé en entreprise

À introduire niveau intermédiaire/avancé

Cursor

Très puissant mais…

- Génère beaucoup de code
- Peut rendre les débutants passifs
- Pas idéal au début

Claude

Bon pour explications

- Compréhension
- Aide théorique

Mais pas intégré comme outil de dev principal

Gemini

Correct, mais moins utilisé en dev Angular pur

Windsurf / Agents.md(Markdown)

Trop avancé
Pas utile au début

Angular crée maintenant automatiquement :
un projet **standalone (sans modules)**

Entrer dans le projet

cd cfitech-app

```
D:\CFITECH\Angular\Cours angular  2026\Exercices>cd cfitech-app

D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech-app>
```

4. Lancement du serveur Angular

Commande

ng serve

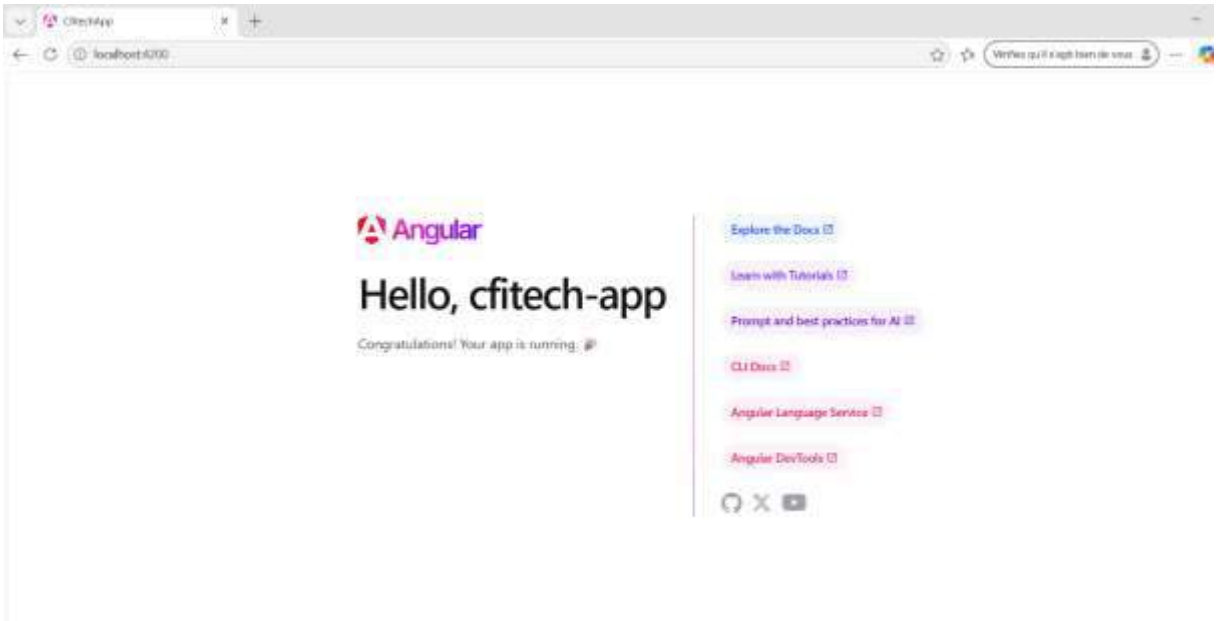
ng serve -o (open)

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech-app>ng serve -o

Would you like to share pseudonymous usage data about this project with the Angular Team
at Google under Google's Privacy Policy at https://policies.google.com/privacy. For more
details and how to change this setting, see https://angular.dev/cli/analytics.

   Yes

Thank you for sharing pseudonymous usage data. Should you change your mind, the following
command will disable this feature entirely:

    ng analytics disable

Global setting: enabled
Local setting: enabled
Effective status: enabled
Initial chunk files | Names         | Raw size
main.js             | main          | 48.50 kB |
styles.css          | styles        | 95 bytes |

                    | Initial total | 48.59 kB

Application bundle generation complete. [2.031 seconds] - 2026-04-29T13:53:52.058Z

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
  ➜  Local:   http://localhost:4200/
  ➜  press h + enter to show help
```

Ouvrir dans le navigateur :

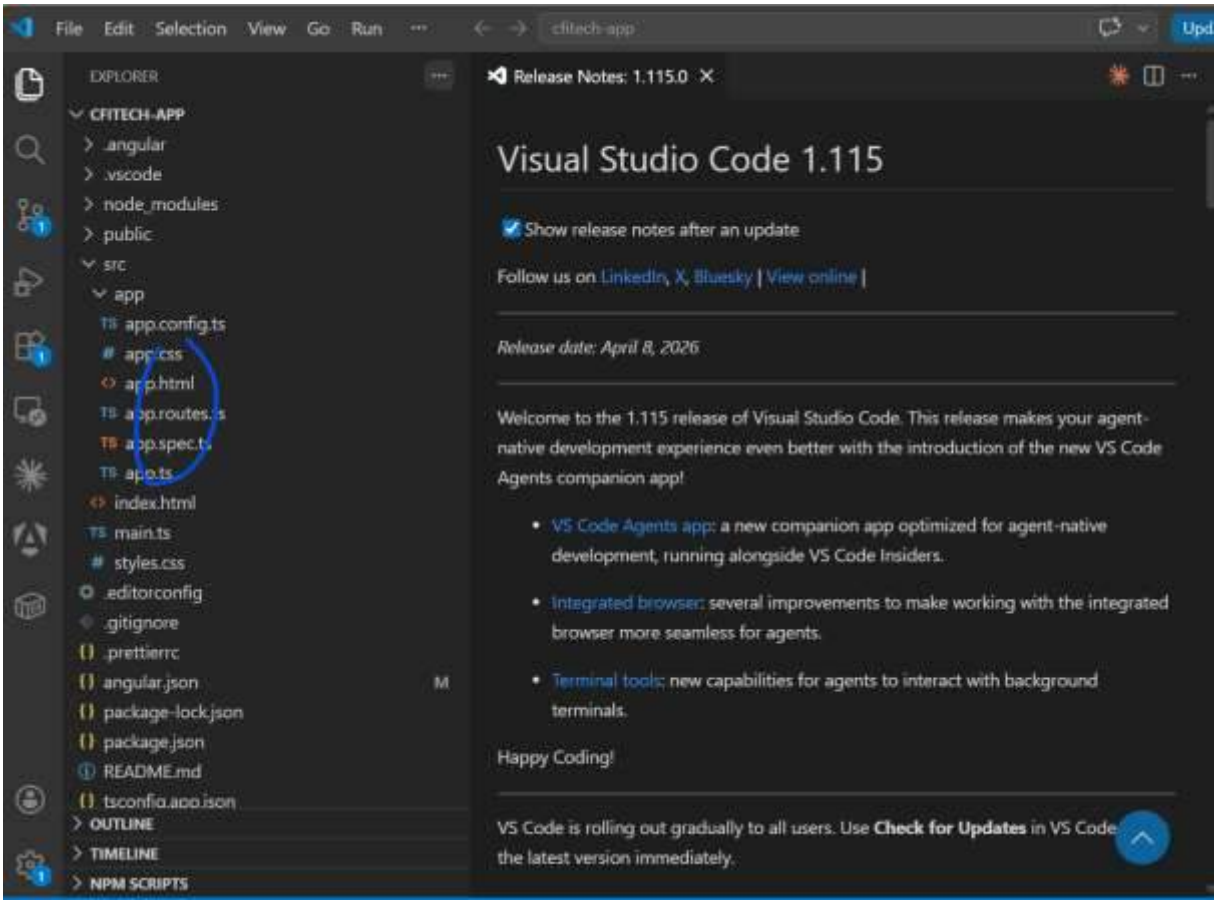
http://localhost:4200

L'application Angular s'affiche



5. Structure moderne du projet Angular

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech-app>code .

D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech-app>
```



Dossiers importants

src/

Contient toute l'application

src/app/

cœur de l'application

- app.ts → logique
- app.html → interface
- app.css → style

main.ts

point d'entrée de l'application

index.html

page principale chargée

Important (Angular moderne)

- Pas de NgModule
- Utilisation de **standalone components**

En resume :

Dans une application Angular, la relation entre index.html, main.ts et le composant (comme app) fonctionne comme une chaîne bien organisée. Le navigateur commence toujours par charger index.html, qui est la seule vraie page HTML du projet. À l'intérieur, on trouve une balise personnalisée comme <app-root>, qui sert simplement de point d'ancrage vide(veut dire un emplacement réservé, une sorte de "conteneur" vide, qui servira à afficher du contenu plus tard) . Ensuite, main.ts entre en jeu : c'est lui qui démarre Angular grâce à la fonction bootstrapApplication, en indiquant quel composant principal utiliser. Une fois Angular lancé, il regarde le **selector** défini dans ce composant (par exemple selector: 'app-root') et fait la correspondance avec la balise <app-root> présente dans index.html. À ce moment-là, Angular injecte automatiquement le contenu du fichier HTML du composant (app.html) à l'intérieur de cette balise. Résultat : même si index.html est la page de départ, c'est le contenu du composant qui est réellement affiché à l'écran.

6. Exercice pratique

Exercice : Créer votre premier projet Angular

Étapes

1. Installer Node.js
2. Installer Angular CLI
3. Créer un projet :

ng new mon-premier-projet

4. Lancer le projet :

ng serve -o

5. Ouvrir dans navigateur

Quel est le port par défaut : 4200

Mini défi

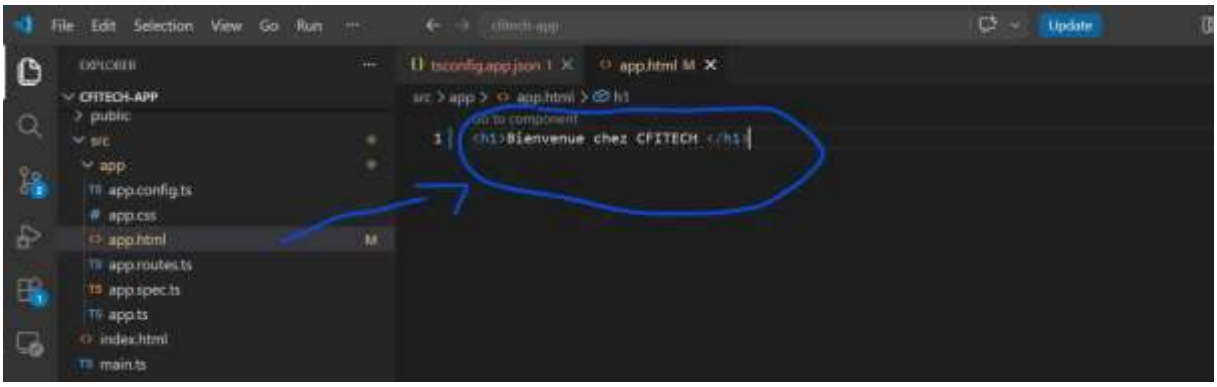
Modifier le fichier :

app.html

Remplacer le contenu par :

<h1>Bienvenue chez CFITECH </h1>

Rep :



Chapitre 3 : Structure d'un projet Angular

1. Comprendre la logique d'un projet Angular

Angular n'est pas un simple fichier HTML.

C'est une **application structurée** avec :

- plusieurs fichiers
- des composants
- une organisation claire

Idée clé

Angular découpe une application en **petites parties appelées composants**

Chaque partie a un rôle précis

Ex :

Application de gestion des stagiaires :

- page liste → composant
- formulaire → composant
- header → composant

Structure typique

src/app/

├── app.ts

├── app.html

├── app.css

Angular utilise une architecture :

LOGIQUE (TS)

+

INTERFACE (HTML)

+

STYLE (CSS)

= un composant

Détail des fichiers

app.ts (le cerveau)

Contient :

- variables
- fonctions
- logique ,...

```
export class App {
 title = "CFITECH";
}
```

app.html (l'affichage)

Ce que voit l'utilisateur

```
<h1>{{ title }}</h1>
```

app.css (le style)

Design :

```
h1 {
 color: blue;
}
```

3. main.ts (le démarrage)

Code réel

```
bootstrapApplication(App, appConfig);
```

"Angular, démarre l'application avec ce composant"

Sans main.ts :

x Angular ne démarre pas

x rien ne s'affiche

4. Standalone Components (Angular moderne)

x Avant

Angular utilisait :

- NgModule (complexe)

v Maintenant (Angular 21)

On utilise :
- standalone components

Ex

```
@Component({
 selector: 'app-root',
 standalone: true,
})
```

c'est :

- plus simple
- moins de fichiers
- plus rapide à apprendre

5. Organisation feature-based

Organiser le projet par **fonctionnalité**

Ex :

```
src/app/
  ├── stagiaires/
  │    ├── stagiaires.ts
  │    ├── stagiaires.ts
  │
  ├── formateurs/
```

On fait ça , Parce que dans un vrai projet :

- beaucoup de fichiers
- beaucoup de logique

Donc on organise

Avantages

- clair
- Professionnel
- Scalable(adaptable/évolutif/augmentation de performance)

6. Les fichiers importants (à connaître absolument)

index.html

page principale
contient :

<app-root></app-root>

main.ts

démarre Angular

app

premier composant affiché

package.json

dépendances

angular.json

configuration

Exercices

Étapes

1. Ouvrir app.ts
2. Modifier :

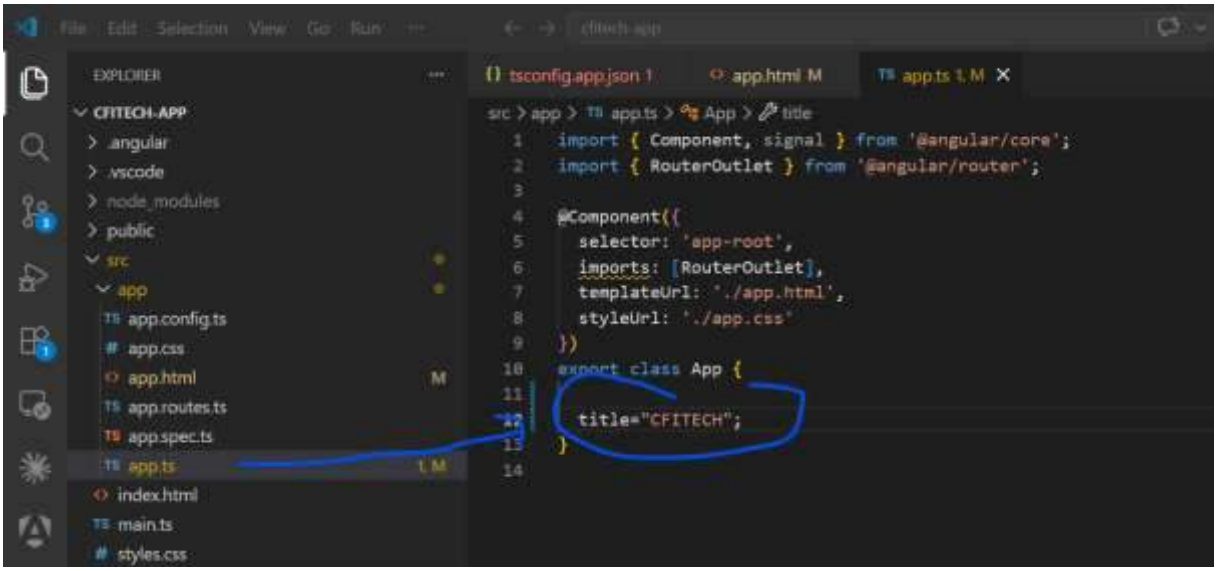
title = "CFITECH";

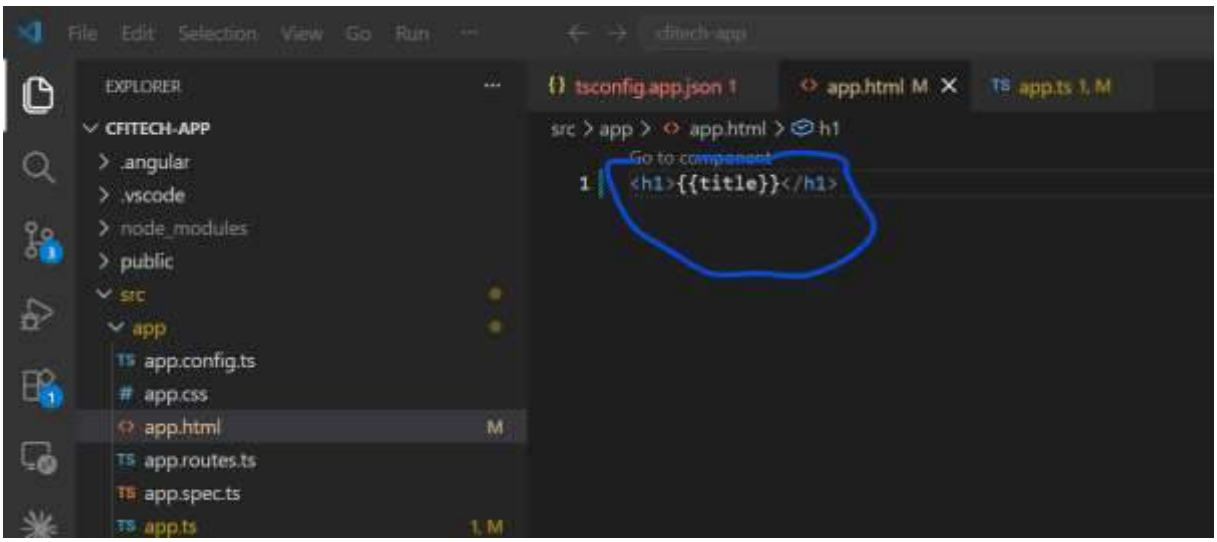
3. Dans HTML :

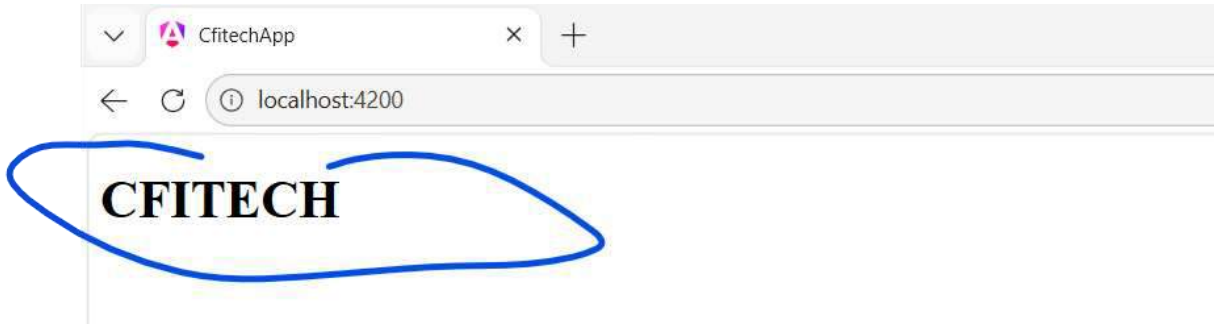
<h1>{{ title }}</h1>

-> affichage automatique

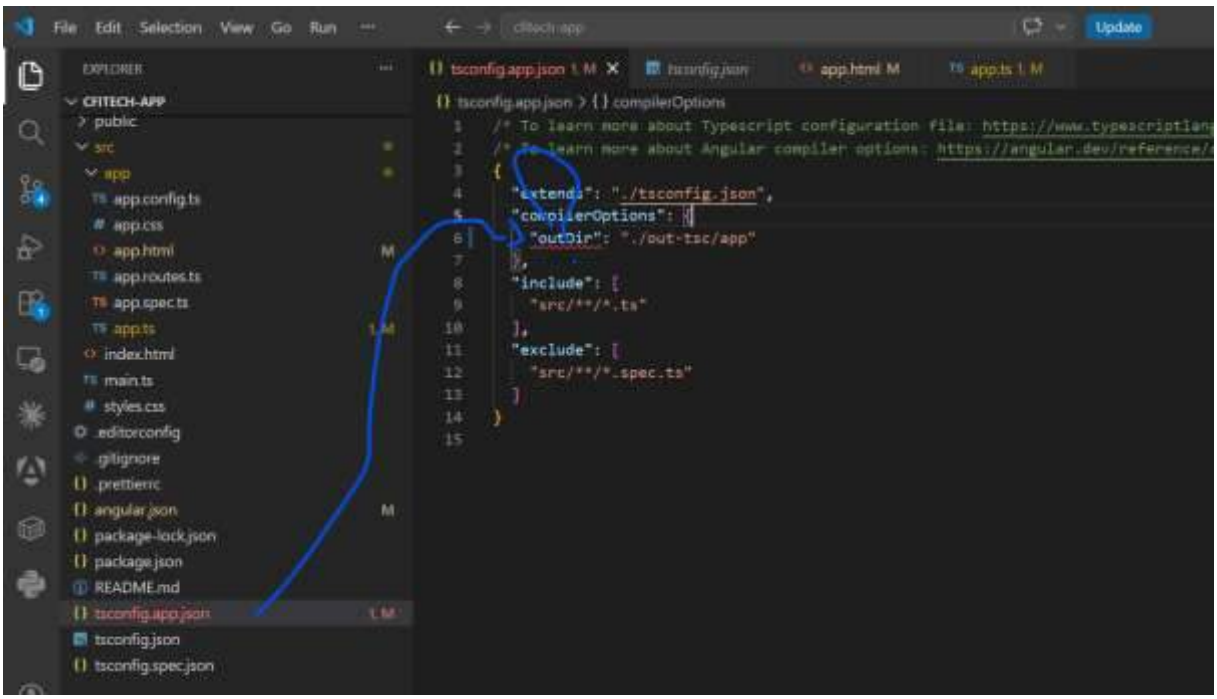
Rep :







Résoudre ce problème



outDir et rootDir travaillent toujours ensemble dans la compilation TypeScript. Le rôle de **rootDir** est de définir **le dossier source de ton code**, c'est-à-dire l'endroit où se trouvent tous les fichiers TypeScript à compiler (généralement src). Ensuite, **outDir** définit **le dossier de destination**, où TypeScript va générer les fichiers JavaScript après compilation. Autrement dit, rootDir représente **le point de départ (input)** et outDir représente **le point d'arrivée (output)**. Lorsque tu précises seulement outDir, TypeScript ne sait pas exactement d'où viennent les fichiers et demande donc explicitement rootDir pour éviter toute ambiguïté. Cela permet aussi de garder la même structure de dossiers entre le code source et le code compilé.

L’ordre n’a AUCUNE importance en JSON.

Donc :

"rootDir": "./src",

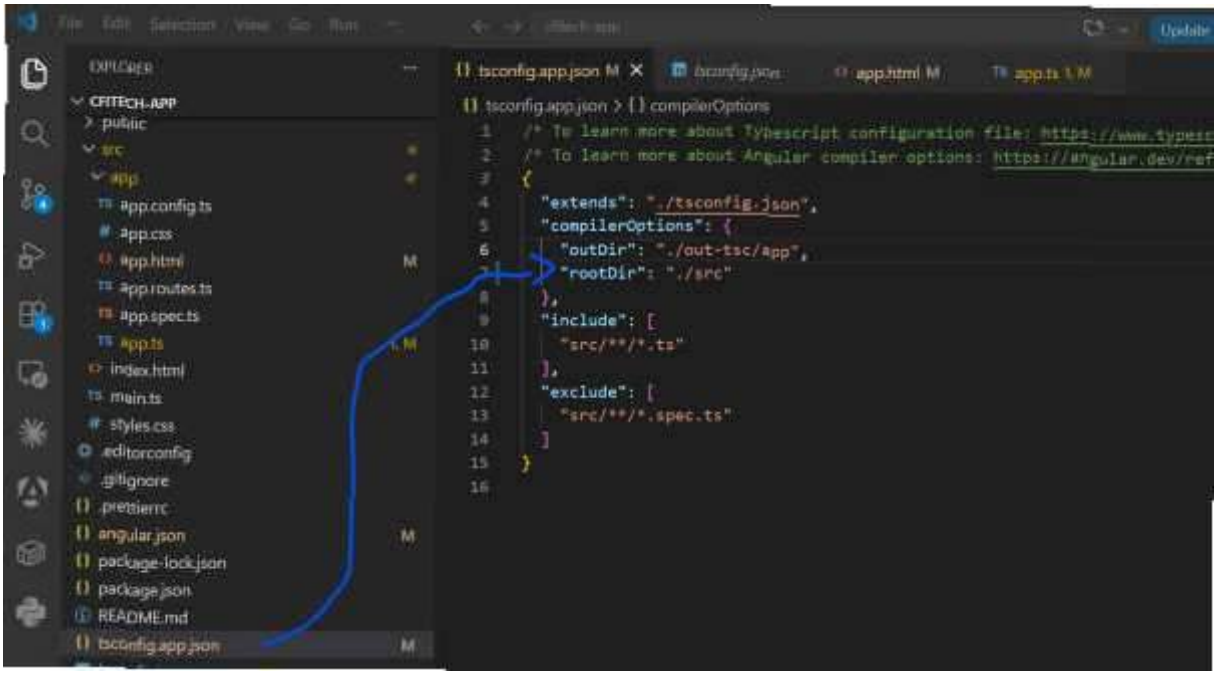
"outDir": "./out-tsc/app"

ou

"outDir": "./out-tsc/app",

"rootDir": "./src"

les deux sont exactement équivalents



Chapitre 4 : Les Composants

1. Définition d'un composant Angular

Un **composant** est une partie de l'interface utilisateur qui contient :

- une structure (HTML)
- une logique (TypeScript)
- un style (CSS)

Angular découpe une application en **petits blocs indépendants**

Chaque bloc = un composant

Ex :

Dans une application :

- le logo → composant
- le menu → composant
- la liste des stagiaires → composant

"Une application Angular est un assemblage de composants."

2. Création d'un composant

Commande

ng generate component header

version courte :

ng g c header

Angular crée automatiquement :

header/

├── header.ts

├── header.html

├── header.css

Angular génère :

- la logique
- l'interface
- le style

3. Structure d'un composant

Un composant = 3 parties

1. TypeScript (la logique)

```
export class Header {
 titre = "CFITECH";
}
```

2. HTML (l'affichage)

```
<h1>{{ titre }}</h1>
```

3. CSS (le style)

```
h1 {
 color: blue;
}
```

Angular sépare :

- ce que fait le composant
- ce qu'il affiche
- son apparence

4. Le décorateur @Component

Définition

Le décorateur @Component est une configuration qui permet à Angular de comprendre comment fonctionne un composant.

Ex

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

@Component({
 selector: 'app-header',
 standalone: true,
 templateUrl: './header.html',
 styleUrls: ['./header.css']
})
export class Header {}
```

selector

définit le nom de la balise HTML

```
selector: 'app-header'
```

utilisation :

```
<app-header></app-header>
```

templateUrl

lien vers le fichier HTML

styleUrls

lien vers le CSS

standalone

indique que le composant est autonome

5. Standalone Components

Définition

Un composant standalone est un composant qui peut fonctionner sans module (NgModule).

Ex :

```
@Component({
 standalone: true
})
```

6. Utiliser un composant

Objectif

Afficher un composant dans un autre

1. Créer le composant

ng g c header

2. Utiliser son selector

<app-header></app-header>

Angular affiche le composant

7. Exercice

Créer et afficher plusieurs composants

Étapes

1. Créer :

ng g c header

ng g c footer

2. Modifier le HTML

header :

<h1>Bienvenue CFITECH</h1>

footer :

© CFITECH

3. Afficher dans le composant principale

Quand Angular lance l'application, il suit cet ordre :

a. Angular lit app.ts

Il regarde :

```
@Component({
 imports: [Header, Footer]
})
```

À ce moment :

V Angular "connaît" les composants
V les enregistre

b. Angular lit app.html

```
<app-header></app-header>
<app-footer></app-footer>
```

Il voit :

- <app-header>
- <app-footer>

3. Angular fait la correspondance

Il dit :

"Ah, <app-header> correspond à Header"

Puis il affiche le composant

L'ordre est :

IMPORT (app.ts)

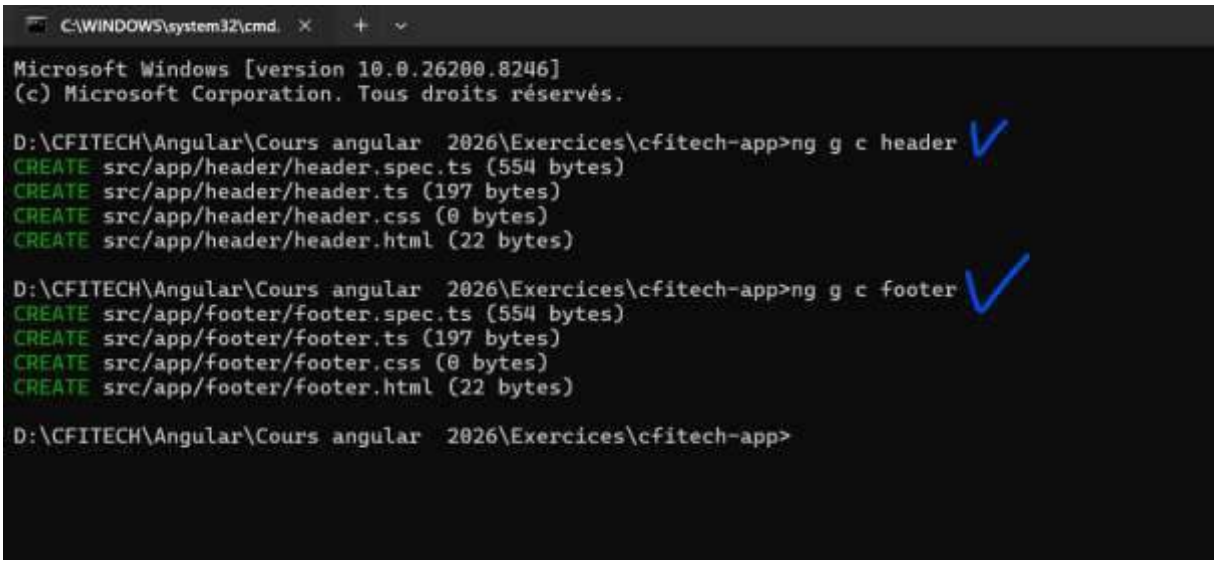
↓

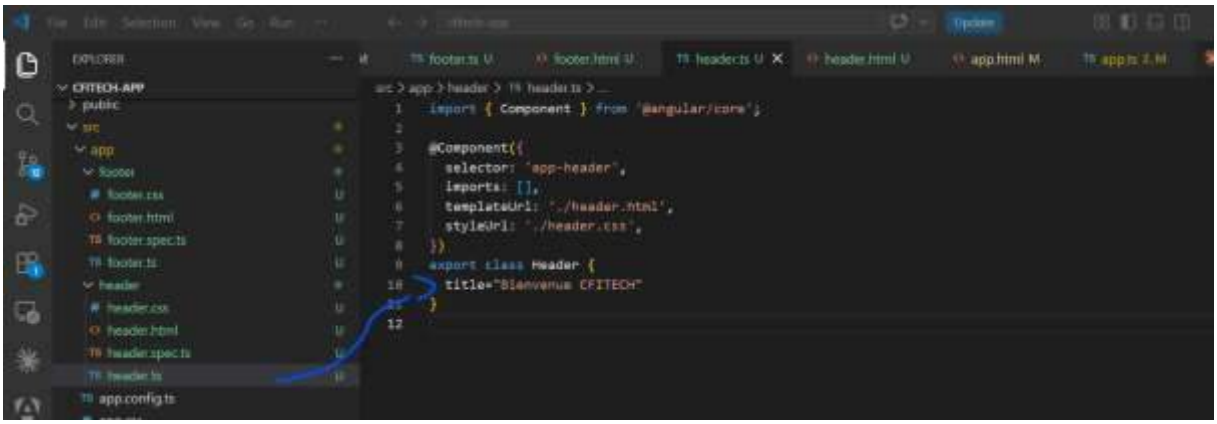
HTML (app.html)

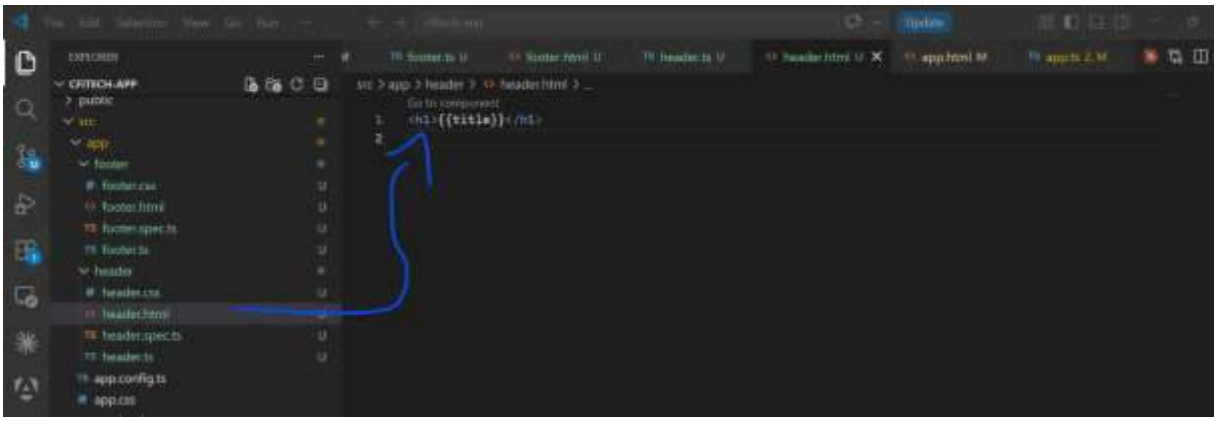
↓

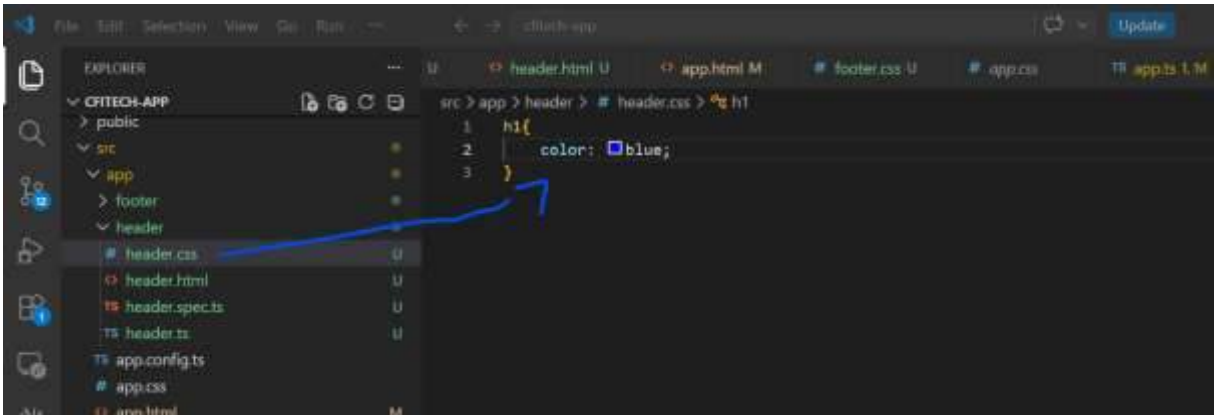
AFFICHAGE

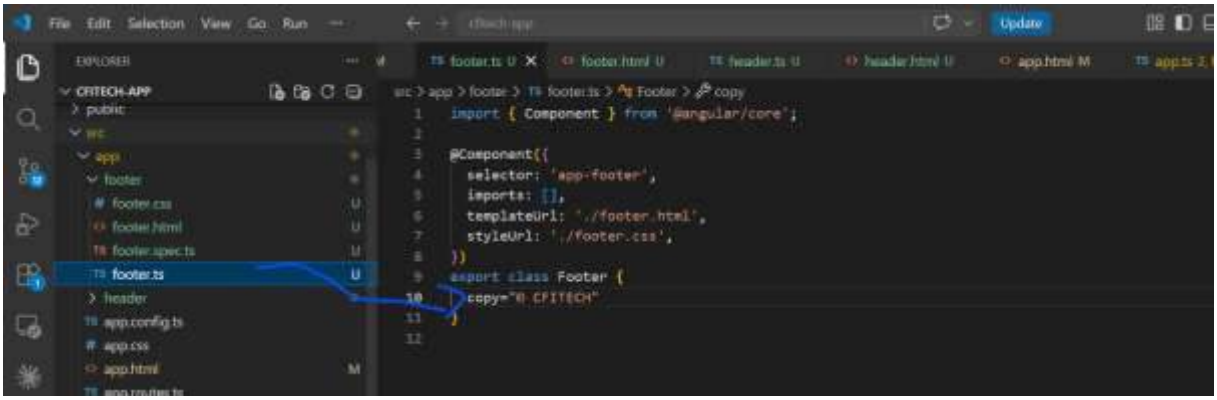
Rep :

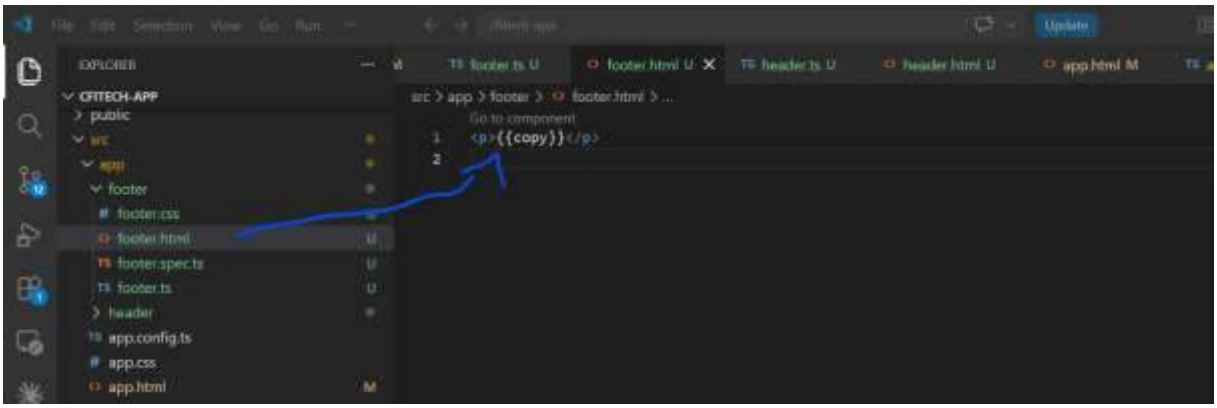

C:\WINDOWS\system32\cmd.
Microsoft Windows [version 10.0.26200.8246]
(c) Microsoft Corporation. Tous droits réservés.
D:\CFITECH\Angular\Cours angular 2026\Exercices\cfitech-app>ng g c header
CREATE src/app/header/header.spec.ts (554 bytes)
CREATE src/app/header/header.ts (197 bytes)
CREATE src/app/header/header.css (0 bytes)
CREATE src/app/header/header.html (22 bytes)
D:\CFITECH\Angular\Cours angular 2026\Exercices\cfitech-app>ng g c footer
CREATE src/app/footer/footer.spec.ts (554 bytes)
CREATE src/app/footer/footer.ts (197 bytes)
CREATE src/app/footer/footer.css (0 bytes)
CREATE src/app/footer/footer.html (22 bytes)
D:\CFITECH\Angular\Cours angular 2026\Exercices\cfitech-app>

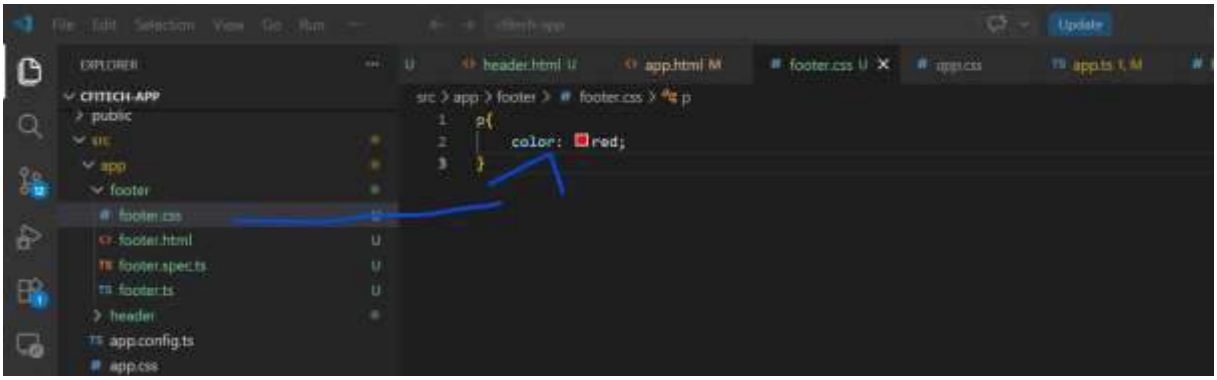

import { Component } from '@angular/core';
@Component({
selector: 'app-header',
imports: [],
templateUrl: './header.html',
styleUrl: './header.css',
})
export class Header {
title="Bienvenue CFITECH"
}

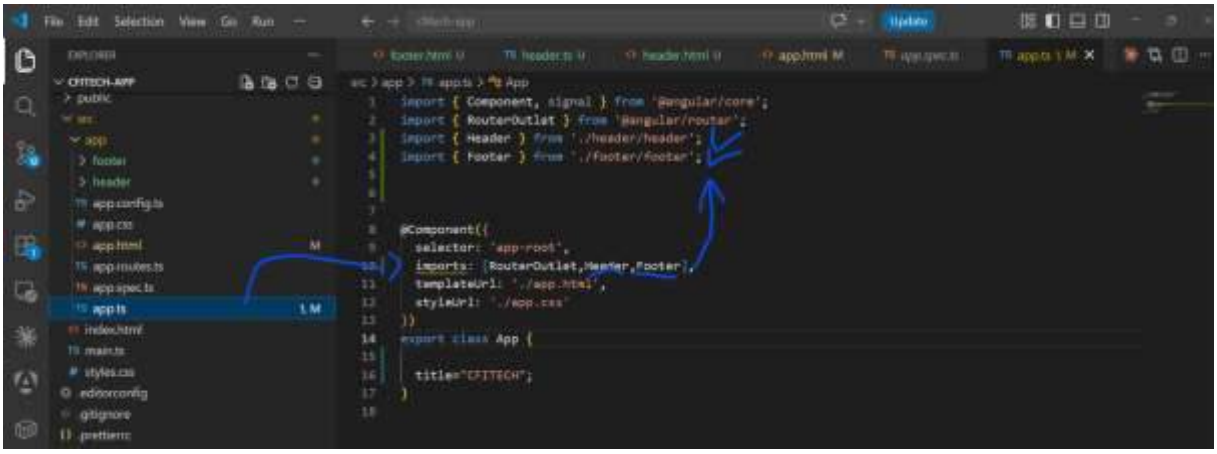

<h1>{{title}}</h1>

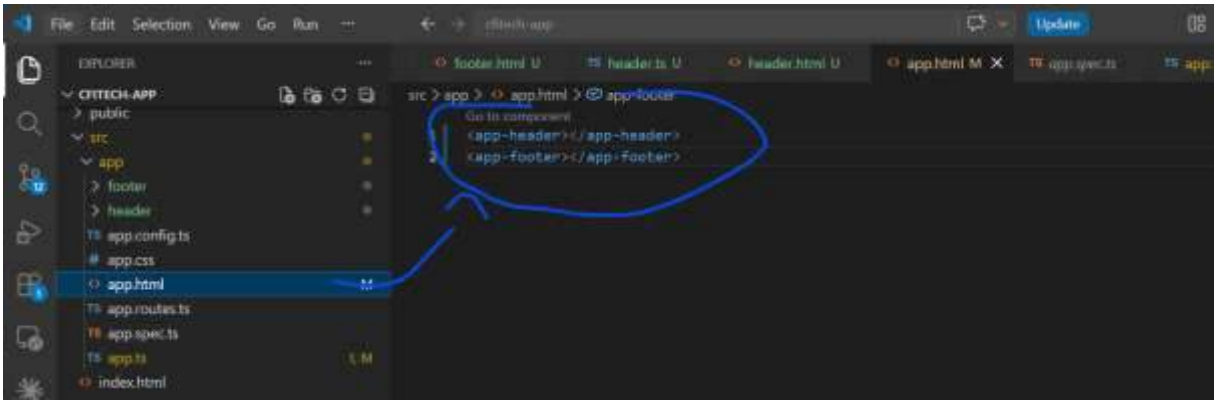

CFITECH-APP
public
src
app
footer
header
header.css
header.html
header.spec.ts
header.ts
app.config.ts
app.css
src > app > header > # header.css > h1
h1{
color: blue;
}


CFITECH-APP
footer.ts
footer.html
header.ts
header.html
app.html
src > app > footer > footer.ts > Footer > copy
import { Component } from '@angular/core';
@Component({
selector: 'app-footer',
imports: [],
templateUrl: './footer.html',
styleUrl: './footer.css',
})
export class Footer {
copy="© CFITECH"
}


CFITECH-APP
footer.html
src > app > footer > footer.html > ...
{{copy}}


CFITECH-APP
footer.css
src > app > footer > # footer.css > p
p{
color: red;
}


import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { Header } from './header/header';
import { Footer } from './footer/footer';
@Component({
selector: 'app-root',
imports: [RouterOutlet,Header,Footer],
templateUrl: './app.html',
styleUrl: './app.css'
})
export class App {
title="CFITECH";
}


<app-header></app-header>
<app-footer></app-footer>


CfitechApp
localhost:4200
Bienvenue CFITECH
© CFITECH

Chapitre 5 : Data Binding & Signals

1. Qu’est-ce que le Data Binding ? Angular

Le **Data Binding** est le mécanisme qui permet de connecter les données (TypeScript) avec l’interface (HTML).

Il y a 4 formes de data Binding(liaison des données)

a. Interpolation (`{{ }}`)

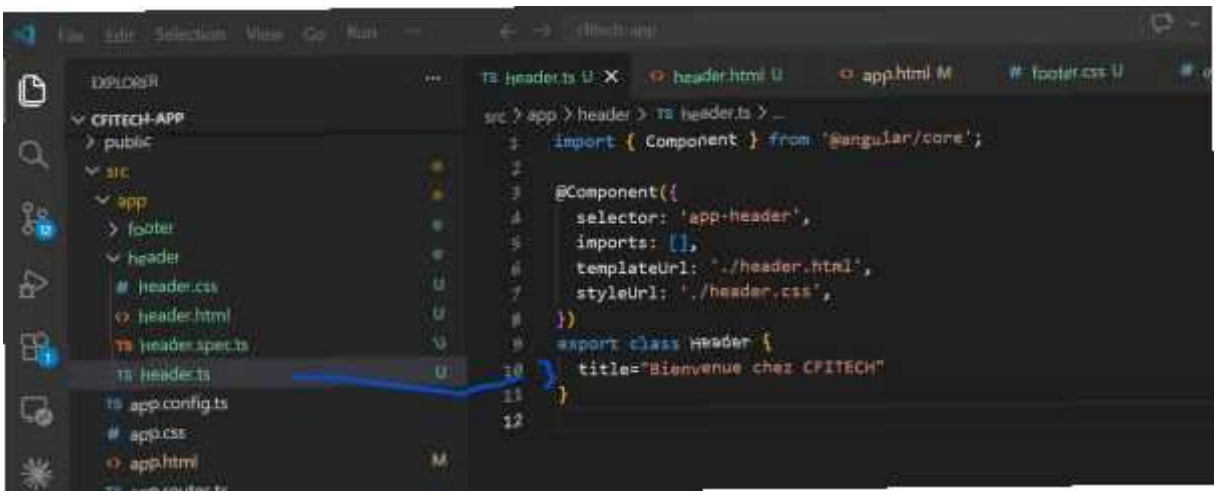
L'**interpolation** en Angular est une technique qui permet d’**afficher des données dynamiques** (généralement depuis le composant TypeScript) **dans le HTML**, en utilisant la syntaxe `{{ variable }}`.

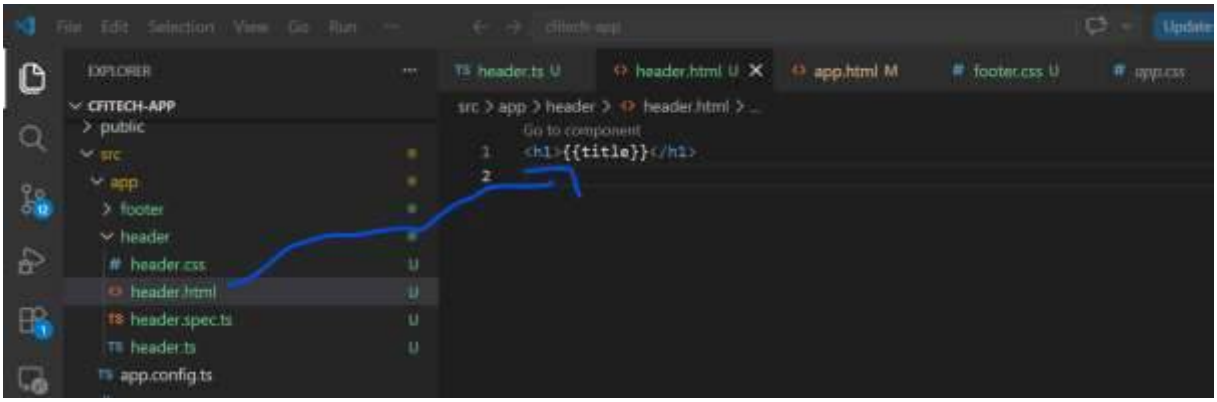
À quoi sert l'interpolation ?

Elle permet de **lier** les données du composant à la vue (template HTML) — c’est ce qu’on appelle **le data binding unidirectionnel** : les données vont du **composant vers la vue**.

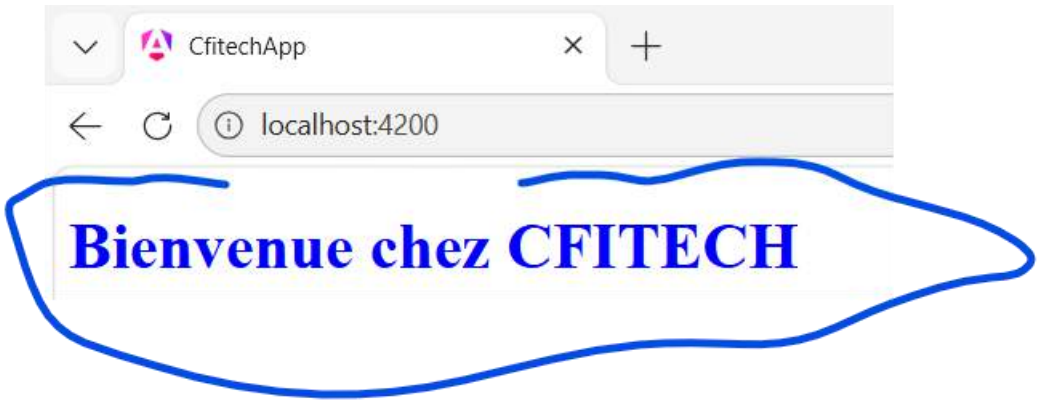
Ex :

Composant TypeScript : `header.ts`





Et puis l'importer dans le composant principal (app.ts-> app.html)



- `Title` est une propriété définie dans le fichier `.ts`.
- Le HTML, `{{ title }}` est remplacé automatiquement par la valeur `"Bienvenue chez CFITECH"`.
- Si la valeur de `title` change dans le composant, Angular **mettra automatiquement à jour l'affichage** sans recharger la page.

b. Property Binding (`[property]`)

Le **property binding** te permet de **lier une propriété d'un élément HTML** (comme `src`, `href`, `disabled`, etc.) à une **valeur définie dans ton composant TypeScript**.

Lier une valeur à une propriété DOM :

[elementProperty]="valeurDuComposant"

Par exemple :

<img [src]="imageUrl">

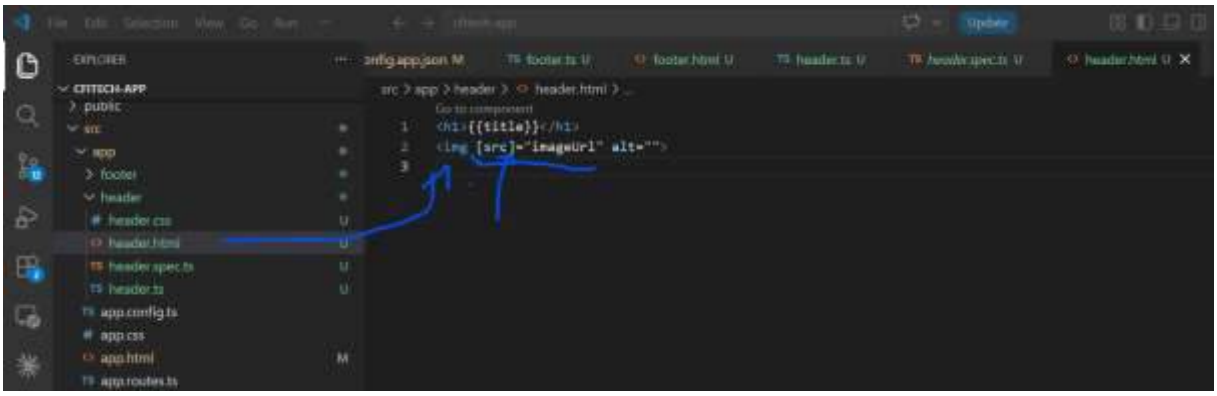
Ici, `imageUrl` est une variable définie dans le composant. Angular **met dynamiquement à jour** l'attribut `src` de l'image avec sa valeur.

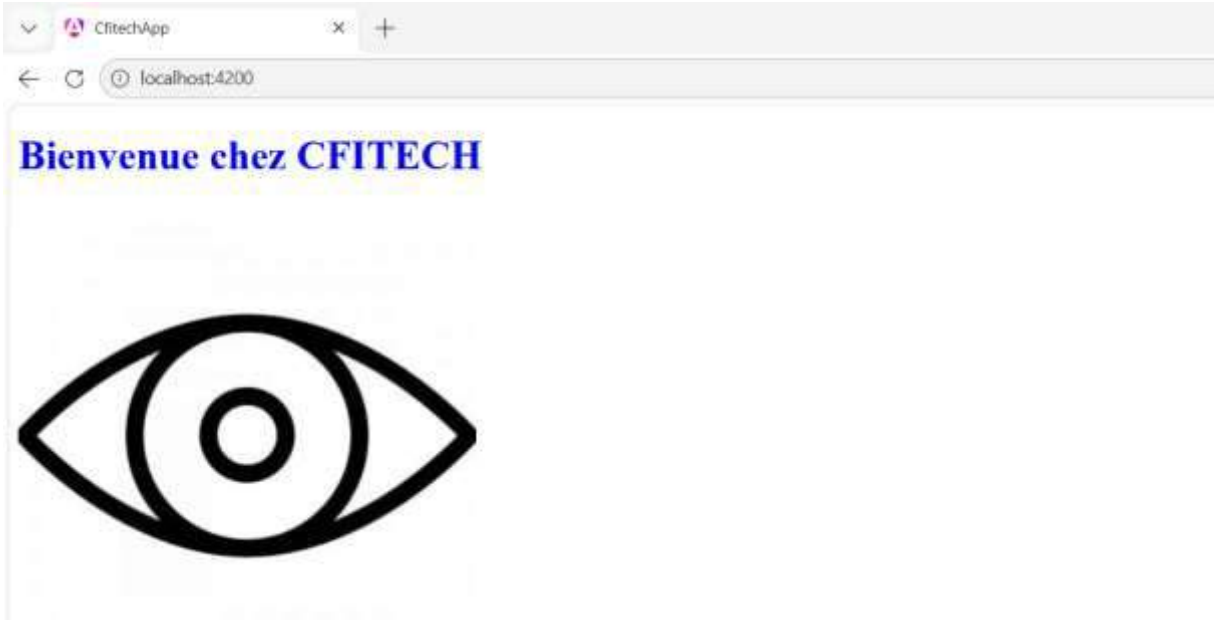
Ex :

`header.ts` (le composant TypeScript)

Ex : https://png.pngtree.com/png-clipart/20191027/ourmid/pngtree-view-icon-png-image_1869829.jpg

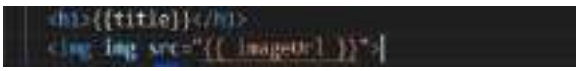
`header.html` (le template) :





Pourquoi ne pas utiliser simplement l'interpolation ({{ }}) ?

<img src="{{ imageUrl }}">



Et ça fonctionne ! Mais attention :

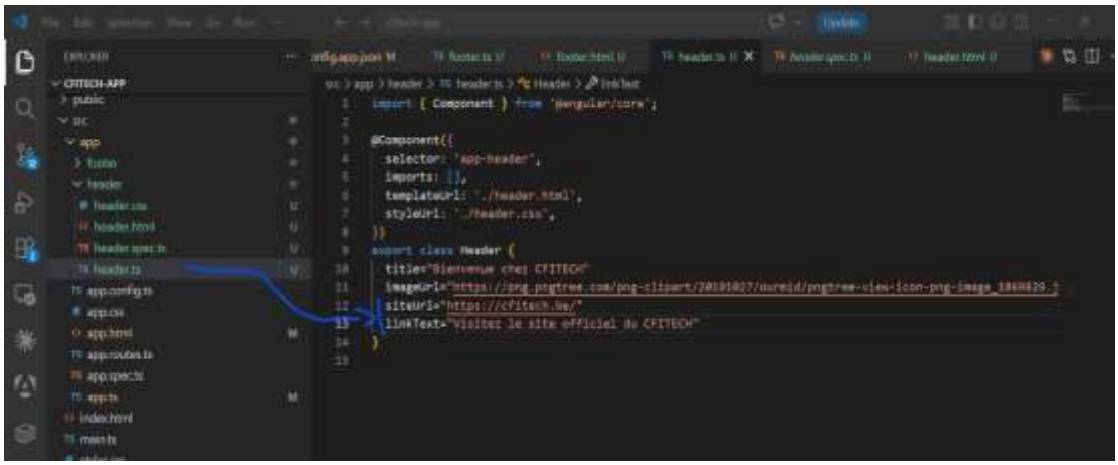
Méthode	Fonctionne	Recommandé	Remarque
`src="{{ imageUrl }}"`	Oui	Non conseillé	Angular traite l'attribut comme une **chaîne de texte**
`[src]="imageUrl"`	Oui	Oui	Lien direct entre le DOM et le composant

Donc, on **utilise toujours** **`[property]="..."`** pour des attributs HTML natifs, surtout si ce sont :

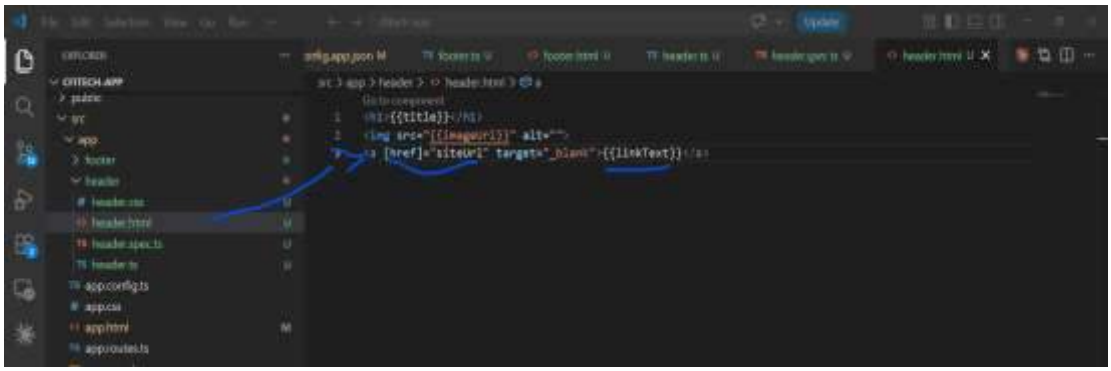
- des URLs (`src`, `href`)
- des booléens (`[disabled]`, `[checked]`)
- ou d'autres propriétés du DOM

➔ **Exemple de Property Binding avec** **`href`** en Angular, pour créer un lien dynamique :

Composant TypeScript (`header.ts`)



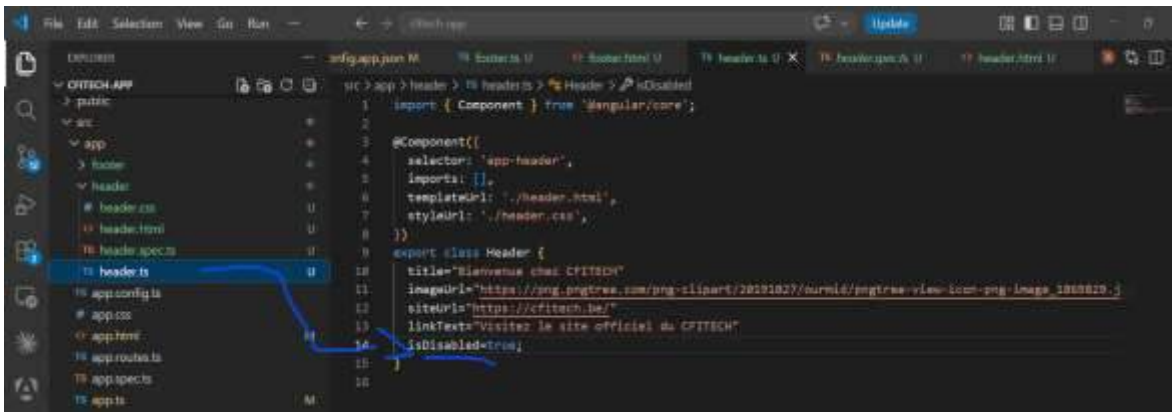
Template HTML (`header.html`)



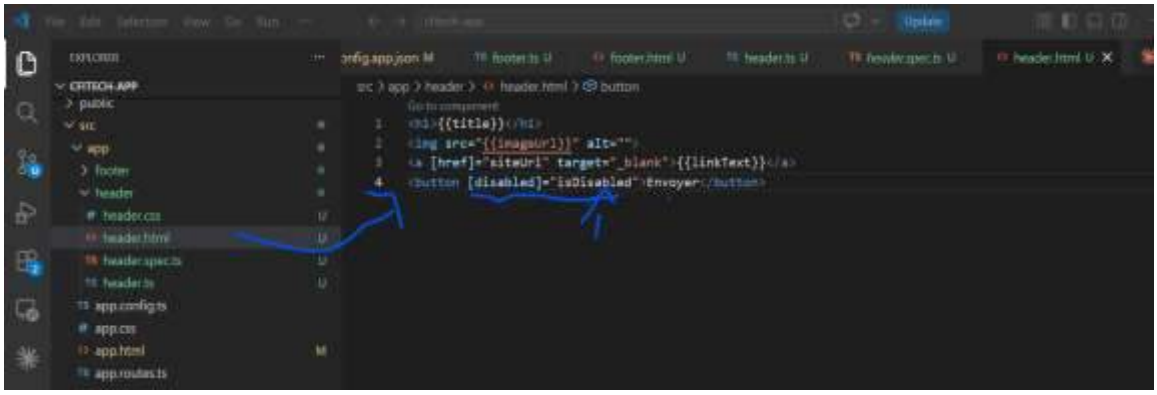
- `[href]="siteUrl"` est un **property binding** : Angular lie la propriété `href` de la balise `<a>` à la variable `siteUrl` du composant.
- `target="_blank"` pour ouvrir le lien dans un nouvel onglet.
- `{{ linkText }}` affiche le texte du lien.

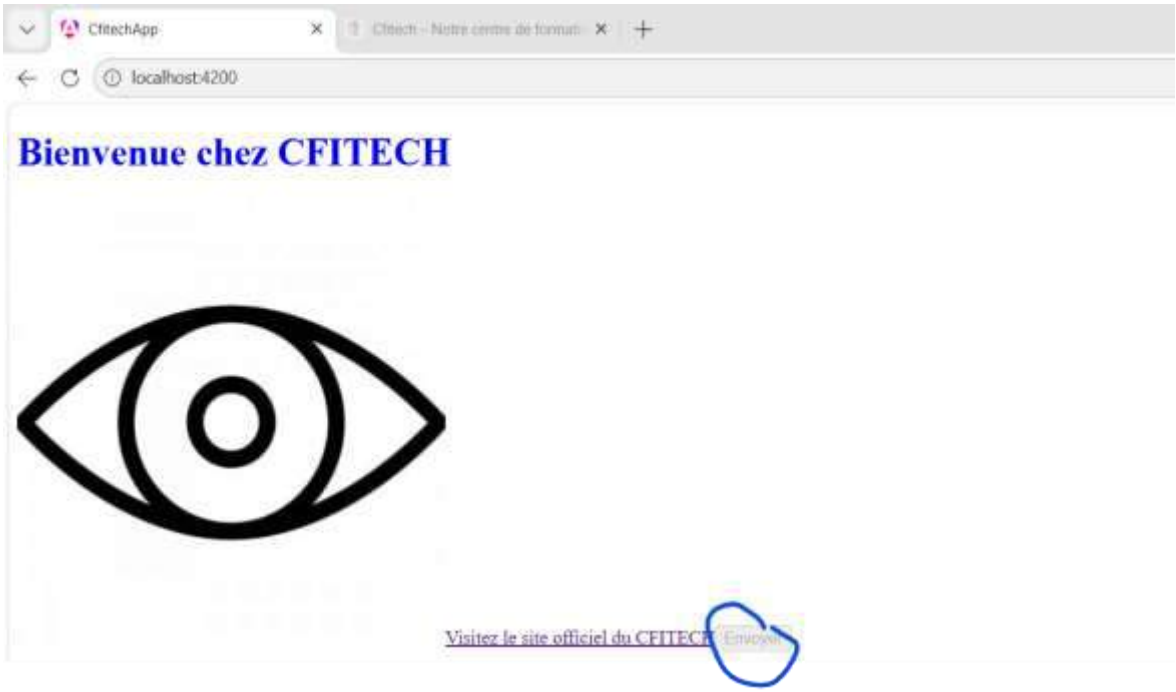
➔ Ex pour disabled :

Composant TypeScript (`header.ts`)

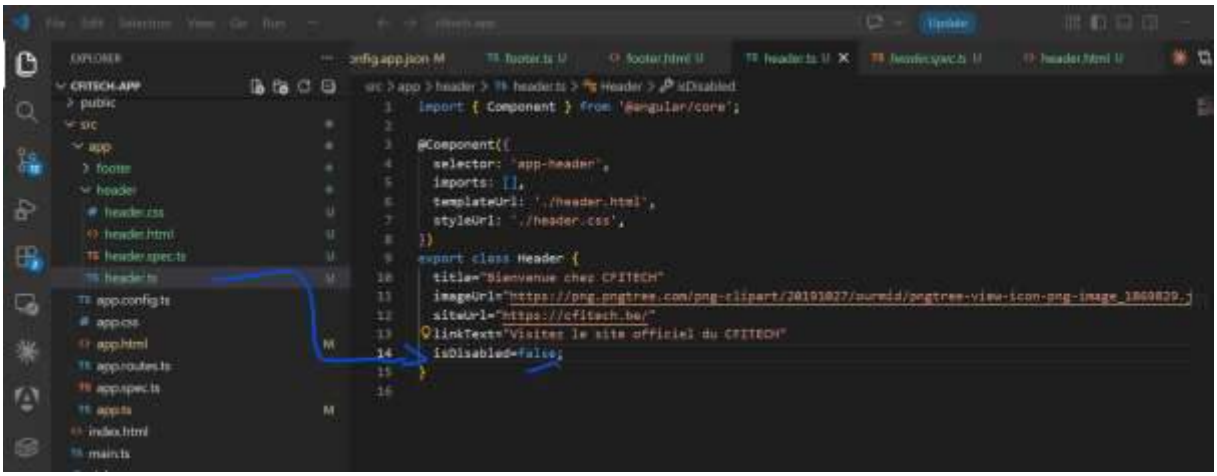


Template HTML (`header.html`)




CftechApp
localhost:4200
Bienvenue chez CFITECH
Visitez le site officiel du CFITECH

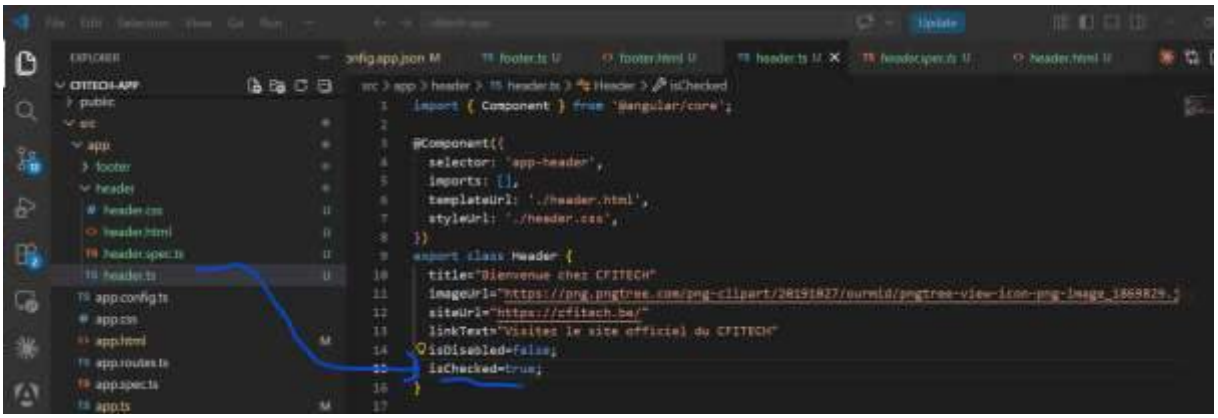
Activer :


EXPLORER
CFITECH-APP
public
src
app
footer
header
header.css
header.html
header.spec.ts
header.ts
app.config.ts
app.css
app.html
app.routes.ts
app.spec.ts
app.ts
index.html
main.ts
import { Component } from '@angular/core';
@Component({
selector: 'app-header',
imports: [],
templateUrl: './header.html',
styleUrl: './header.css',
})
export class Header {
title="Bienvenue chez CFITECH"
imageUrl="https://cfitech.be/"
siteUrl="https://cfitech.be/"
linkText="Visitez le site officiel du CFITECH"
isDisabled=false;
}

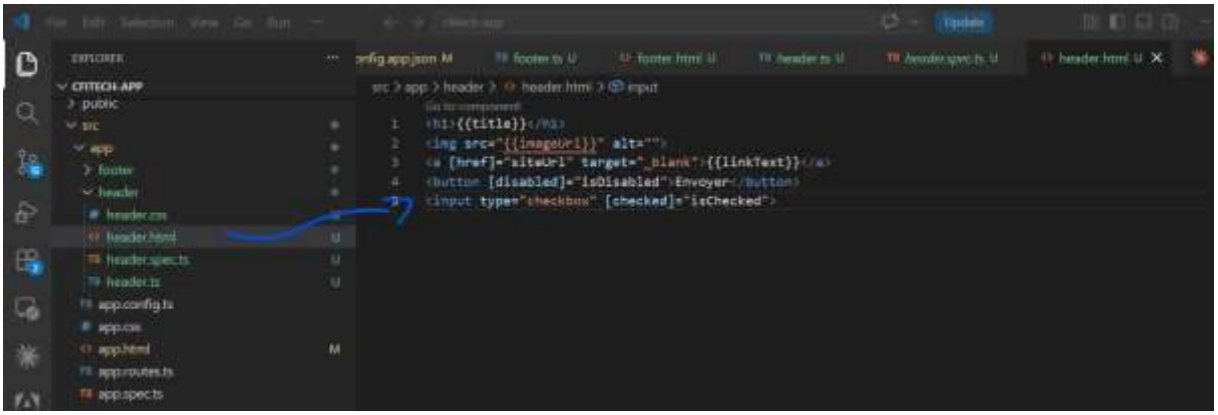
- `[disabled]="isDisabled"` lie dynamiquement l'attribut `disabled` à la variable `isDisabled`.
- Si `isDisabled` vaut `true`, le bouton est désactivé.
- Si `isDisabled` vaut `false`, le bouton est activé.

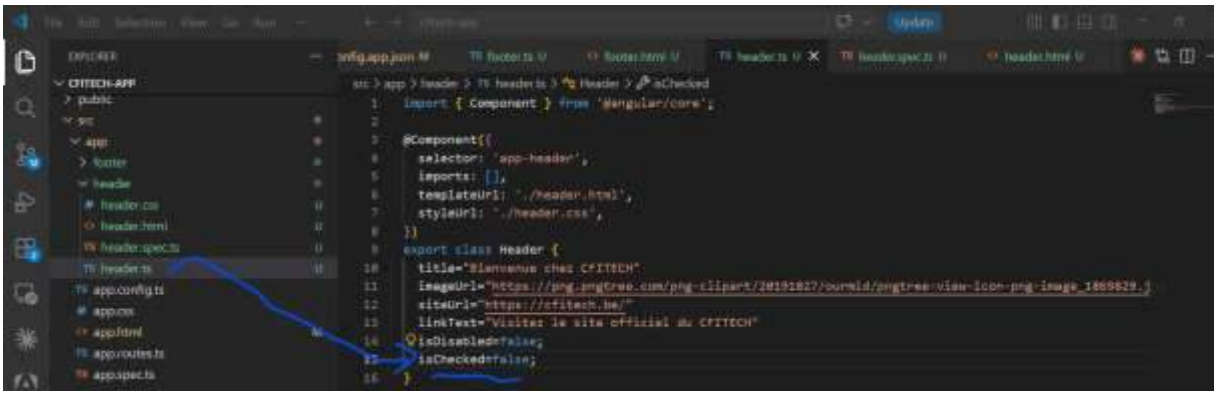
➔ Ex pour `checked` :

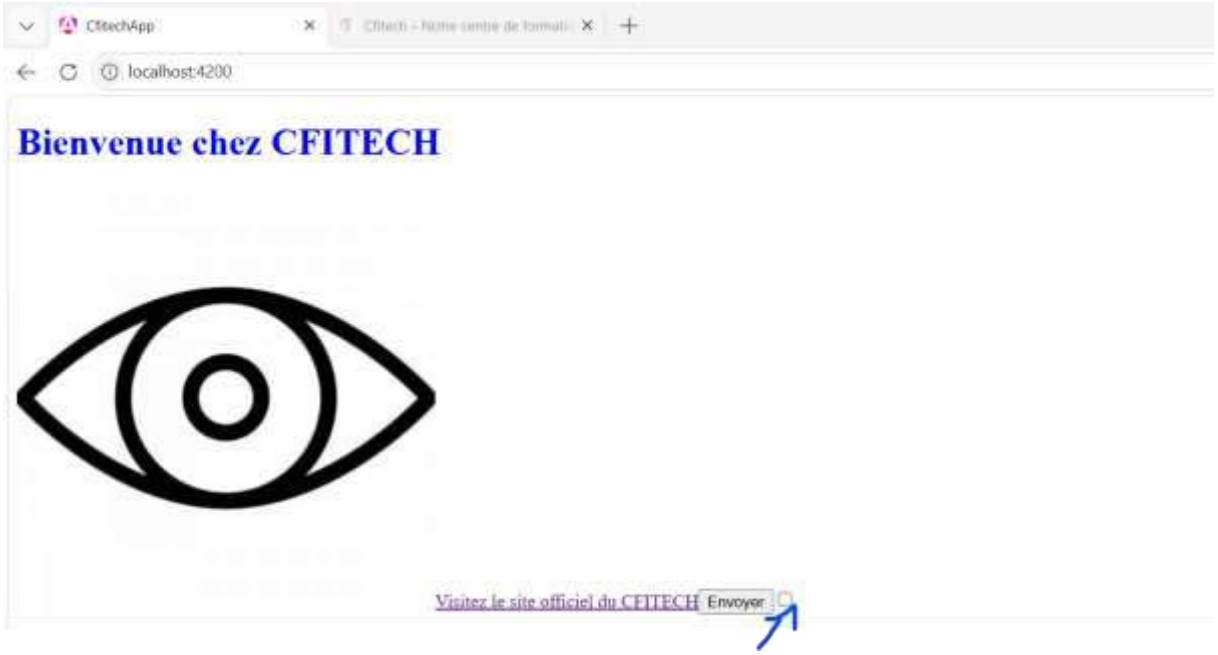
`header.ts`



header.html




CFITECH-APP
public
src
app
footer
header
header.css
header.html
header.spec.ts
header.ts
app.config.ts
app.css
app.html
app.routes.ts
app.spec.ts
import { Component } from '@angular/core';
@Component({
selector: 'app-header',
imports: [],
templateUrl: './header.html',
styleUrl: './header.css',
})
export class Header {
title="Bienvenue chez CFITECH"
imageUrl="https://png.pngtree.com/png-clipart/20191027/ourmid/pngtree-view-icon-png-image_1869629.
siteUrl="https://cfitech.be/"
linkText="Visitez le site officiel du CFITECH"
isDisabled=false;
isChecked=false;
}


CitechApp
localhost:4200
Bienvenue chez CFITECH
Visitez le site officiel du CFITECH
Envoyer

c. Event Binding ((event))

Le **Event Binding** permet de **réagir à une action de l'utilisateur** (comme un clic, un survol, une saisie clavier, etc.).

La syntaxe est :

(elementDOM)="méthodeDuComposant()"

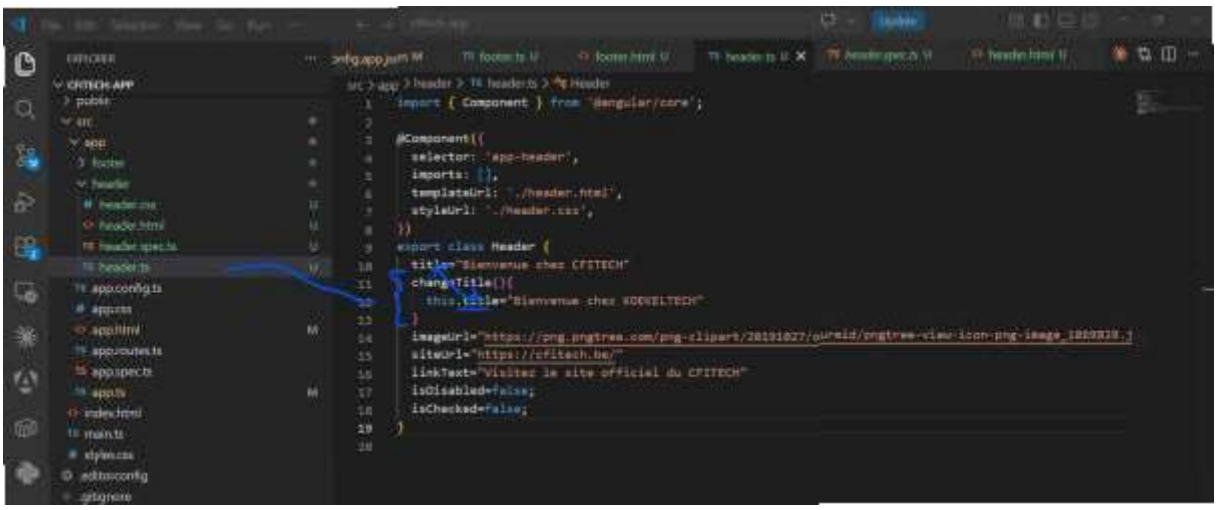
EX :

<button (click)="changeTitle()">Change</button>

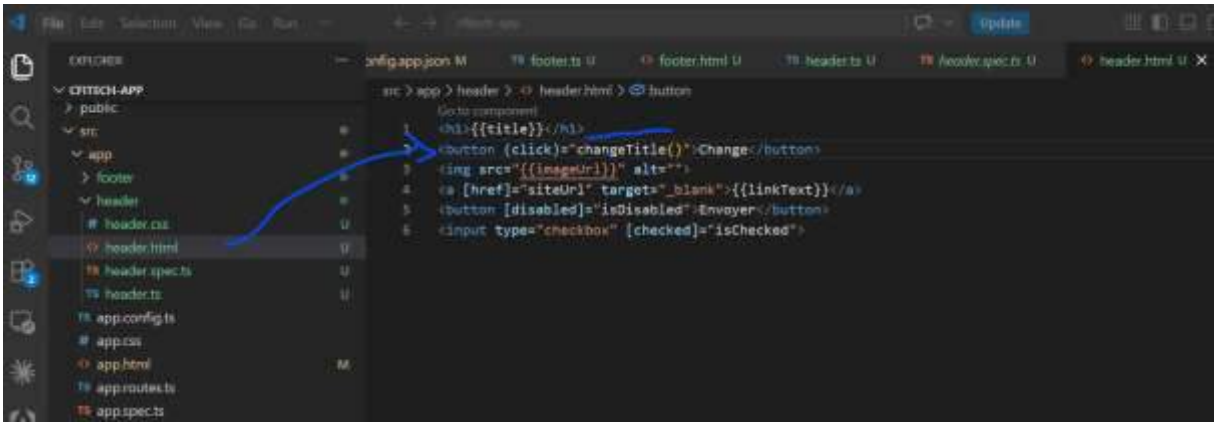
Quand l'utilisateur **clique** sur le bouton, Angular appelle la méthode `changeTitle()` définie dans le composant TypeScript.

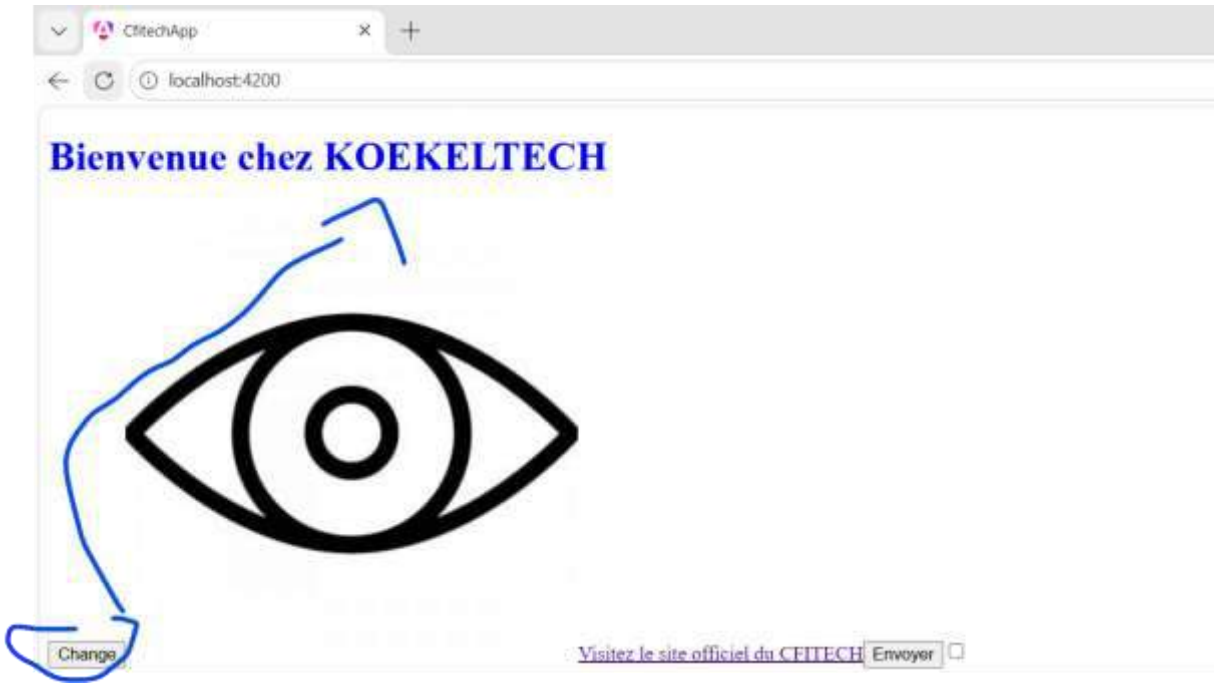
Ex complet :

`header.ts` (le composant TypeScript) :



`header.html` (le template)





Objectif	Syntaxe
Réagir à un clic	`(click)="maMethode()"`

Le fichier `index.html` (dans `src/`) contient déjà le document HTML global (`<html>`, `<head>`, `<body>`), et Angular injecte le composant via `<app-root></app-root>`.

Donc si tu mets encore une fois `<html>`, `<head>`, `<body>` dans `app.component.html`, Angular ne peut pas comprendre/interpréter `{{ title }}` ou `(click)`.

d. Two-Way Binding ([(ngModel)])

- Si tu modifies la variable dans le code → la vue change.
- Si tu modifies le champ dans la vue → la variable change.

Ex: champ texte lié à une variable

On va utiliser la directive `ngModel` est au cœur du **Two-Way Data Binding** en Angular : elle synchronise automatiquement la **vue** (le template) et le **modèle** (la propriété du composant), dans les deux sens.

Qu'est-ce que `ngModel` ?

- C'est une **directive** fournie par le **`FormsModule`**.
- Une **directive** est une instruction qui permet de modifier le comportement ou l'apparence d'un élément HTML.
- Elle permet de :
    1. **Initialiser** un champ de formulaire avec la valeur d'une propriété du composant.
    2. **Mettre à jour** cette propriété quand l'utilisateur modifie le champ.

Angular fonctionne avec une architecture modulaire. Cela signifie que le framework est découpé en plusieurs modules, chaque module regroupant des fonctionnalités spécifiques. Par exemple, les formulaires sont regroupés dans FormsModule, les directives courantes comme ngClass et ngStyle sont regroupées dans CommonModule, et la navigation entre pages est gérée par RouterModule. Ces modules ne sont pas des librairies externes : ils font déjà partie du framework Angular. Lorsque Angular est installé, tous ces modules sont déjà disponibles dans le projet.

Cependant, Angular ne charge pas automatiquement toutes ses fonctionnalités. Le développeur doit importer uniquement les modules dont il a besoin. Cette approche permet d'optimiser les performances de l'application et d'éviter de charger du code inutile. Par exemple, si une application ne contient aucun formulaire, il serait inutile de charger les fonctionnalités liées aux formulaires. C'est pour cela que, pour utiliser ngModel, il faut importer FormsModule, et pour utiliser ngClass ou ngStyle, il faut importer CommonModule.

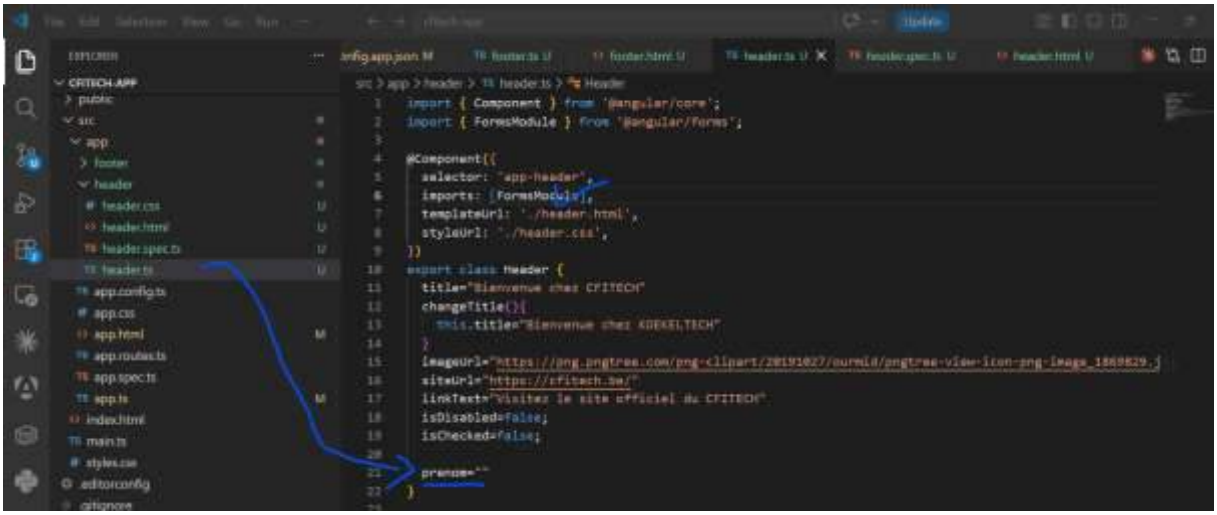
Prenons un exemple concret avec ngModel. Cette directive permet de lier automatiquement un champ de saisie à une variable TypeScript. Comme cette fonctionnalité appartient au système de formulaires Angular, elle est placée dans FormsModule. Si le développeur oublie d'importer ce module, Angular ne reconnaît pas ngModel et affiche une erreur. Ce fonctionnement montre qu'Angular est organisé comme une boîte à outils : toutes les fonctionnalités existent déjà dans le framework, mais le développeur choisit uniquement les outils nécessaires à son application.

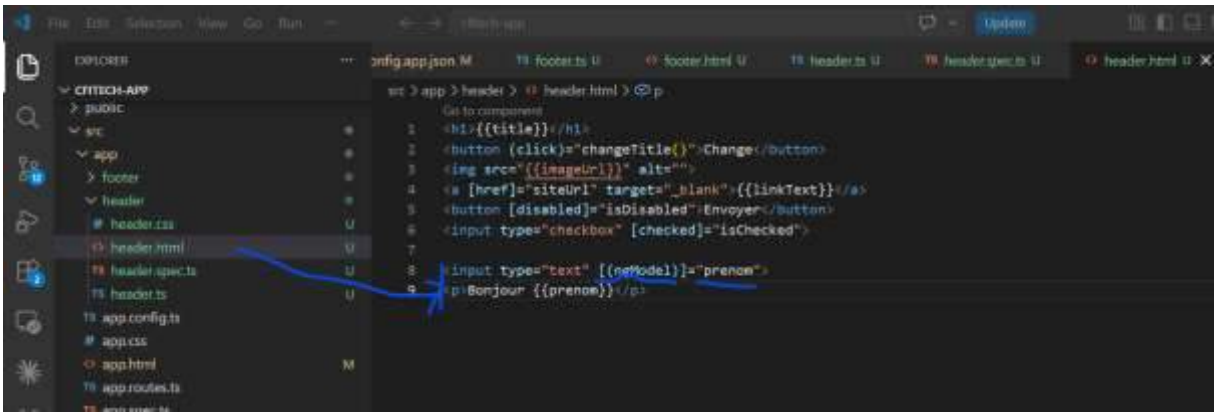
Cette architecture modulaire est très importante dans les projets professionnels, car elle rend les applications plus propres, mieux organisées, plus performantes et plus faciles à maintenir.

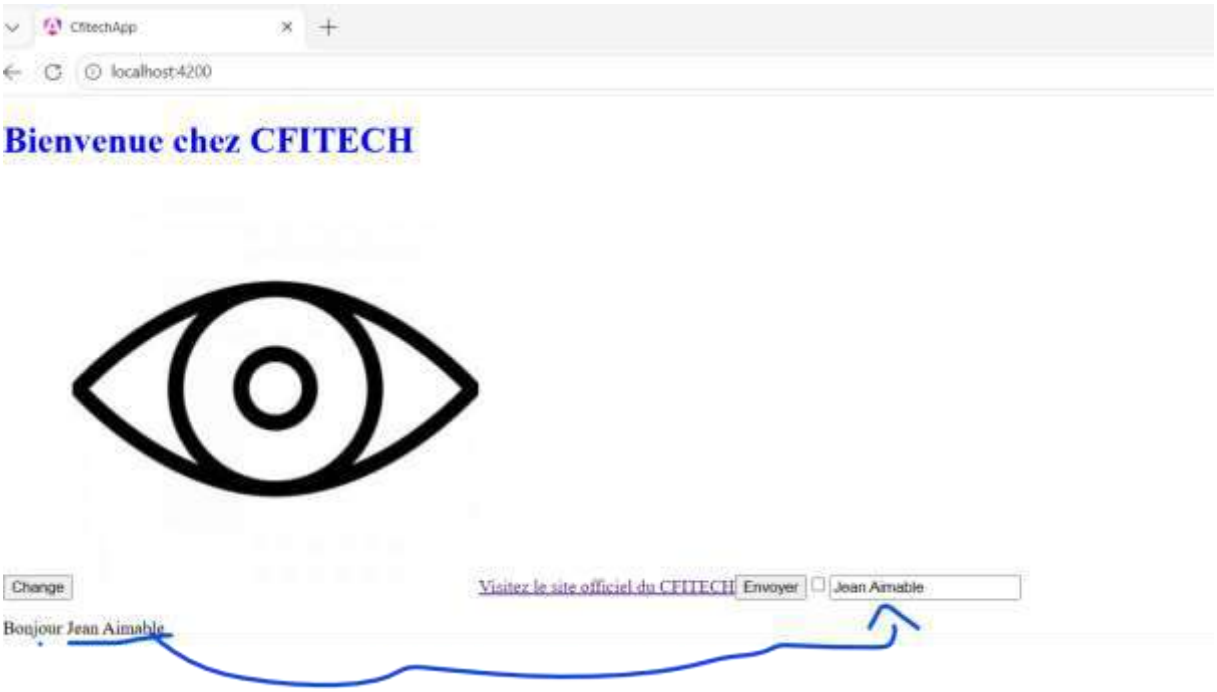
Dans React, la philosophie est différente. React est principalement une librairie d'interface utilisateur. Il ne fournit pas directement certaines fonctionnalités importantes comme le routing, les formulaires avancés ou la gestion globale des données. Le développeur doit installer lui-même des librairies externes selon les besoins du projet. Par exemple, pour la navigation, on installe souvent react-router-dom, pour les formulaires on utilise parfois react-hook-form ou formik, et pour les requêtes HTTP on ajoute souvent axios. Contrairement à Angular, ces outils ne font pas partie du cœur de React.

La différence importante est donc la suivante : dans Angular, les modules comme FormsModule sont déjà intégrés au framework mais doivent être activés par import pour optimiser l'application. Dans React, beaucoup de fonctionnalités doivent être ajoutées depuis des librairies externes téléchargées séparément. Angular fournit déjà la boîte à outils complète, tandis que React fournit surtout la partie interface utilisateur et laisse le développeur construire le reste de l'écosystème autour du projet.

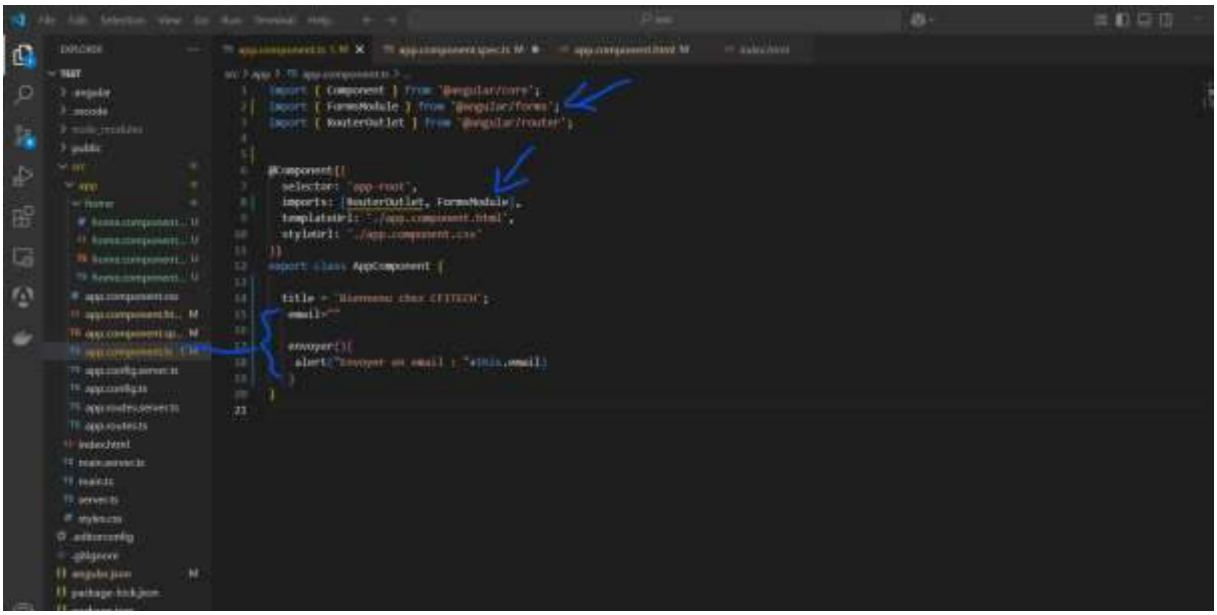
C'est pour cette raison qu'Angular est considéré comme un framework complet, alors que React est considéré comme une librairie UI flexible.

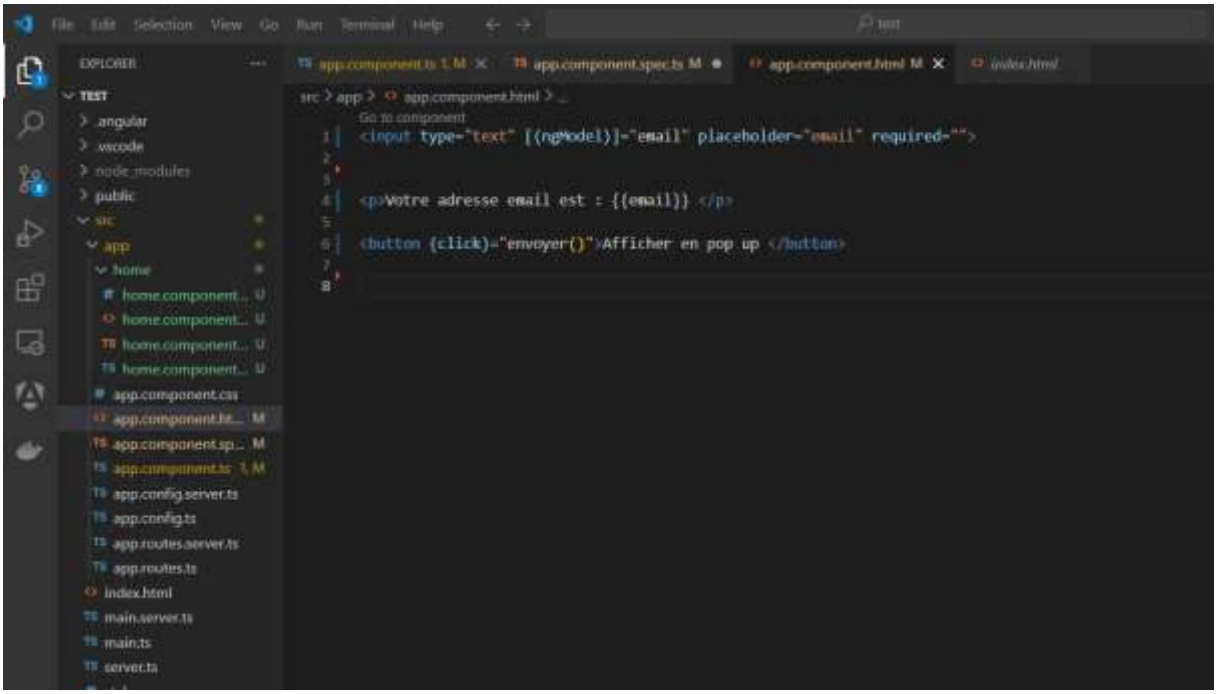

import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
@Component({
selector: 'app-header',
imports: [FormsModule],
templateUrl: './header.html',
styleUrl: './header.css',
})
export class Header {
title="Bienvenue chez CFITECH"
changeTitle(){
this.title="Bienvenue chez KOKEELTECH"
}
imageUrl="https://png.pngtree.com/png-clipart/20191027/ourmid/pngtree-view-icon-png-image_1869829.j
siteUrl="https://cfitech.bw/"
linkText="Visitez le site officiel du CFITECH"
isDisabled=false;
isChecked=false;
prenom=""
}

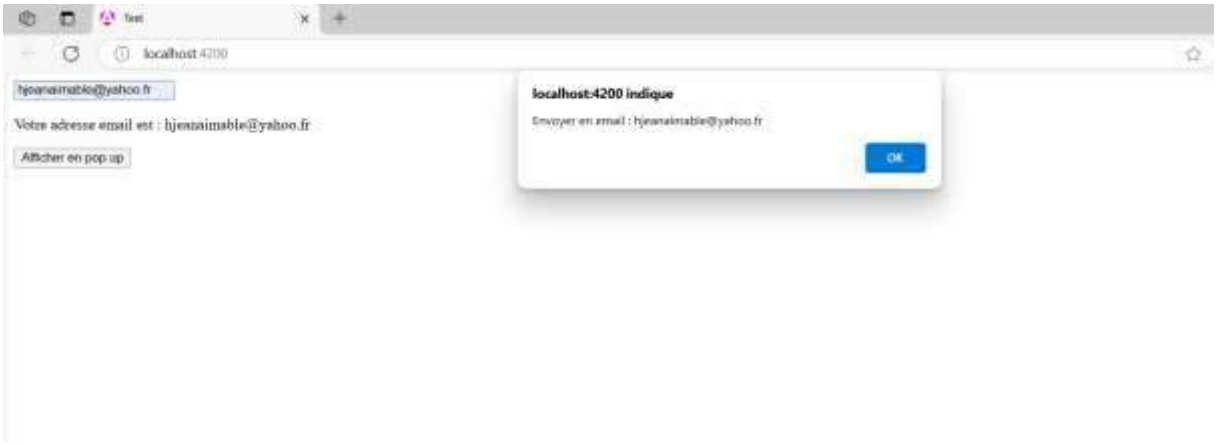

<h1>{{title}}</h1>
<button (click)="changeTitle()">Change</button>
<img src="{{imageUrl}}" alt="">
<a [href]="siteUrl" target="_blank">{{linkText}}
<button [disabled]="isDisabled">Envoyer</button>
<input type="checkbox" [checked]="isChecked">
<input type="text" [(ngModel)]="prenom">
Bonjour {{prenom}}



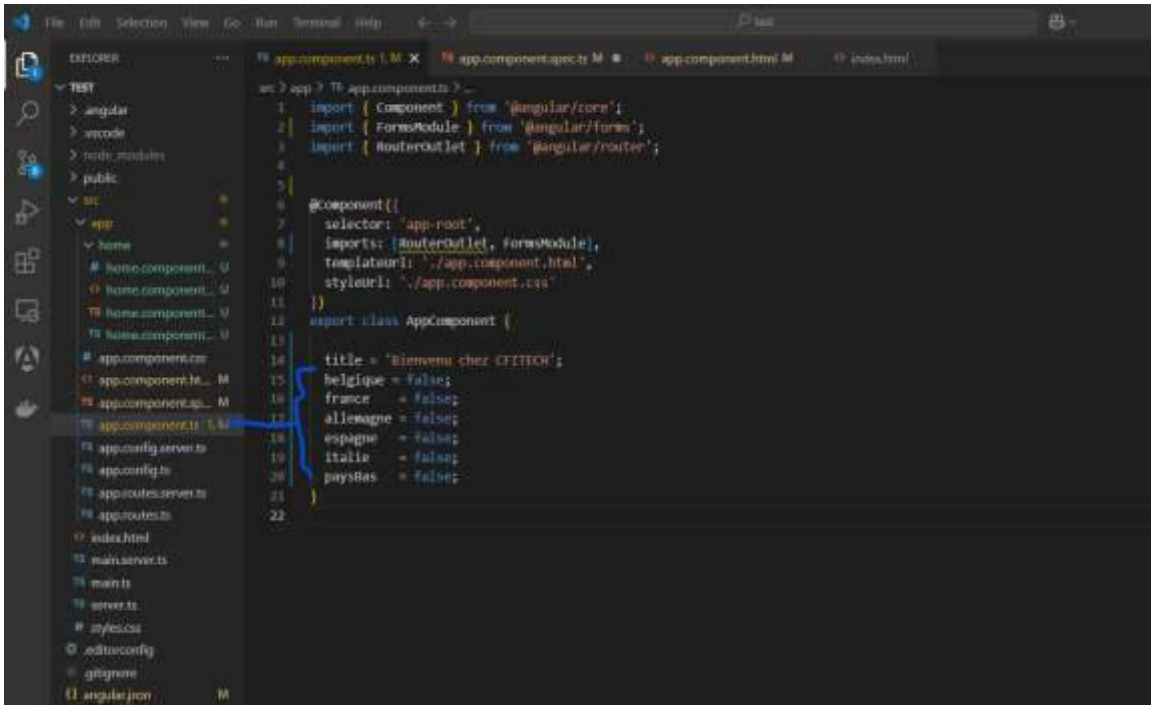
➔ Un autre exemple de comment lier un champ texte à une variable `email` avec `[(ngModel)]` :



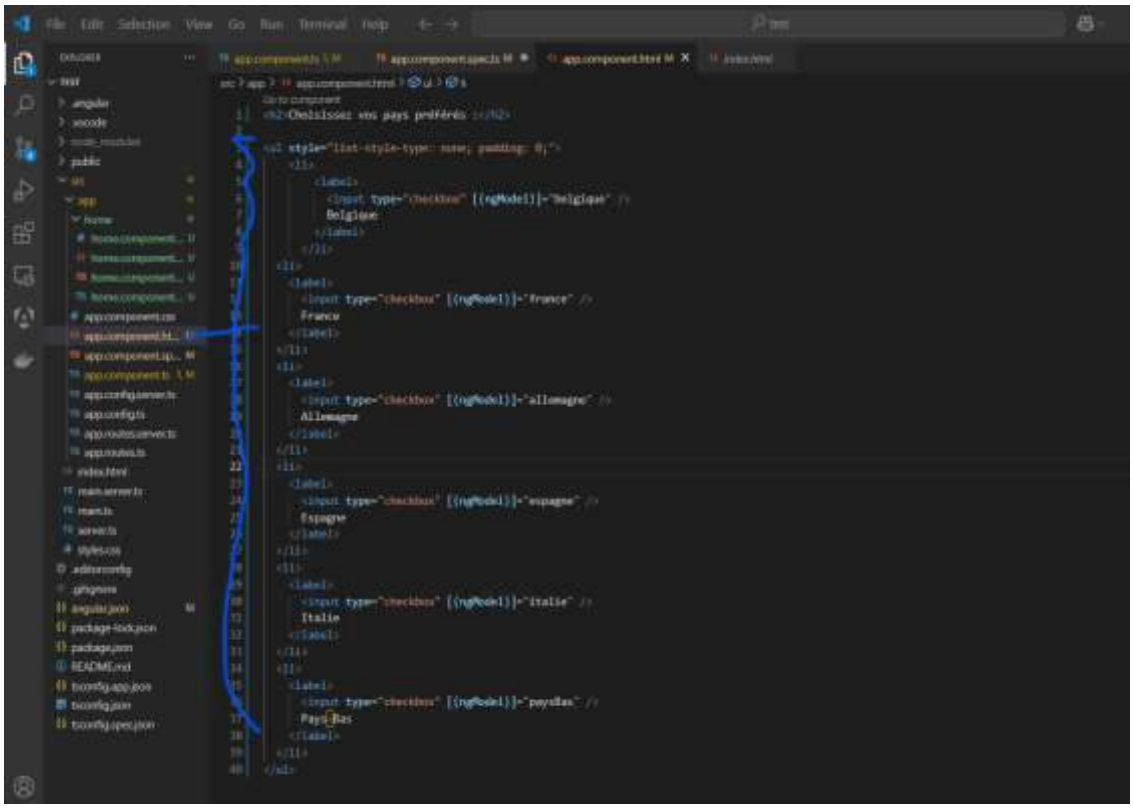




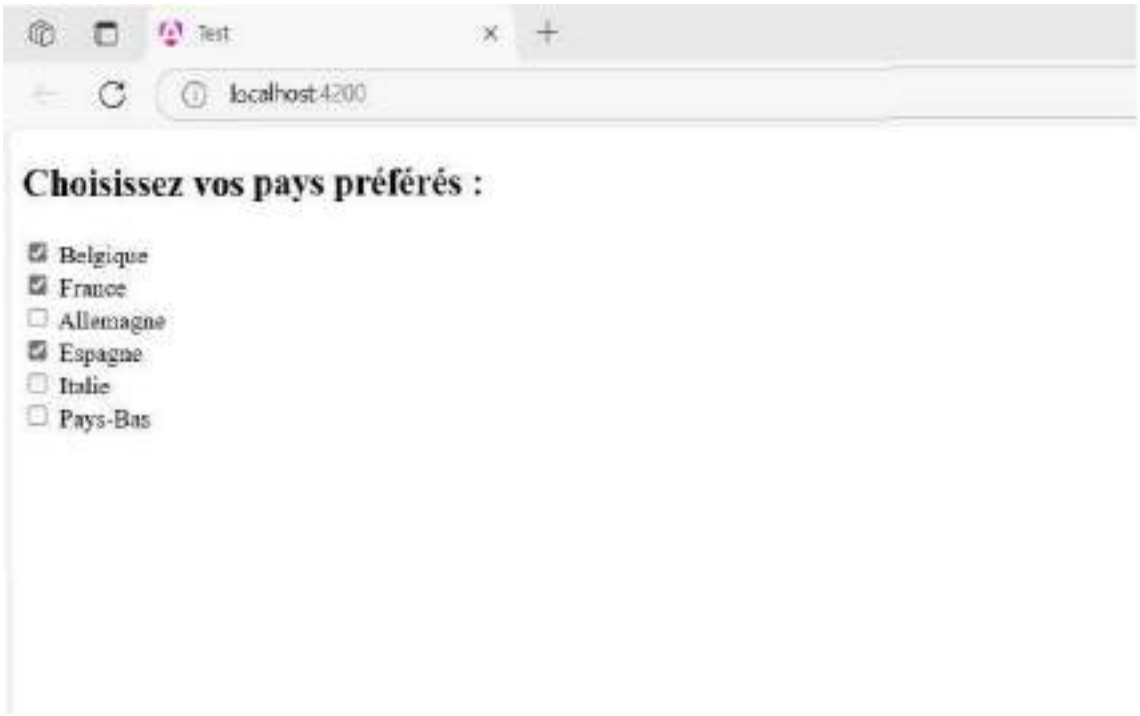
➔ Un exemple complet montrant comment utiliser un **checkbox** avec `[(ngModel)]` pour une liaison bidirectionnelle :



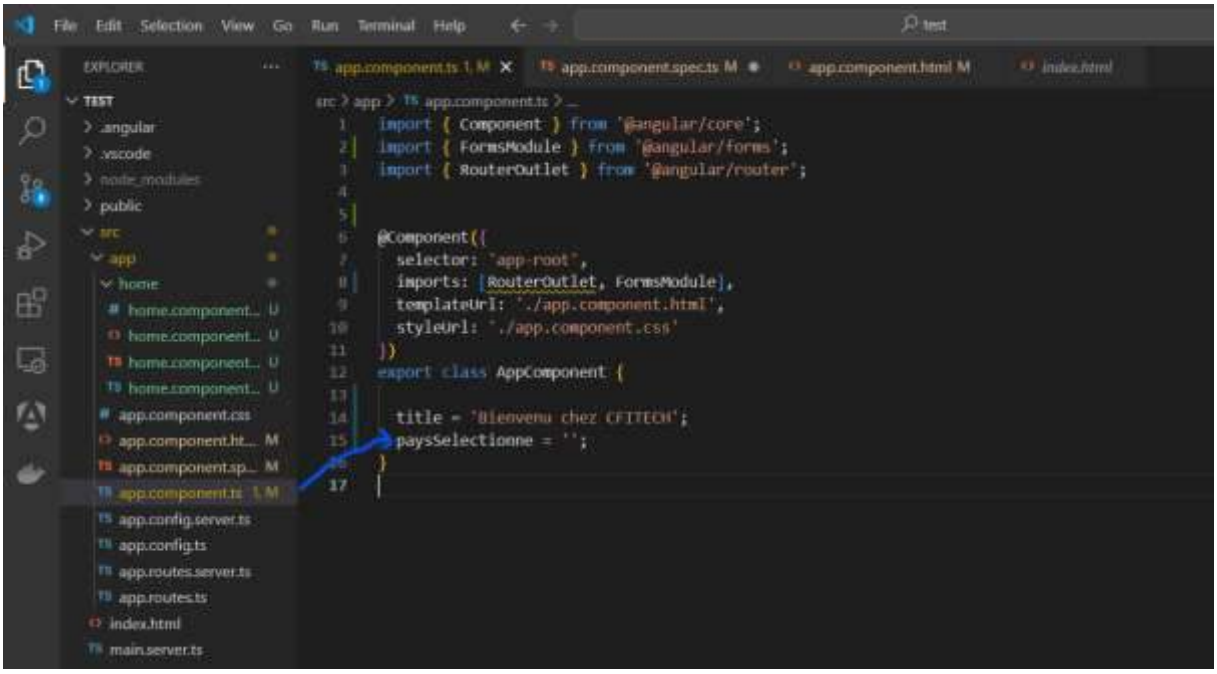
```ts
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';


@Component({
  selector: 'app-root',
  imports: [RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {

  title = 'Bienvenu chez CFITECH';
  belgique = false;
  france    = false;
  allemagne = false;
  espagne   = false;
  italie    = false;
  paysBas   = false;
}
```

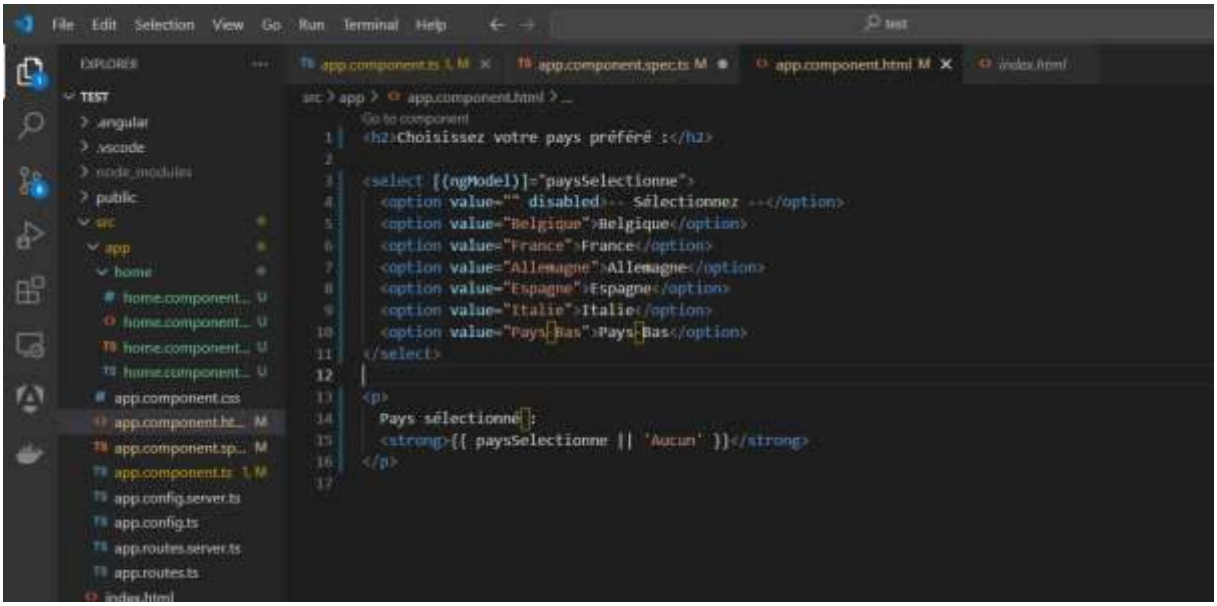


```html
<h2>Choisissez vos pays préférés :</h2>

<ul style="list-style-type: none; padding: 0;">
  <li>
    <label>
      <input type="checkbox" [(ngModel)]="belgique" />
      Belgique
    </label>
  </li>
  <li>
    <label>
      <input type="checkbox" [(ngModel)]="france" />
      France
    </label>
  </li>
  <li>
    <label>
      <input type="checkbox" [(ngModel)]="allemagne" />
      Allemagne
    </label>
  </li>
  <li>
    <label>
      <input type="checkbox" [(ngModel)]="espagne" />
      Espagne
    </label>
  </li>
  <li>
    <label>
      <input type="checkbox" [(ngModel)]="italie" />
      Italie
    </label>
  </li>
  <li>
    <label>
      <input type="checkbox" [(ngModel)]="paysBas" />
      Pays-Bas
    </label>
  </li>
</ul>
```



➔ Un exemple qui montre **une liste déroulante** pour sélectionner un seul pays parmi une liste disponible, en utilisant `[(ngModel)]`


File Edit Selection View Go Run Terminal Help
test
EXPLORER
TEST
.angular
.vscode
node_modules
public
src
app
home
app.component.css
app.component.ts
app.config.server.ts
app.config.ts
app.routes.server.ts
app.routes.ts
index.html
main.server.ts
app.component.ts M
app.component.spec.ts M
app.component.html M
index.html
src > app > app.component.ts > ...
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
@Component({
selector: 'app-root',
imports: [RouterOutlet, FormsModule],
templateUrl: './app.component.html',
styleUrl: './app.component.css'
})
export class AppComponent {
title = 'Bienvenu chez CFITECH';
paysSelectionne = '';
}


File Edit Selection View Go Run Terminal Help
test
EXPLORER
TEST
.angular
.vscode
node_modules
public
src
app
home
app.component.css
app.component.ts
app.config.server.ts
app.config.ts
app.routes.server.ts
app.routes.ts
index.html
app.component.ts M
app.component.spec.ts M
app.component.html M
index.html
src > app > app.component.html > ...
Go to component
<h2>Choisissez votre pays préféré :</h2>
<select [(ngModel)]="paysSelectionne">
<option value="" disabled>-- Sélectionnez --</option>
<option value="Belgique">Belgique</option>
<option value="France">France</option>
<option value="Allemagne">Allemagne</option>
<option value="Espagne">Espagne</option>
<option value="Italie">Italie</option>
<option value="Pays-Bas">Pays-Bas</option>
</select>

Pays sélectionné :
{{ paysSelectionne || 'Aucun' }}

Exercices :

Exercice 1 — Interpolation simple

Dans le composant, une propriété `message = "Bonjour Jean"` est définie. Affichez ce message dans une balise `<h2>`

`Rep :`

<h2>{{ message }}</h2>

Exercice 2 — Interpolation avec expression

Une variable `age = 25` est déclarée dans le composant. Affichez dans une balise `<p>` : `Vous avez 25 ans.` (le nombre doit venir de la variable)

Rep :

Vous avez {{ age }} ans.

Exercice 3 — Property binding (src)

Une variable

`logo = "https://angular.io/assets/images/logos/angular/angular.png"` est définie. Affichez l'image à l'aide de `property binding`.

Rep :

<img [src]="logo" alt="Logo Angular" width="150">

Exercice 4 — Property binding (disabled)

Une variable `isDisabled = true` est définie. Utilisez `[disabled]` pour désactiver un bouton quand cette valeur est vraie.

Rep :

<button [disabled]="isDisabled">Envoyer</button>

Exercice 5 — Event binding (click)

Créez un bouton qui déclenche la méthode `saluer()` du composant. Cette méthode doit afficher `console.log("Bonjour !")`.

Rep :

<!-- HTML -->

<button (click)="saluer()">Cliquez-moi</button>

<!-- TypeScript -->

```
saluer() {
  console.log("Bonjour !");
}
```

Exercice 6 — Event binding + Interpolation

Une variable `message = ""`. En cliquant sur un bouton, changez-la en `"Bienvenue !"`, et affichez-la.

Rep :

<!-- HTML -->

<button (click)="message = 'Bienvenue !'">Afficher message</button>

{{ message }}

Exercice 7 — Two-Way Binding (ngModel)

Une variable `nom = ''`. Faites un champ texte lié à `nom`, et affichez en direct : `Bonjour [nom].`

Rep :

Une variable `nom = ''`. Faites un champ texte lié à `nom`, et affichez en direct : `Bonjour [nom].`

Rep :

<input [(ngModel)]="nom" placeholder="Entrez votre nom">

Bonjour {{ nom }}

!! N'oubliez pas `FormsModule`

Exercice 8 — Two-Way Binding avec checkbox

Une variable `accept = false`. Liez-la à une case à cocher, et affichez `"Accepté"` ou `"Refusé"` selon l'état.

Rep :

Html :

<input type="checkbox" [(ngModel)]="accept"> Accepter les termes

{{ accept ? 'Accepté' : 'Refusé' }}

Ts :

```
export class AppComponent {
  title = 'Bienvenue Chez CFITECH';
  accept='';
}
```

Exercice 9 — Application des 4 types ensemble

Une variable `nom = ''`. Affichez un bouton `"Continuer"` **uniquement si le champ `nom` est rempli.**

Créez un mini-formulaire avec :

- un champ texte lié à `prenom`
- un bouton pour valider (désactivé si le champ est vide)
- un message `Bonjour [prenom]` après clic

rep :

html

<input [(ngModel)]="prenom" placeholder="Votre prénom">

<button [disabled]="prenom === '"" (click)="saluer()">Valider</button>

{{ message }}

Ts

prenom = '';

message = '';

saluer() {

 this.message = 'Bonjour ' + this.prenom;

}

Exercice 10 — Interpolation + expression conditionnelle

Déclarer une variable `score` représentant la note obtenue par l'apprenant.

Si le score est **supérieur ou égal à 10**, le programme doit afficher :
"Réussi"

Sinon, le programme doit afficher :
"Échoué"

Rep :

Html :

```
<h1>Data Binding</h1>
<p>{{score >=10 ? 'Reussi':'Echoué'}}</p>
```

Ts :

```
export class AppComponent {
title = 'Bienvenue Chez CFITECH';
score=12;
}
```

Exercice 11 — Property Binding avec input

Déclarez une variable JavaScript `readOnlyMode` qui peut être soit `true`, soit `false`.
Utilisez cette variable pour contrôler dynamiquement l'attribut `readonly` d'un champ `<input>` HTML.

- Si `readOnlyMode` est `true` → le champ doit être en lecture seule.
- Si `readOnlyMode` est `false` → l'utilisateur doit pouvoir modifier le champ.

Rep :

Ts :

```
export class AppComponent {
title = 'Bienvenue Chez CFITECH';
readOnlyMode: boolean = true;
}
```

Hmtl :

```
<h1>Data Binding</h1>
<label for="username">Nom d'utilisateur :</label>
<input id="username" type="text" [readonly]="readOnlyMode"
value="JeanAimable123">
```

Exercice 12 — Two-way Binding avec transformation

Liez un input à `ville`, et affichez-la en **majuscules** à côté.

Ts :

```
export class AppComponent {
title = 'Bienvenue Chez CFITECH';
ville='';
}
```

Hmtl :

```
<h1>Data Binding</h1>
<input [(ngModel)]="ville">
<p>{{ ville.toUpperCase() }}</p>
```

Exercice 13 — Bouton activé uniquement si case cochée

Une case à cocher liée à `conditionsAcceptees`. Le bouton "Continuer" doit être activé seulement si `true`

`Ts :`

```
export class AppComponent {
title = 'Bienvenue Chez CFITECH';
conditionsAcceptees=false
}
```

Html :

```
<h1>Data Binding</h1>
<input type="checkbox" [(ngModel)]="conditionsAcceptees"> J'accepte
<button [disabled]="!conditionsAcceptees">Continuer</button>
```

⇨ **Les Signals dans Angular**

Les Signals sont le nouveau modèle réactif d'Angular moderne. Visuellement, ils produisent souvent le même résultat qu'une variable classique, ce qui explique pourquoi leur intérêt n'est pas toujours évident au début. Leur véritable avantage apparaît dans les applications professionnelles comportant beaucoup de composants et de données. Avec une variable classique, Angular doit vérifier une grande partie de l'application lorsqu'une donnée change. Avec un Signal, Angular sait immédiatement quelle donnée a changé et quelles parties de l'interface doivent être mises à jour. Les Signals permettent ainsi d'obtenir des applications plus performantes, plus réactives et plus faciles à maintenir. C'est pour cette raison qu'ils sont aujourd'hui une fonctionnalité centrale d'Angular 21

Les Signals ont été introduits pour la première fois dans Angular Angular 16, publié en 2023.

1. Introduction : Pourquoi Angular a créé les Signals ?

Pendant de nombreuses années, Angular utilisait principalement son système de détection de changements (Change Detection) pour mettre à jour l'interface lorsqu'une donnée changeait.

Par exemple :

```
nom = 'Ali';

changerNom() {
  this.nom = 'Sara';
}
```

```
<p>{{ nom }}</p>

<button (click)="changerNom()">
  Changer
</button>
```

Lorsque l'utilisateur clique sur le bouton :

Ali
↓
Sara

Tout fonctionne parfaitement.

Alors une question se pose :

Pourquoi Angular a-t-il créé les Signals si cela fonctionnait déjà ?

La réponse est simple :

Sur les petites applications, le système classique fonctionne très bien.

Mais dans les grosses applications professionnelles contenant des centaines de composants et des milliers de données, Angular devait effectuer beaucoup de vérifications inutiles.

Les Signals ont été créés pour rendre Angular plus performant et plus intelligent.

2. Qu'est-ce qu'un Signal ?

Définition

Un Signal est une variable réactive qui informe automatiquement Angular lorsqu'elle change.

Une variable classique stocke une valeur.

Un Signal stocke une valeur ET notifie Angular lorsqu'elle est modifiée.

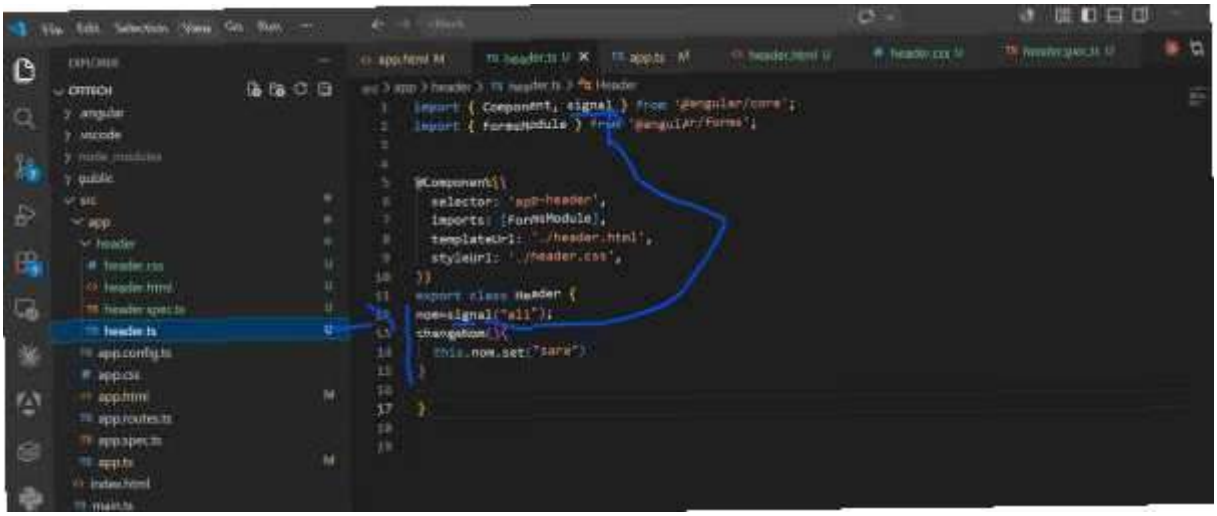
3. Pourquoi les Signals semblent-ils produire le même résultat qu'une variable classique ?

Dans une petite application, une variable classique et un Signal donnent souvent le même résultat visuel : les données s'affichent correctement et les modifications sont immédiatement visibles à l'écran. Il est donc légitime de se demander quel est l'intérêt des Signals. En réalité, leur principal avantage ne concerne pas l'affichage, mais la manière dont Angular détecte et gère les changements en arrière-plan. Les bénéfices des Signals deviennent particulièrement importants dans les applications de grande taille, où ils permettent à Angular d'être plus précis et plus performant dans la mise à jour de l'interface.

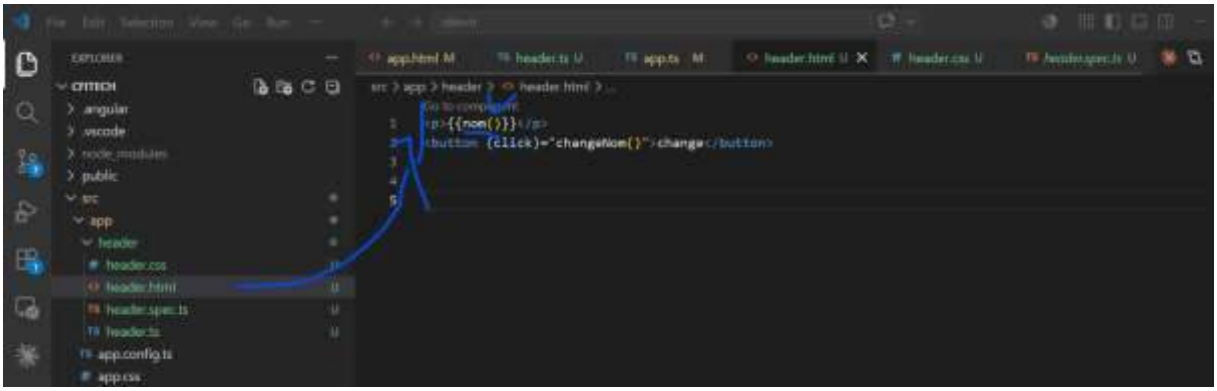
Sans Signal

```
nom = 'Ali';

changerNom() {
 this.nom = 'Sara';
}

<p>{{ nom }}</p>

<button (click)="changerNom()">
 Changer
</button>
```

Avec Signal



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

nom = signal('Ali');

changerNom() {
 this.nom.set('Sara');
}
```

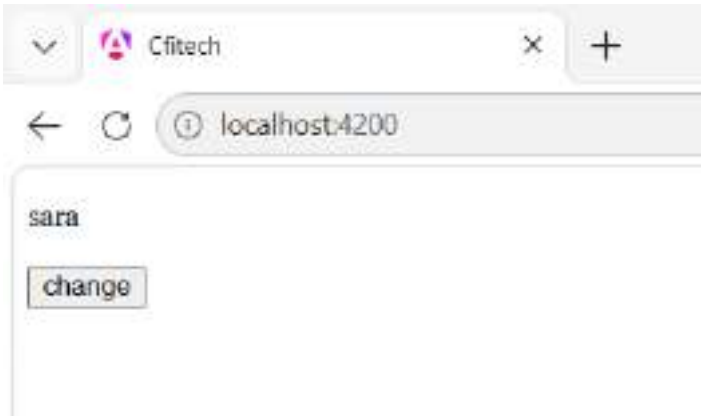


{{ nom() }} <!--() : important-->

<button (click)="changerNom()">
  Changer
</button>

Résultat visuel

Dans les deux cas :



Ali
↓ clic
Sara

4. **Fonctionnement sans Signal**

Imaginons une application CFITECH contenant :

- 50 composants
- 100 formulaires
- 500 données affichées

Lorsque cette ligne est exécutée :

nom = 'Sara';

Angular déclenche son système de détection de changements.

Il se dit :

"Quelque chose a changé dans l'application. Je vais vérifier ce qui doit être mis à jour."

Angular va donc parcourir une grande partie de l'application pour vérifier les données.

Cela fonctionne très bien.

Mais cela représente beaucoup de travail.

5. Fonctionnement avec Signal

Lorsque cette ligne est exécutée :

nom.set('Sara');

Angular sait immédiatement :

"Le Signal nom vient de changer."

Et il sait également :

"Voici exactement les éléments qui utilisent ce Signal."

Angular met alors à jour uniquement ce qui dépend de ce Signal.

6. Analogie

Imaginez une école contenant 500 stagiaires.

Ancien système

Le directeur annonce :

"Quelque chose a changé."

Tous les formateurs doivent vérifier leurs listes.

Même ceux qui ne sont pas concernés.

Système avec Signals

Le directeur annonce :

"Ali a changé de groupe."

Seul le formateur concerné reçoit l'information.

Les autres continuent leur travail.

Conclusion

Les Signals permettent à Angular de travailler de manière beaucoup plus ciblée.

7. Exemple concret

Supposons :

```
nom = signal('Ali');
age = signal(25);
ville = signal('Bruxelles');
```

HTML :

```
<p>{{ nom() }}</p>
<p>{{ age() }}</p>
<p>{{ ville() }}</p>
```

Vous modifiez uniquement :

nom.set('Sara');

Angular sait :

v mettre à jour le nom

x ne pas toucher à l'âge

x ne pas toucher à la ville

8. Lecture d'un Signal

Une variable classique :

nom

Un Signal :

nom()

Exemple

{{ nom() }}

9. Modifier un Signal avec set()

Définition

set() remplace complètement la valeur.

Exemple

nom = signal('Ali');

changerNom() {
  this.nom.set('Sara');
}

Résultat

Avant :

Ali

Après :

Sara

10. Modifier un Signal avec update()

Définition

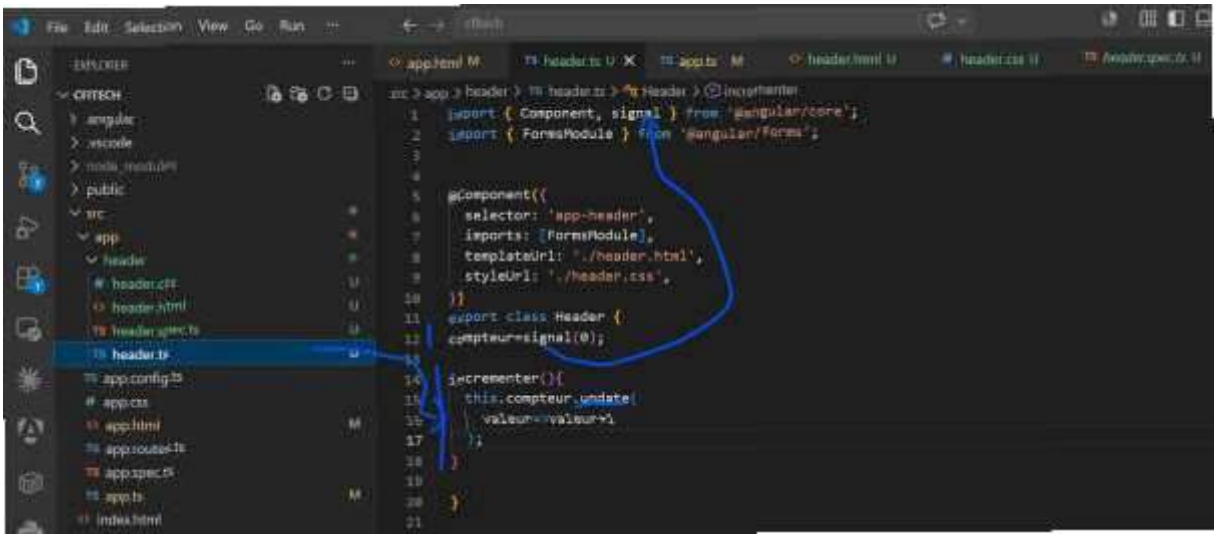
update() modifie la valeur en utilisant l'ancienne valeur.

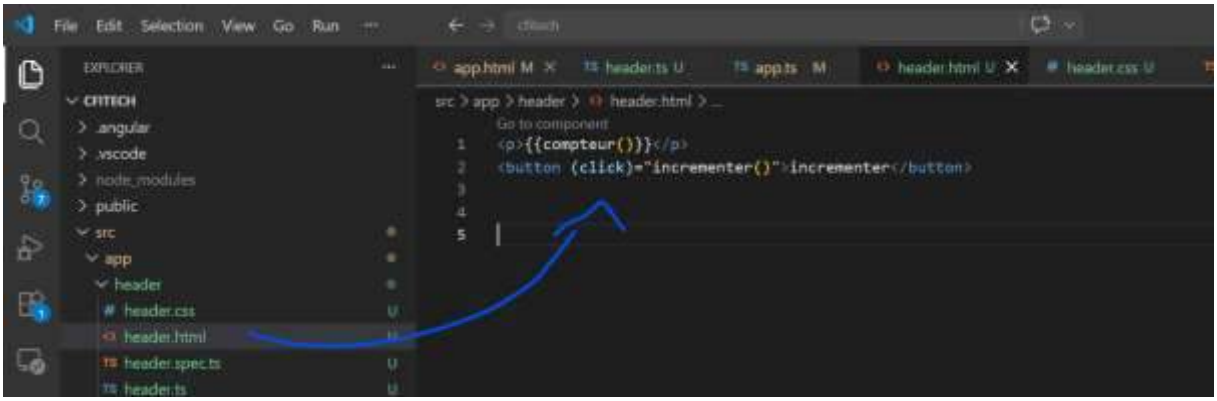
Exemple

```
compteur = signal(0);

incrementer() {
 this.compteur.update(
  valeur => valeur + 1
 );
}
```

ou

```
stock.update(
 s => s - 1
);
```





Résultat

0
1
2
3
4

...

11.Différence entre set() et update()

set()	**update()**
remplace la valeur	modifie à partir de l'ancienne valeur
valeur connue	calcul à partir de l'ancienne valeur

12. Autre Exemple

Compteur de stagiaires :

```
nbStagiaires = signal(0);

ajouterStagiaire() {
  this.nbStagiaires.update(
    n => n + 1
  );
}
```

HTML :

```
<p>
Nombre de stagiaires :
{{ nbStagiaires() }}
</p>

<button (click)="ajouterStagiaire()">
Ajouter
</button>
```

Exercice 1

Créer un Signal contenant :

CFITECH et l'afficher.

Correction

```
centre = signal('CFITECH');

<p>{{ centre() }}</p>
```

Exercice 2

Créer un bouton permettant de changer :

Débutant en Intermédiaire

Correction

```
niveau = signal('Débutant');

changer() {
 this.niveau.set('Intermédiaire');
}

<p>{{ niveau() }}</p>

<button (click)="changer()">
Changer
</button>
```

Exercice 3

Créer un compteur.

Correction

```
compteur = signal(0);

incrementer() {
 this.compteur.update(
   c => c + 1
 );
}

<p>{{ compteur() }}</p>

<button (click)="incrementer()">
+
</button>
```

Exercice 4 (Projet CFITECH)

Créer un compteur de stagiaires.

Correction

```
nbStagiaires = signal(0);

ajouterStagiaire() {
 this.nbStagiaires.update(
   n => n + 1
```

```
  );
}
```

```
<p>
Nombre de stagiaires :
{{ nbStagiaires() }}
</p>

<button (click)="ajouterStagiaire()">
Ajouter un stagiaire
</button>
```

Mini Projet : Formulaire Stagiaire

Fonctionnalités :

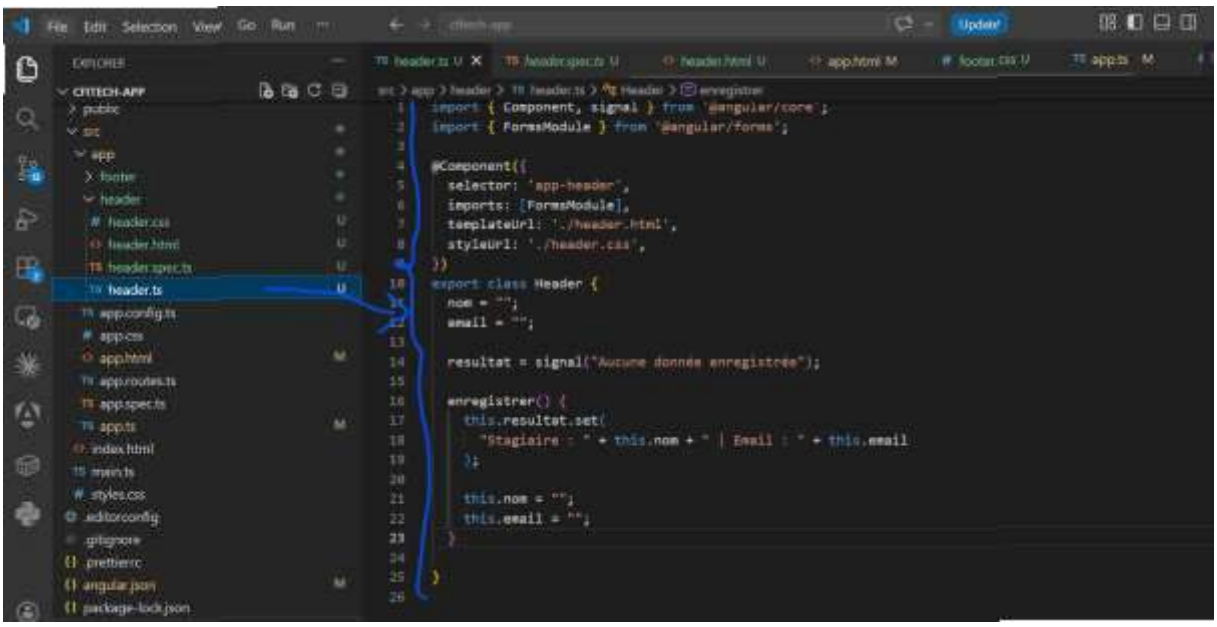
- Nom
- Email
- Bouton
- Résumé affiché

Rep :

1. TypeScript

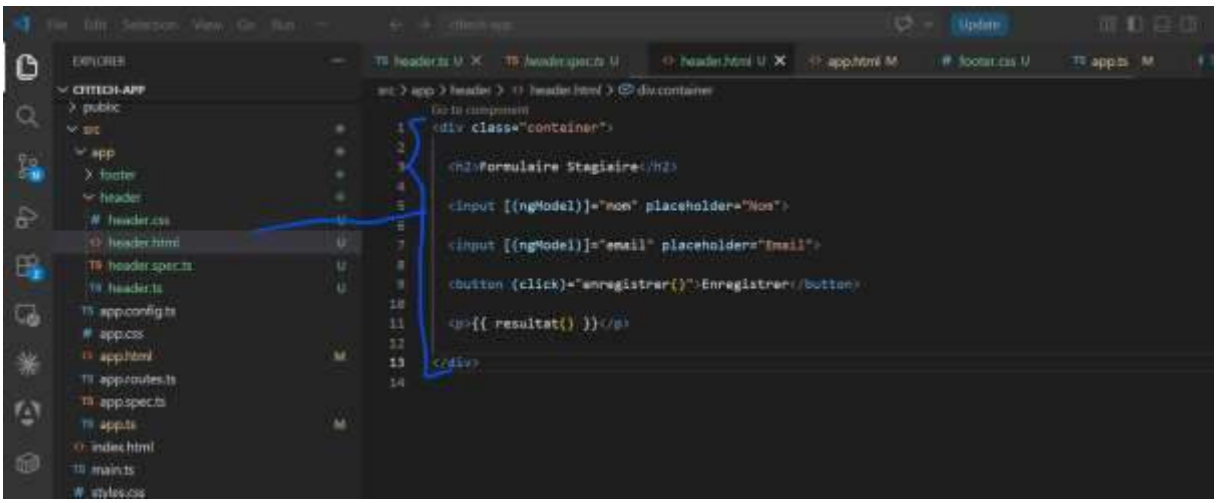
```
import { Component, signal } from '@angular/core';

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



@Component({

 selector: 'app-root',

 standalone: true,

 imports: [FormsModule],

 templateUrl: './app.html',

 styleUrls: ['./app.css']

})

export class App {
```

```
  nom = "";

  email = "";


  resultat = signal("Aucune donnée enregistrée");


  enregistrer() {

    this.resultat.set(

      "Stagiaire : " + this.nom + " | Email : " + this.email

    );


    this.nom = "";

    this.email = "";

  }


}
```



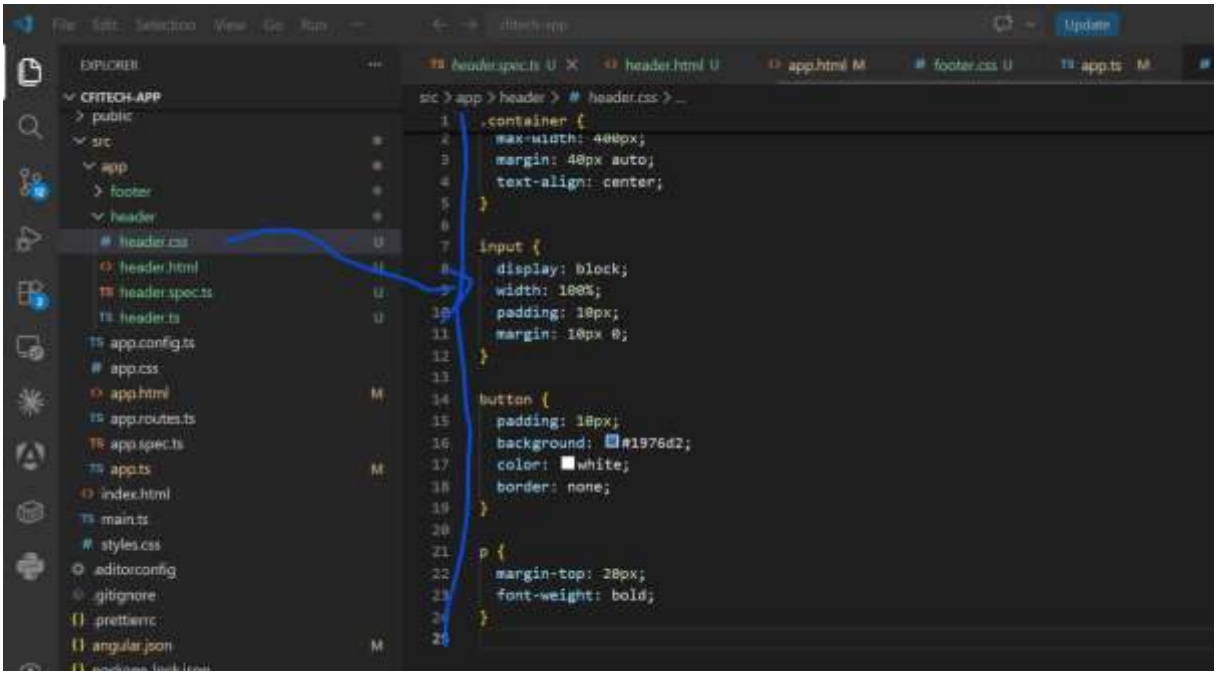
2.HTML

```
<div class="container">

 <h2>Formulaire Stagiaire</h2>

 <input [(ngModel)]="nom" placeholder="Nom">

 <input [(ngModel)]="email" placeholder="Email">

 <button (click)="enregistrer()">Enregistrer</button>

 <p>{{ resultat() }}</p>

</div>
```



3. CSS

```
.container {
 max-width: 400px;
 margin: 40px auto;
 text-align: center;
}

input {
 display: block;
 width: 100%;
 padding: 10px;
 margin: 10px 0;
}

button {
 padding: 10px;
 background: #1976d2;
 color: white;
 border: none;
}

p {
 margin-top: 20px;
 font-weight: bold;
}
```



En résumé :
À ce stade, nous sommes capables de mettre en place un formulaire interactif permettant de saisir et d'afficher des données en temps réel grâce au data binding et aux signals. Toutefois, la gestion complète des données (comme l'affichage de listes ou le stockage multiple) nécessite l'utilisation de directives, qui seront abordées dans le chapitre suivant.

Two-way binding

[(ngModel)]="nom"

saisir → variable mise à jour

Event binding

(click)="enregistrer()"

clic → action

Signal

resultat = signal(...)

affichage automatique

L’utilisateur remplit :

Nom: Ali

Email: ali@mail.com

Clique :

Affichage :

Stagiaire : Ali | Email : ali@mail.com

Chapitre 6 : Control Flow & Directives (Angular moderne)

Objectif

- Manipuler l’affichage dynamiquement
- Créer des interfaces réactives
- Remplacer les anciennes directives (*ngIf, *ngFor,*ngSwitch)

1. Introduction aux nouveaux Control Flow (*flux de contrôle*)

Définition

Le **Control Flow** permet de contrôler ce qui s’affiche dans le HTML en fonction des données.

“Le Control Flow permet de rendre l’interface intelligente.”

2. @if (Condition)

Définition

@if permet d’afficher un élément **seulement si une condition est vraie**

Syntaxe

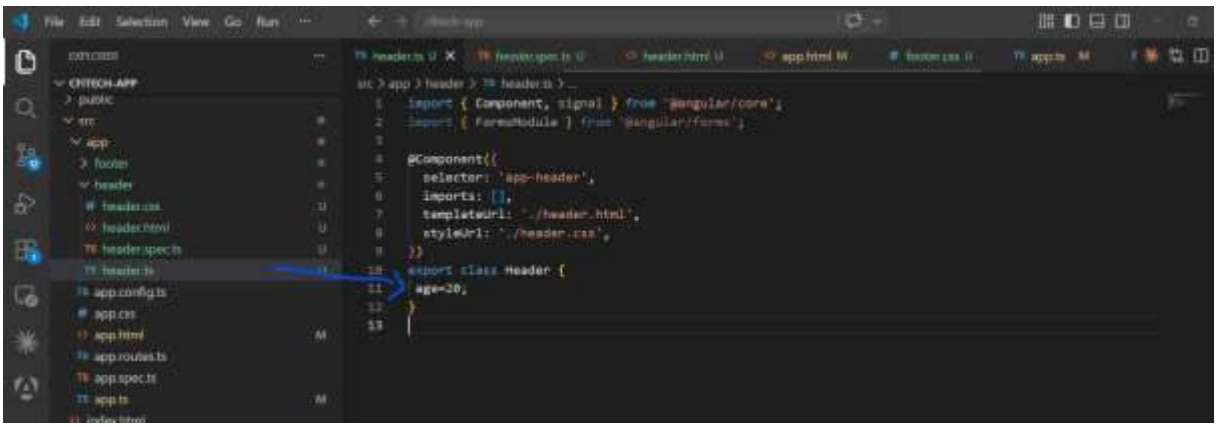
```
@if (condition) {
  <p>Contenu affiché</p>
}
```

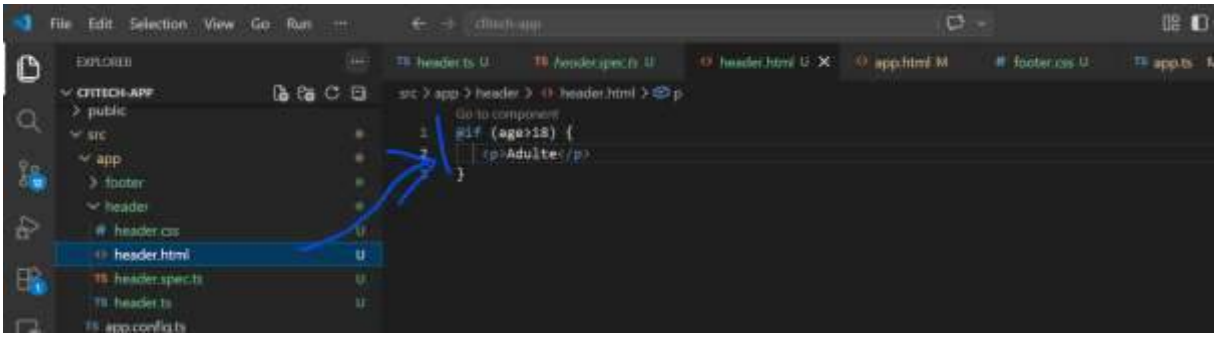
Ex

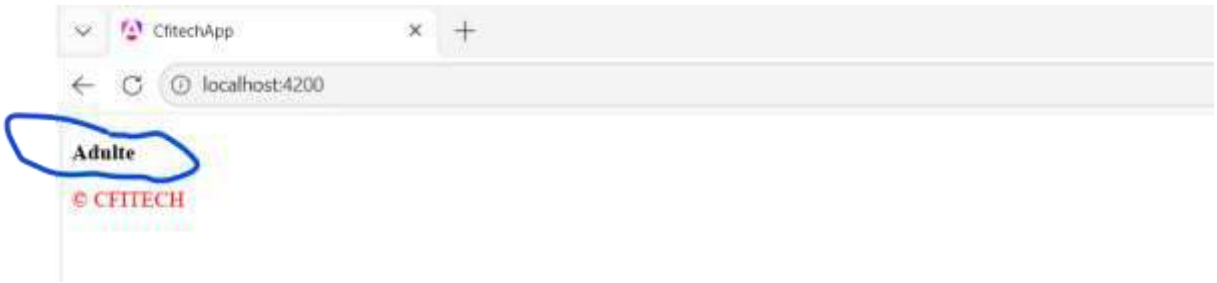
```
age = 20;

@if (age >= 18) {
  <p>Adulte</p>
}
```

affiche :

Adulte





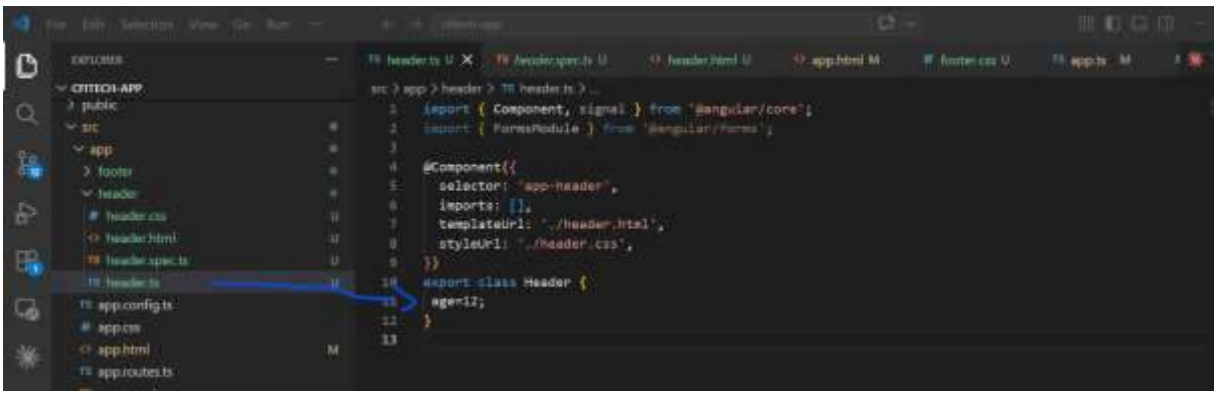


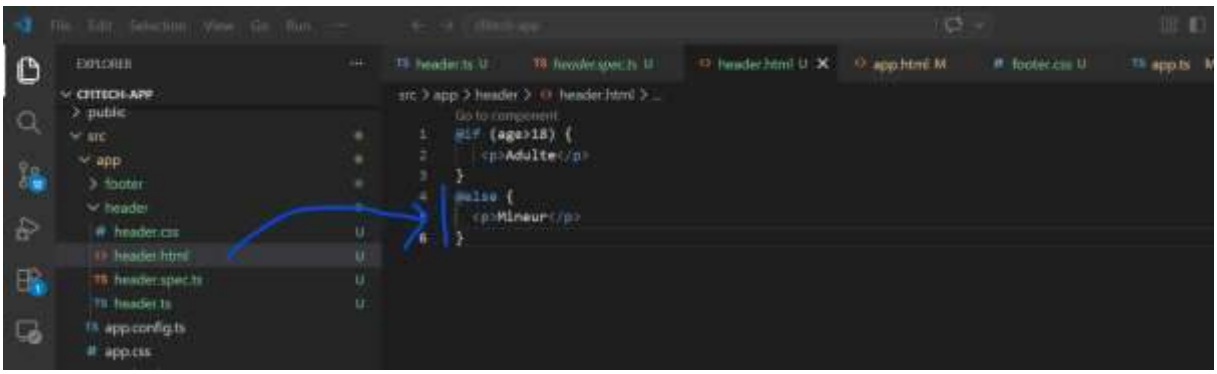
Avec @else

```
@if (age >= 18) {
  <p>Adulte</p>
} @else {
```

```
 <p>Mineur</p>
}
```

@if = condition





Exercice 1

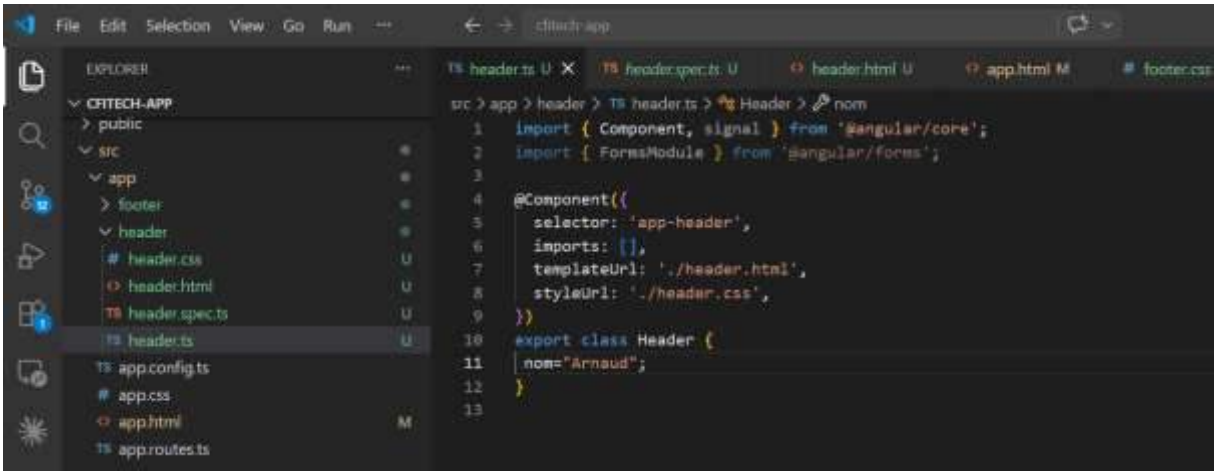
TS :

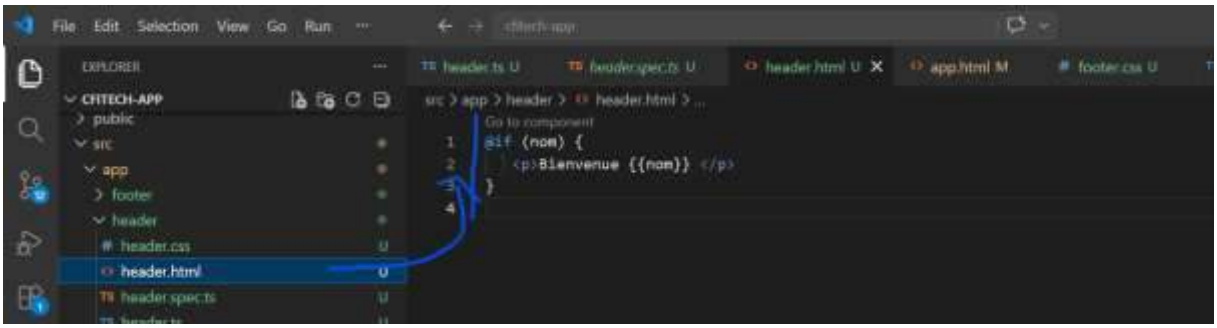
nom = "Arnaud";

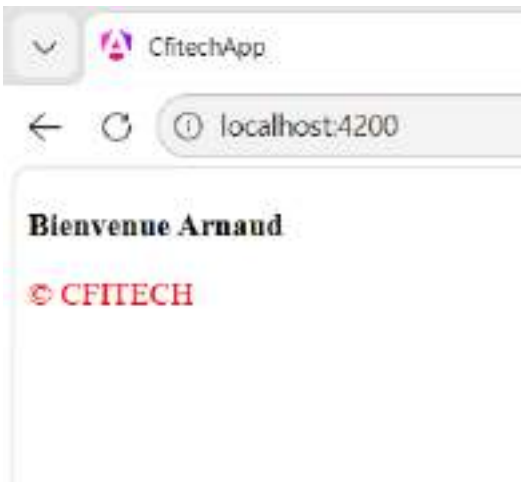
Afficher “Bienvenue Arnaud” seulement si nom existe

Correction

```
@if (nom) {
 <p>Bienvenue {{ nom }}</p>
}
```







Exercice 2

TS :

score = 8;

Si score ≥ 10 → “Validé” sinon “Échec”

Correction

```
@if (score >= 10) {
  <p>Validé</p>
} @else {
  <p>Échec</p>
}
```

3. @for (Boucle)

Définition

@for permet de répéter un élément pour chaque valeur d'une liste

Syntaxe

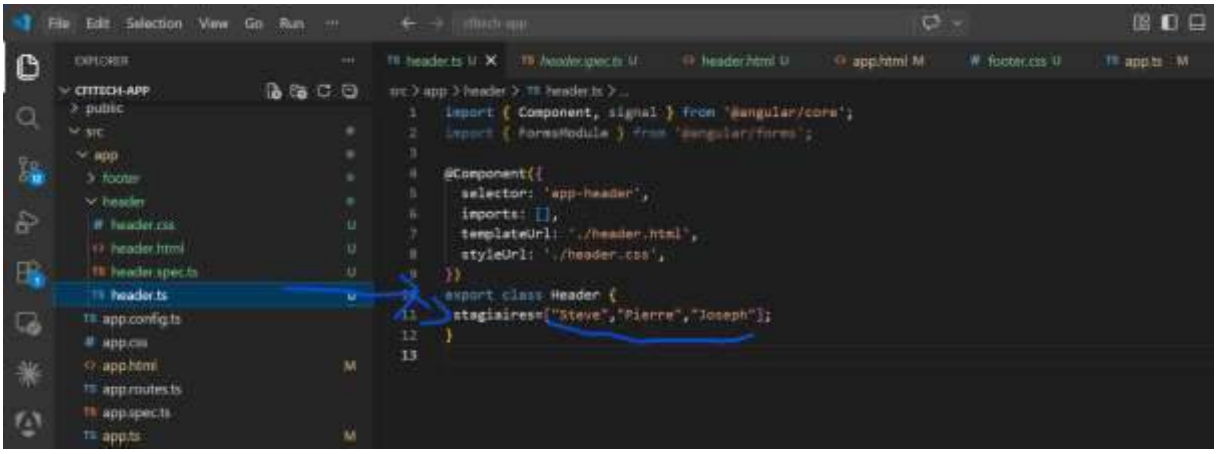
```
@for (item of liste; track item) {
  <p>{{ item }}</p>
}
```

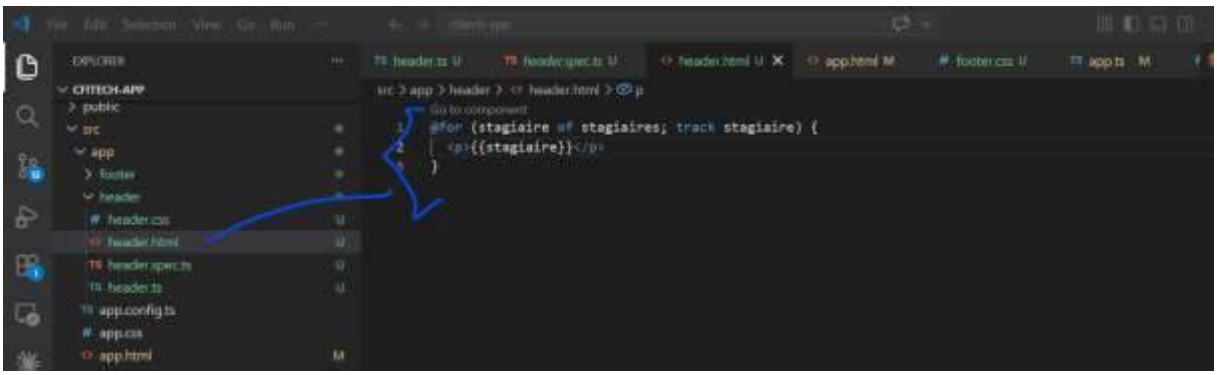
Exemple

```
stagiaires = ["Ali", "Sara", "Omar"];

@for (s of stagiaires; track s) {
  <p>{{ s }}</p>
}
```

Affichage :

Ali
Sara
Omar





track = “suivre / identifier”

“track item” = “suivre chaque élément (pour l’identifier)”

track améliore la performance

export class Header {

stagiaires=["ali","patrick","pierre"]

}

```
export class Header {

stagiaires=["ali","patrick","pierre"]



}
```

1. Pourquoi Angular a ajouté track ?

Lorsque Angular affiche une liste avec @for, il crée des éléments HTML dans le DOM.

Exemple :

```
stagiaires = [
  "Ali",
  "Sara",
  "Omar"
];

@for (s of stagiaires; track s) {
  <p>{{ s }}</p>
}
```

Angular crée :

```
<p>Ali</p>
<p>Sara</p>
<p>Omar</p>
```

Jusqu'ici tout semble simple.

Mais lorsqu'une liste change, Angular doit répondre à une question importante :

Quels éléments ont réellement changé ?

C'est précisément pour répondre à cette question que track existe.

2. Le rôle de track

Définition

Le mot-clé track permet à Angular d'identifier chaque élément d'une liste de manière unique.

Grâce à cette identification, Angular sait exactement :

- quels éléments existent déjà ;
- quels éléments ont été ajoutés ;
- quels éléments ont été supprimés ;
- quels éléments ont été modifiés.

3. Une confusion fréquente

Beaucoup pensent que :

Angular crée automatiquement un identifiant lorsqu'on utilise track.

Ce n'est pas le cas.

Angular ne crée aucun identifiant.

C'est le développeur qui indique à Angular quelle valeur doit servir d'identifiant.

4. Exemple avec une liste de chaînes

```
stagiaires = [
 "Ali",
 "Sara",
 "Omar"
];

@for (s of stagiaires; track s) {
 <p>{{ s }}</p>
}
```

Ici :

Ali
Sara
Omar

Angular considère :

Ali → identifiant = Ali
Sara → identifiant = Sara
Omar → identifiant = Omar

La valeur elle-même devient l'identifiant.

5. Que se passe-t-il lorsqu'on ajoute un élément ?

Liste initiale :

Ali
Sara
Omar

Nouvelle liste :

Ali
Sara
Omar
Yasmine

Grâce à :

track s

Angular comprend :

Ali → existe déjà
Sara → existe déjà
Omar → existe déjà
Yasmine → nouvel élément

Il ajoute uniquement :

Yasmine

Cas où les valeurs sont uniques

```
stagiaires = [
  "Ali",
  "Sara",
  "Omar"
];
```

@for (s of stagiaires; track s)

Angular considère :

Ali → clé = "Ali"
Sara → clé = "Sara"
Omar → clé = "Omar"

Aucun problème.

Cas où deux valeurs sont identiques

```
stagiaires = [
  "Ali",
  "Sara",
  "Ali"
];
```

Avec :

@for (s of stagiaires; track s)

Angular voit :

Ali → clé = "Ali"
Sara → clé = "Sara"
Ali → clé = "Ali"

Et là il y a un problème :

Ali
Ali
↑
même clé

Angular ne peut plus identifier chaque élément de façon unique.

Tu risques d'obtenir une erreur du type :

Duplicate keys detected ou un comportement incorrect.

Pourquoi ?

Parce que track doit toujours fournir une valeur unique.

Angular a besoin de pouvoir dire :

Cet élément = celui-ci
Cet autre élément = celui-là

Si deux éléments ont la même clé :

Ali
Ali

Angular ne sait plus lequel est lequel.

Solution professionnelle

Au lieu de :

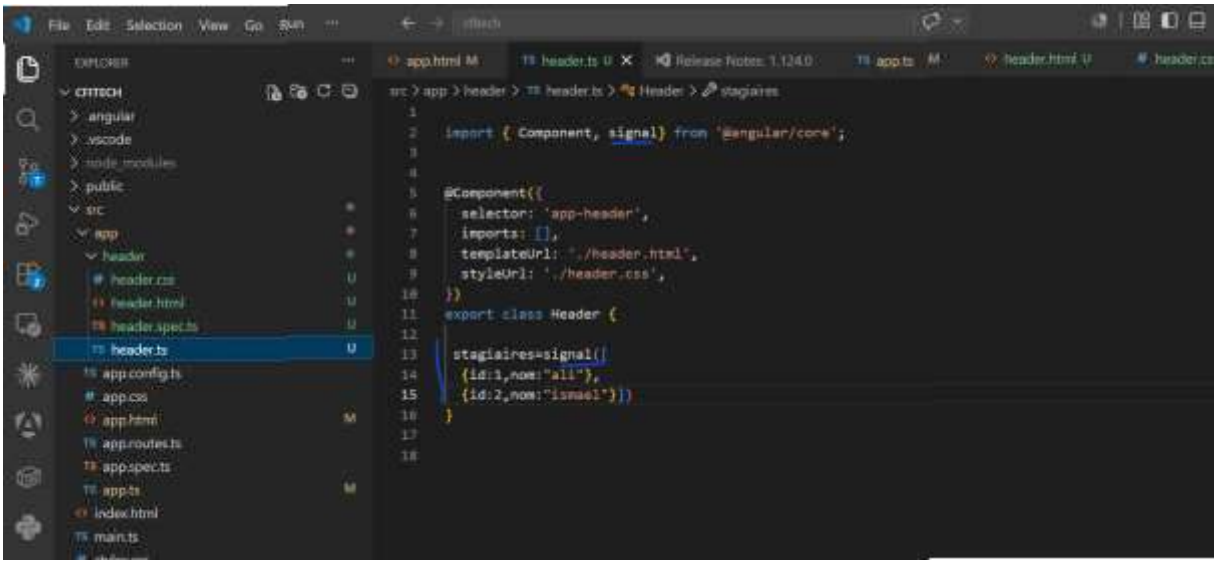
```
[
 "Ali",
 "Sara",
 "Ali"
]
```

on utilise généralement :

```
[
 { id: 1, nom: "Ali" },
 { id: 2, nom: "Sara" },
 { id: 3, nom: "Ali" }
]
```

Puis :

```
@for (s of stagiaires; track s.id) {
 <p>{{ s.nom }}</p>
}
```



Angular voit :

id 1 → Ali
id 2 → Sara
id 3 → Ali

Même si les noms sont identiques :

Ali
Ali

les identifiants sont différents :

1
3

et Angular peut les distinguer parfaitement.

Lorsque l'on utilise track s, Angular utilise directement la valeur de l'élément comme identifiant. Cette approche fonctionne uniquement si toutes les valeurs sont uniques. Si deux éléments possèdent la même valeur, Angular ne peut plus les distinguer correctement. C'est pourquoi, dans les applications professionnelles, on préfère généralement utiliser un identifiant unique comme id avec track s.id, afin que chaque élément soit reconnu de manière fiable même si plusieurs objets possèdent le même nom ou les mêmes données.

6. Pourquoi est-ce intéressant ?

Imaginons une application CFITECH contenant :

1000 stagiaires sans identification précise, Angular devrait analyser une grande partie de la liste.

Avec track, Angular sait immédiatement ce qui a changé.

Cela réduit les traitements inutiles.

7. Cas professionnel : utilisation d'objets

En entreprise, les listes contiennent rarement de simples chaînes.

On manipule souvent des objets.

Exemple :

```
stagiaires = [
 { id: 1, nom: "Ali" },
 { id: 2, nom: "Sara" },
 { id: 3, nom: "Omar" }
];
```

Affichage

```
@for (s of stagiaires; track s.id) {
  <p>{{ s.nom }}</p>
}
```

Angular utilise :

1
2
3

comme identifiants.

8. Pourquoi utiliser s.id ?

Supposons que :

Sara

devienne :

Sarah

Nouvelle liste :

```
[
 { id: 1, nom: "Ali" },
 { id: 2, nom: "Sarah" },
 { id: 3, nom: "Omar" }
]
```

Angular regarde uniquement :

id 1
id 2
id 3

Il comprend immédiatement :

Le stagiaire 2 existe toujours.
Seul son nom a changé.

Il met à jour uniquement cet élément.

9. Cas de suppression

Liste initiale :

1 Ali
2 Sara
3 Omar

Nouvelle liste :

1 Ali
3 Omar

Angular détecte immédiatement :

L'élément avec l'id 2 a disparu.

Il supprime uniquement cet élément du DOM.

10. Cas de réorganisation

Liste initiale :

1 Ali
2 Sara
3 Omar

Nouvelle liste :

3 Omar
1 Ali
2 Sara

Grâce à :

track s.id

Angular sait :

id 1 existe
id 2 existe
id 3 existe

Il déplace simplement les éléments.

Il ne les recrée pas.

11. Pourquoi éviter track $index ?

Beaucoup de gens écrivent :

@for (s of stagiaires; track $index)

Cela fonctionne.

Mais ce n'est généralement pas la meilleure solution.

Exemple

Liste :

0 Ali
1 Sara
2 Omar

On ajoute :

0 Yasmine
1 Ali
2 Sara
3 Omar

Tous les index changent.

Angular peut croire que plusieurs éléments ont été modifiés alors qu'ils existent déjà.

Avec un identifiant

track s.id

Les identifiants restent :

1 Ali
2 Sara
3 Omar
4 Yasmine

Angular retrouve immédiatement chaque élément.

12. Analogie CFITECH

Imagine une classe.

Chaque stagiaire possède un numéro d'inscription.

101 Ali
102 Sara
103 Omar

Le lendemain :

101 Ali
102 Sarah
103 Omar

Le numéro d'inscription n'a pas changé.

Le formateur comprend immédiatement :

Le stagiaire existe toujours.
Seul son nom a été modifié.

Le numéro d'inscription joue exactement le rôle de :

track s.id

Les bonnes pratiques

Pour une liste de chaînes

@for (ville of villes; track ville)

Pour une liste d'objets

@for (stagiaire of stagiaires; track stagiaire.id)

Pour une liste de produits

@for (produit of produits; track produit.id)

Pour une liste de clients

@for (client of clients; track client.id)

À éviter

@for (s of stagiaires; track $index)

sauf lorsque la liste est très simple et que son ordre ne changera jamais.

Exercice 1

Soit :

```
formations = [
  "Angular",
  "React",
  "Vue"
];
```

Écrire le @for.

Correction

```
@for (f of formations; track f) {
  <p>{{ f }}</p>
}
```

Exercice 2

Soit :

```
stagiaires = [
 { id: 1, nom: "Ali" },
 { id: 2, nom: "Sara" }
];
```

Afficher les noms.

Correction

```
@for (s of stagiaires; track s.id) {
  <p>{{ s.nom }}</p>
}
```

Exercice 3

Pourquoi cette solution est-elle moins recommandée ?

```
@for (s of stagiaires; track $index)
```

Réponse

Parce que l'index peut changer lorsqu'un élément est ajouté, supprimé ou déplacé. Angular risque alors de refaire davantage de traitements qu'il n'est nécessaire.

Résumé

Le mot-clé track permet à Angular d'identifier chaque élément d'une liste de manière unique. Il ne crée pas automatiquement un identifiant ; c'est le développeur qui indique quelle valeur doit être utilisée pour reconnaître chaque élément. Grâce à cette identification, Angular peut savoir précisément quels éléments ont été ajoutés, supprimés, déplacés ou modifiés, ce qui évite des mises à jour inutiles du DOM et améliore les performances de l'application. Dans les projets professionnels, il est recommandé d'utiliser un identifiant unique comme id plutôt que l'index de la liste.

Exercice 3

TS :

```
fruits = ["Pomme", "Banane", "Orange"];
```

afficher la liste

Correction

```
@for (f of fruits; track f) {
  <p>{{ f }}</p>
}
```

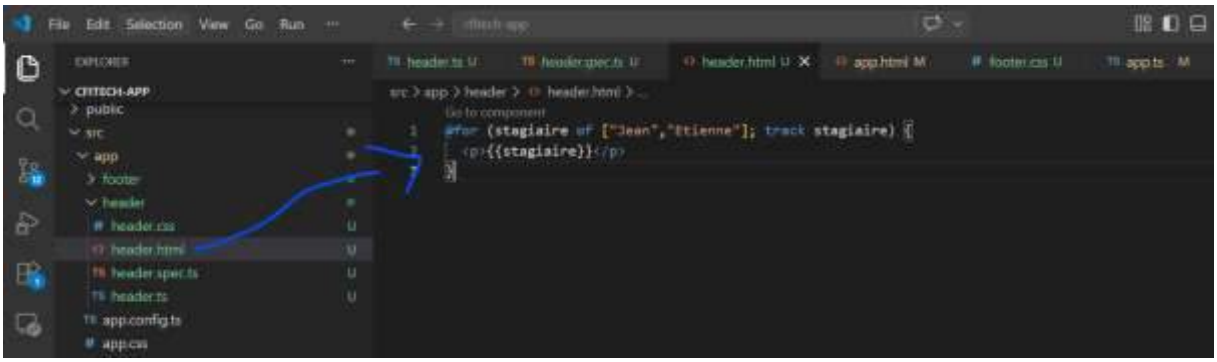
Exercice 4

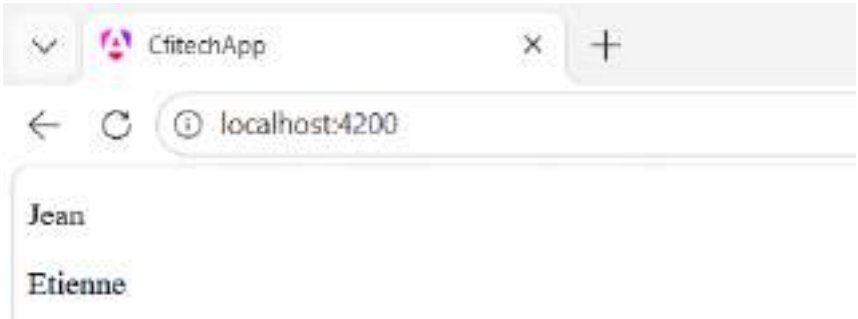
afficher :

Bonjour Ali
Bonjour Sara

Correction

```
@for (s of ["Ali", "Sara"]; track s) {
  <p>Bonjour {{ s }}</p>
}
```





4. @switch (Choix multiple)

Définition

@switch permet de gérer plusieurs cas selon une valeur

Syntaxe

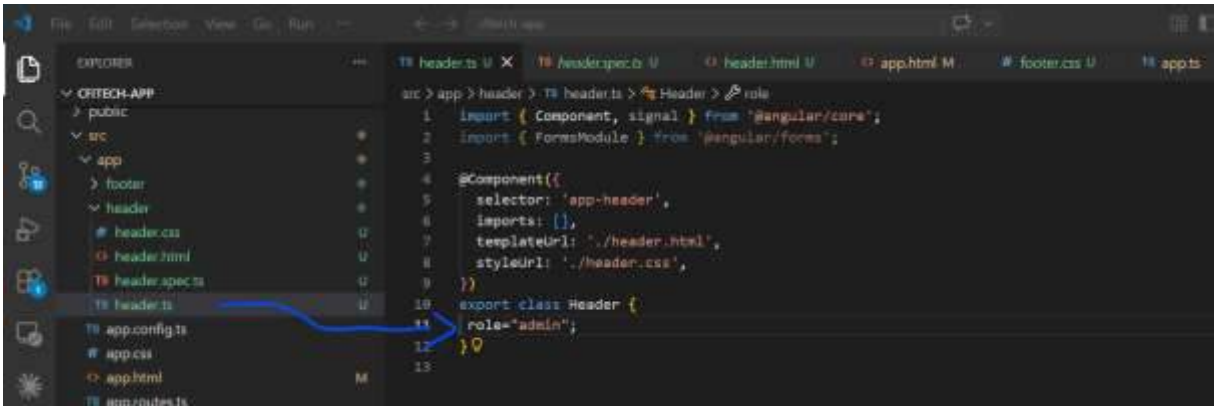
```
@switch (valeur) {
  @case ("A") { ... }
  @case ("B") { ... }
  @default { ... }
}
```

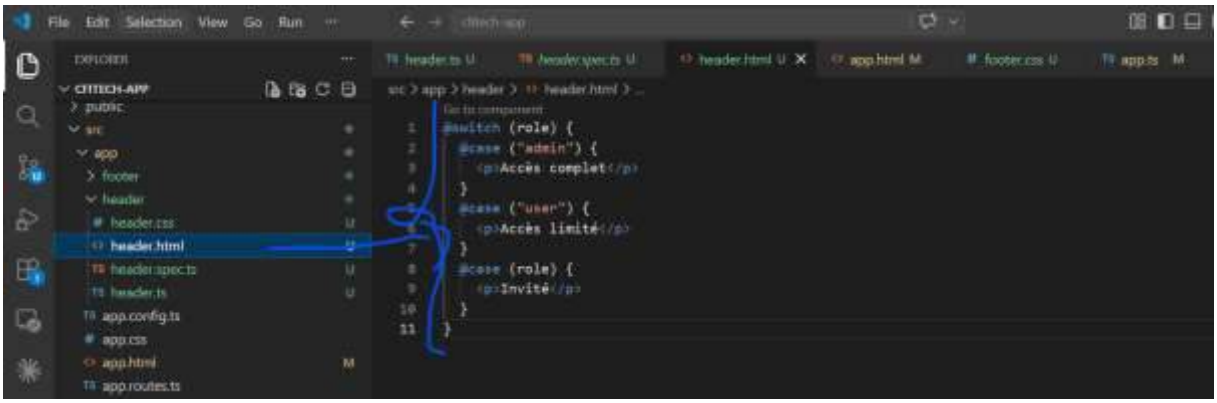
Exemple

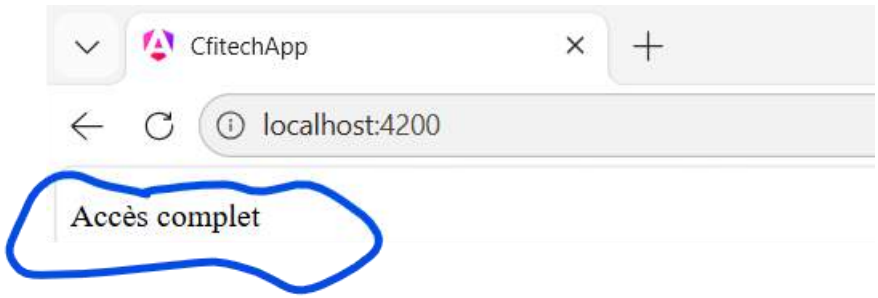
```
role = "admin";

@switch (role) {

  @case ("admin") {
    <p>Accès complet</p>
  }

  @case ("user") {
    <p>Accès limité</p>
  }

  @default {
    <p>Invité</p>
  }

}
```

Affichage

Accès complet







Exercice 5

TS :

niveau = "intermediaire";

afficher :

- debutant → “Niveau 1”
- intermediaire → “Niveau 2”
- autre → “Inconnu”

Correction

```
@switch (niveau) {

 @case ("debutant") {
   <p>Niveau 1</p>
 }

 @case ("intermediaire") {
   <p>Niveau 2</p>
 }

 @default {
   <p>Inconnu</p>
 }

}
```

Exercice 6 (COMPLET)

TS :

```
nom = "Ali";
notes = [12, 15, 9];
role = "user";
```

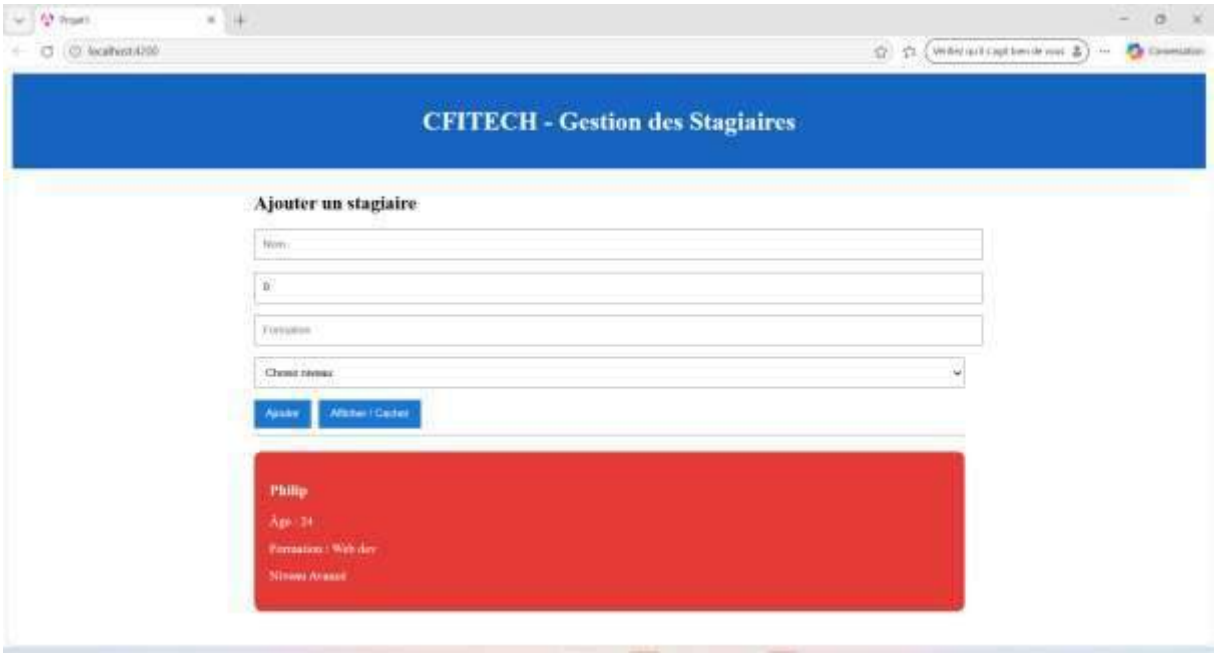
Objectif :

- afficher nom si existe
- afficher notes
- afficher rôle

Correction

```
@if (nom) {
  <p>{{ nom }}</p>
}

@for (n of notes; track n) {
  <p>Note : {{ n }}</p>
}

@switch (role) {

 @case ("admin") {
   <p>Admin</p>
 }

 @case ("user") {
```

```
  <p>User</p>
 }

}
```


CFITECH - Gestion des Stagiaires
Ajouter un stagiaire
Ajouter
Afficher / Cacher
Philip

1. **Directives d'attribut** (ex. `ngClass`, `ngStyle`)

- Les **directives d'attribut** sont des **instructions qu'on ajoute aux balises HTML** pour **modifier leur apparence ou leur comportement** (comme changer un style, cacher, répéter un élément, etc.).
- Elles **n'ajoutent pas de nouveaux éléments** dans le DOM, elles **modifient ceux qui existent déjà**.

a. Qu'est-ce que `ngClass` ?

- `ngClass` est une **directive d'attribut Angular** qui permet d'**ajouter ou de retirer dynamiquement des classes CSS** à un élément HTML.

- Elle est équivalente à un `class` HTML statique, mais beaucoup plus puissante car elle est **liée aux données du composant**.

`ngClass` sous toutes ses formes :

- Ajouter/retirer dynamiquement des classes CSS.
- Utiliser `ngClass` avec **string**, **array**, **objet**, **expressions conditionnelles**.
- Manipuler `ngClass` avec des **interactions utilisateur**.

-> Syntaxes possibles de `ngClass` :

1. **Avec une chaîne de caractères (`string`)**

<div [ngClass]="'ma-classe'">Exemple

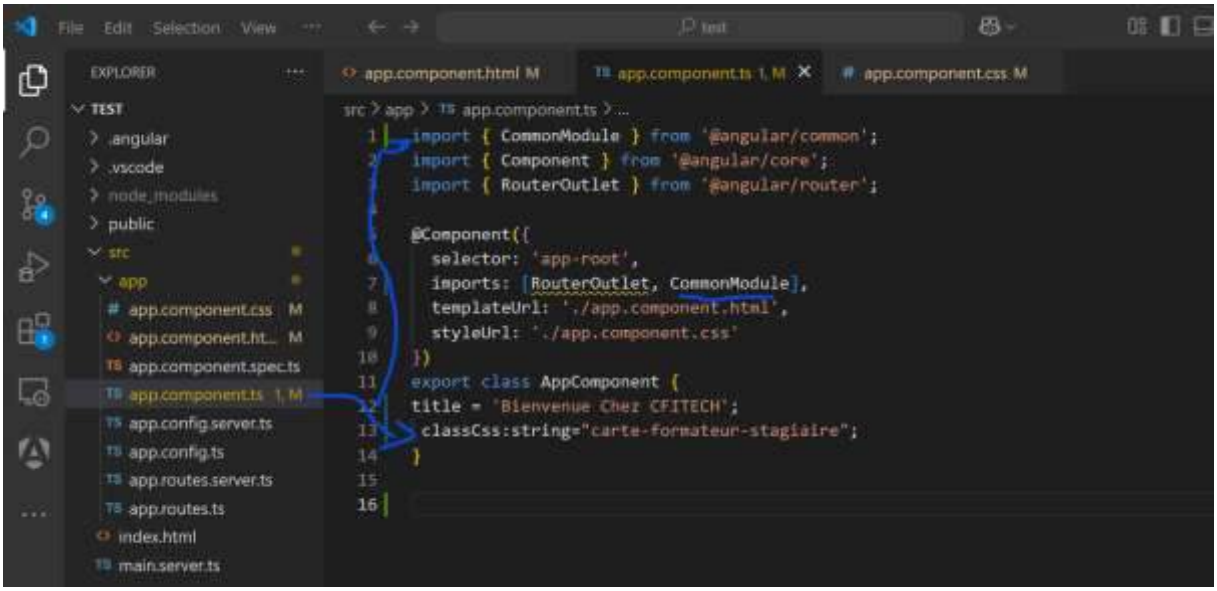
Ajoute la classe CSS `"ma-classe"`.

Ex :

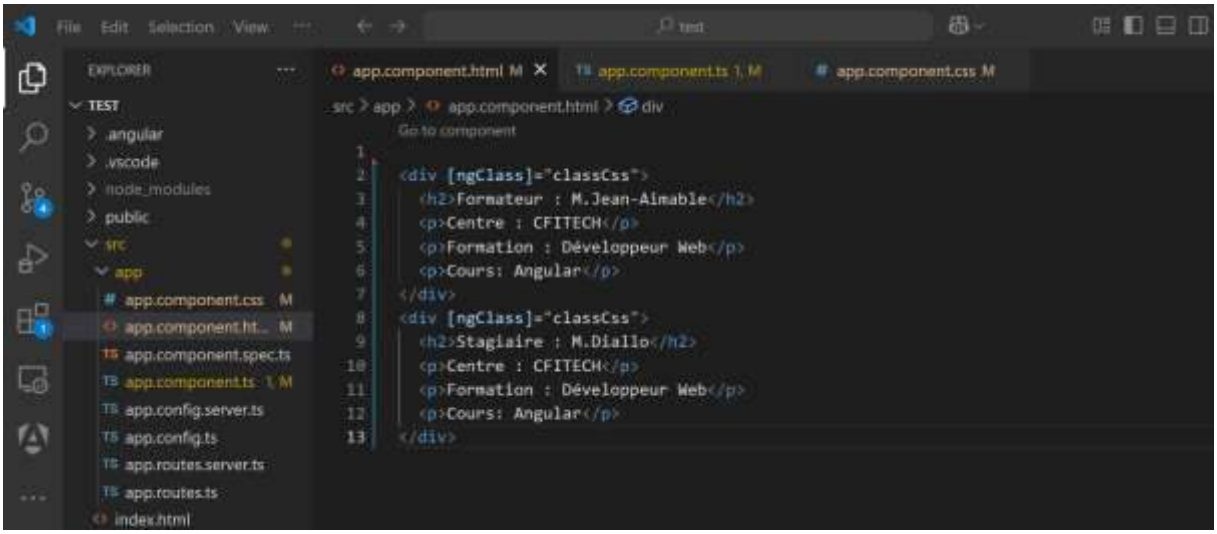
Affichage d'une **carte de présentation** d'un formateur ou d'un stagiaire dans un composant Angular. On souhaite appliquer une classe CSS pour styliser le bloc de présentation.

Rep :

ts



Html :

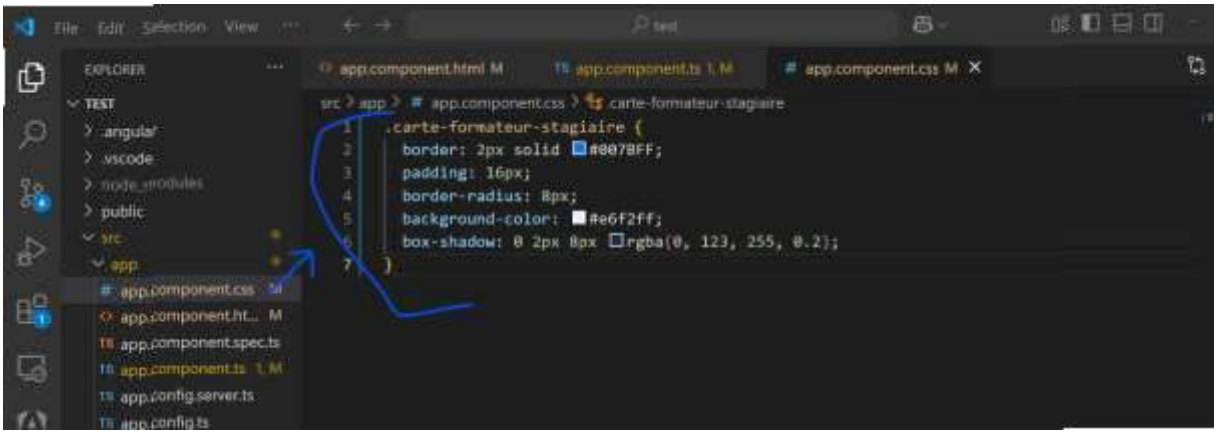



```
<div [ngClass]="classCss">
  <h2>Formateur : M.Jean-Aimable</h2>
  <p>Centre : CFITECH</p>
  <p>Formation : Développeur Web</p>
  <p>Cours: Angular</p>
</div>
<div [ngClass]="classCss">
  <h2>Stagiaire : M.Diallo</h2>
  <p>Centre : CFITECH</p>
  <p>Formation : Développeur Web</p>
  <p>Cours: Angular</p>
</div>
```

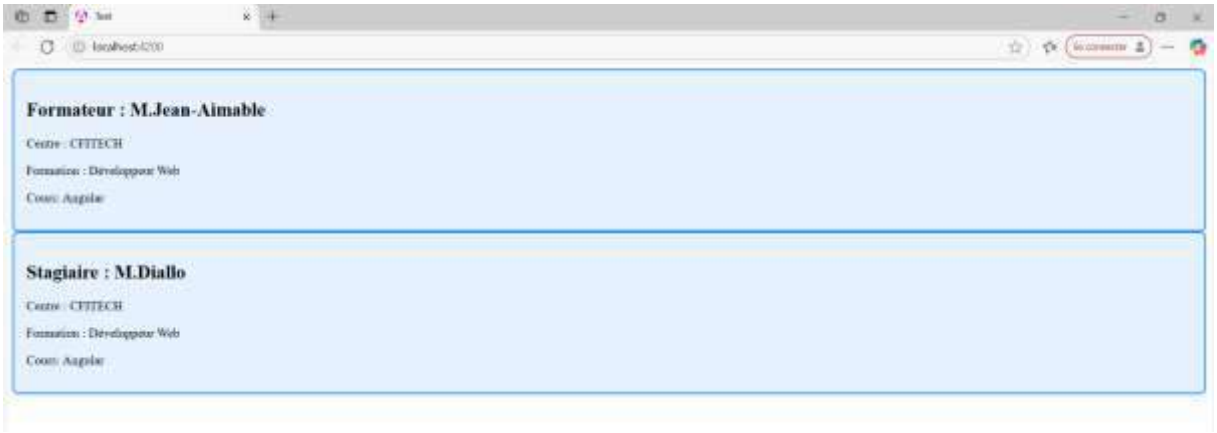
Sans ts :

```
div [ngClass]=" 'carte-formateur-stagiaire'">
  <h2>Formateur : M.Jean-Aimable</h2>
  <p>Centre : CFITECH</p>
  <p>Formation : Développeur Web</p>
  <p>Cours: Angular</p>
</div>
<div [ngClass]="classCss">
  <h2>Stagiaire : M.Diallo</h2>
  <p>Centre : CFITECH</p>
  <p>Formation : Développeur Web</p>
  <p>Cours: Angular</p>
</div>
```

Css :




```
.carte-formateur-stagiaire {
  border: 2px solid #007BFF;
  padding: 16px;
  border-radius: 8px;
  background-color: #e6f2ff;
  box-shadow: 0 2px 8px rgba(0, 123, 255, 0.2);
}
```



2. Avec un tableau (`array`)

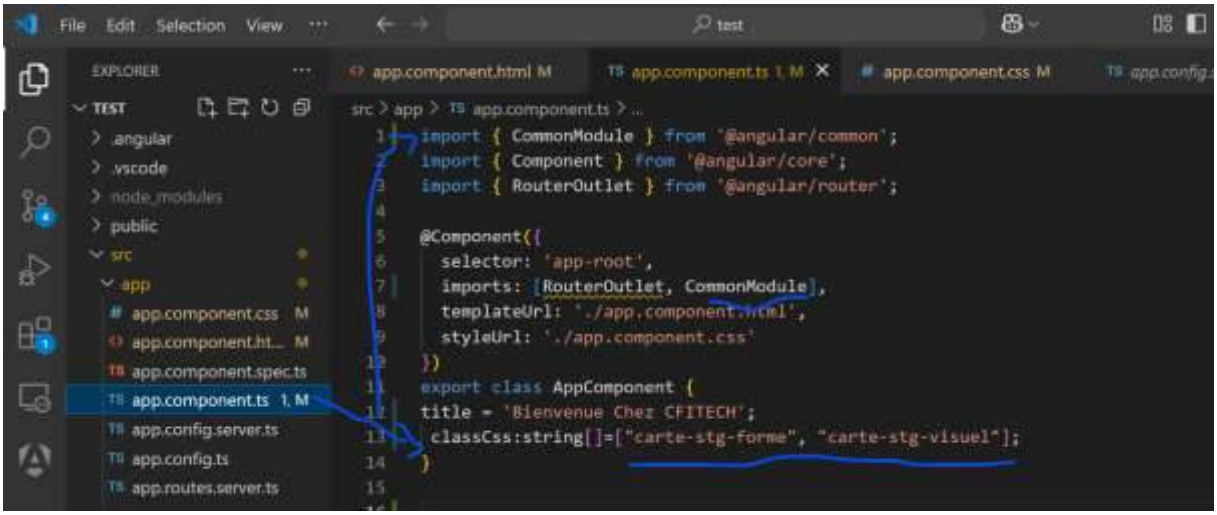
**<div [ngClass]="['classe1', 'classe2']">Texte **

Ex :

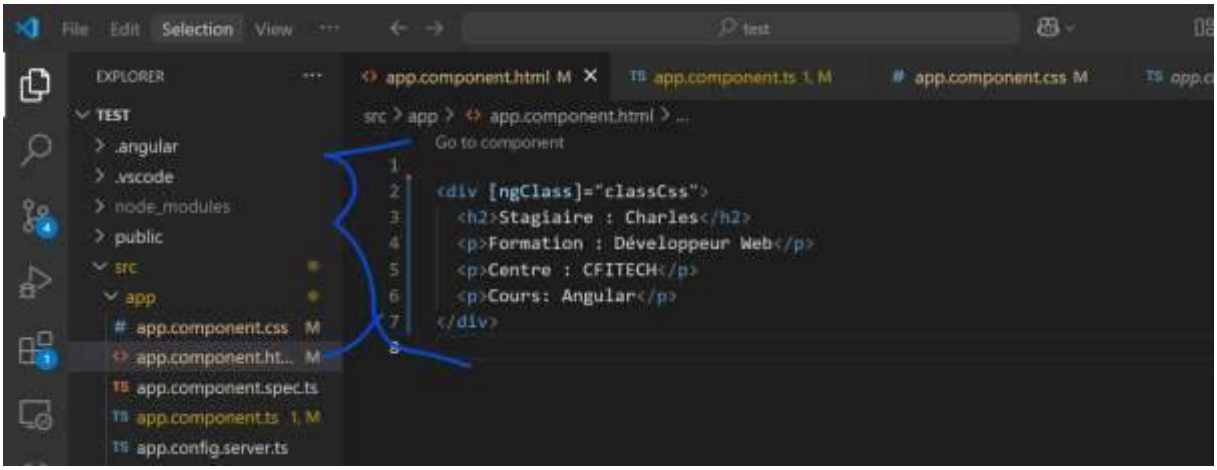
Afficher un **stagiaire** dans une carte stylisée avec **plusieurs classes CSS**, comme :

- Une classe pour la couleur du fond,
- Une autre pour les bordures ou les marges.

ts : dynamique avec un tableau dans le `.ts`



Html :

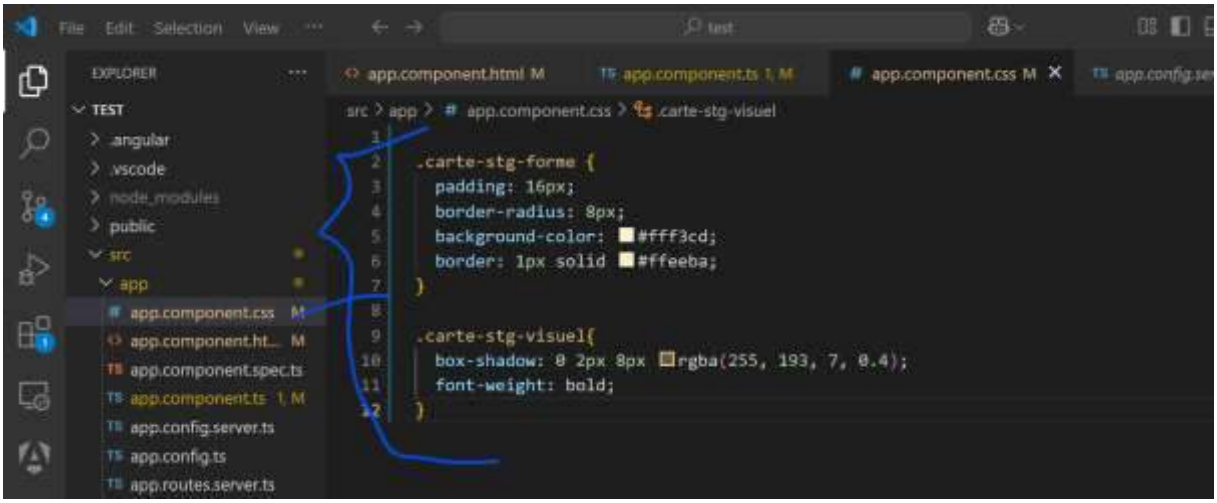



```
<div [ngClass]="classCss">
  <h2>Stagiaire : Charles</h2>
  <p>Formation : Développeur Web</p>
  <p>Centre : CFITECH</p>
  <p>Cours: Angular</p>
</div>
```

Sans ts :

```
<div [ngClass]=" ['carte-stg-forme','carte-stg-visuel']">

  <h2>Stagiaire : Charles</h2>
  <p>Formation : Développeur Web</p>
  <p>Centre : CFITECH</p>
  <p>Cours: Angular</p>
</div>
```

Css :




```css
.carte-stg-forme {
  padding: 16px;
  border-radius: 8px;
  background-color: #fff3cd;
  border: 1px solid #ffeeba;
}

.carte-stg-visuel{
  box-shadow: 0 2px 8px rgba(255, 193, 7, 0.4);
  font-weight: bold;
}
```

3. Avec un objet (`object`)

<div [ngClass]="{ 'active': isActive, 'disabled': isDisabled }">

Exemple

** **

Ajoute `"active"` si `isActive === true`, et `"disabled"` si `isDisabled === true`.

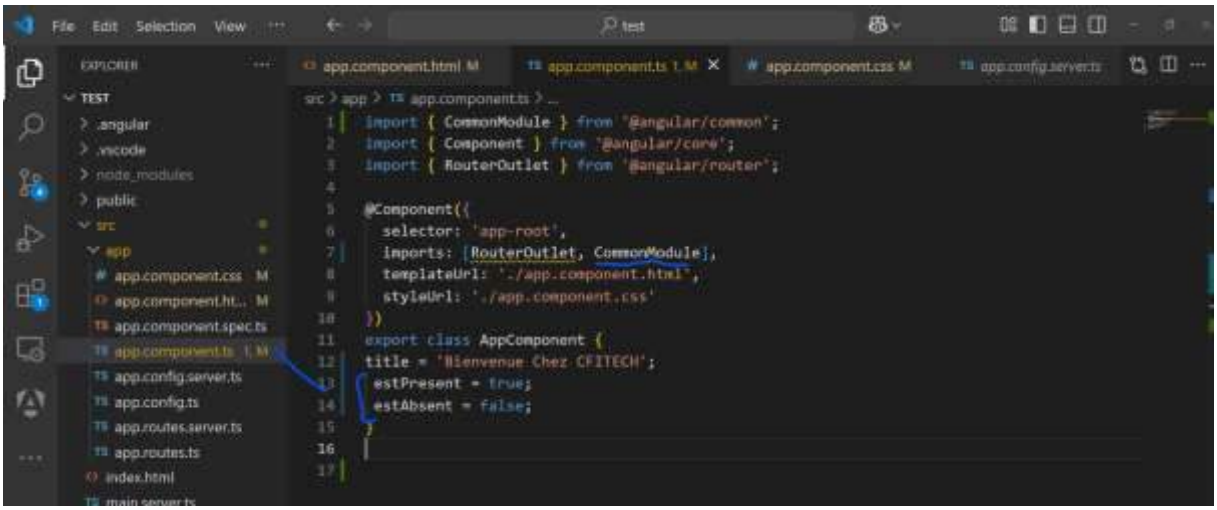
Ex :

Afficher une **fiche de stagiaire** dans une liste.

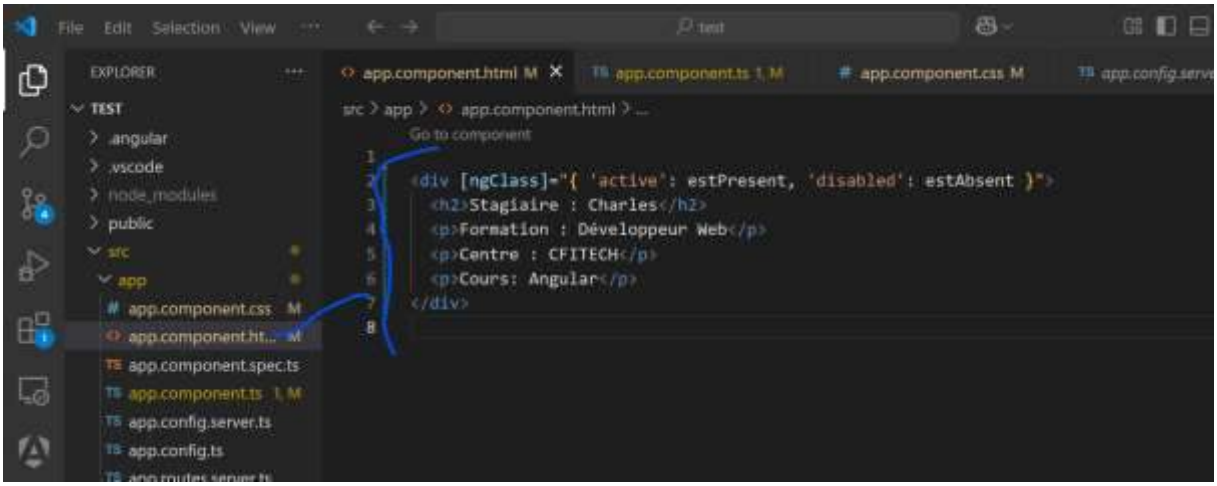
- Lui appliquer la classe `"active"` s'il est **présent aujourd'hui**,
- Lui appliquer `"disabled"` s'il est **absent ou inactif**.

Rep :

ts :

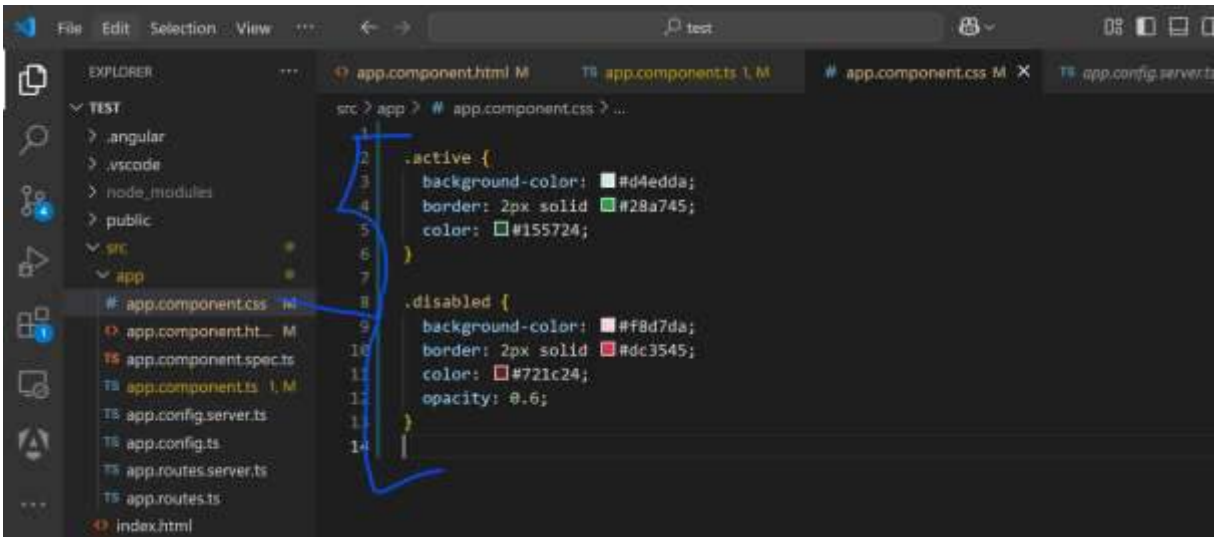


Html :




```
<div [ngClass]="{ 'active': estPresent, 'disabled': estAbsent }">
  <h2>Stagiaire : Charles</h2>
  <p>Formation : Développeur Web</p>
  <p>Centre : CFITECH</p>
  <p>Cours: Angular</p>
</div>
```

Css :

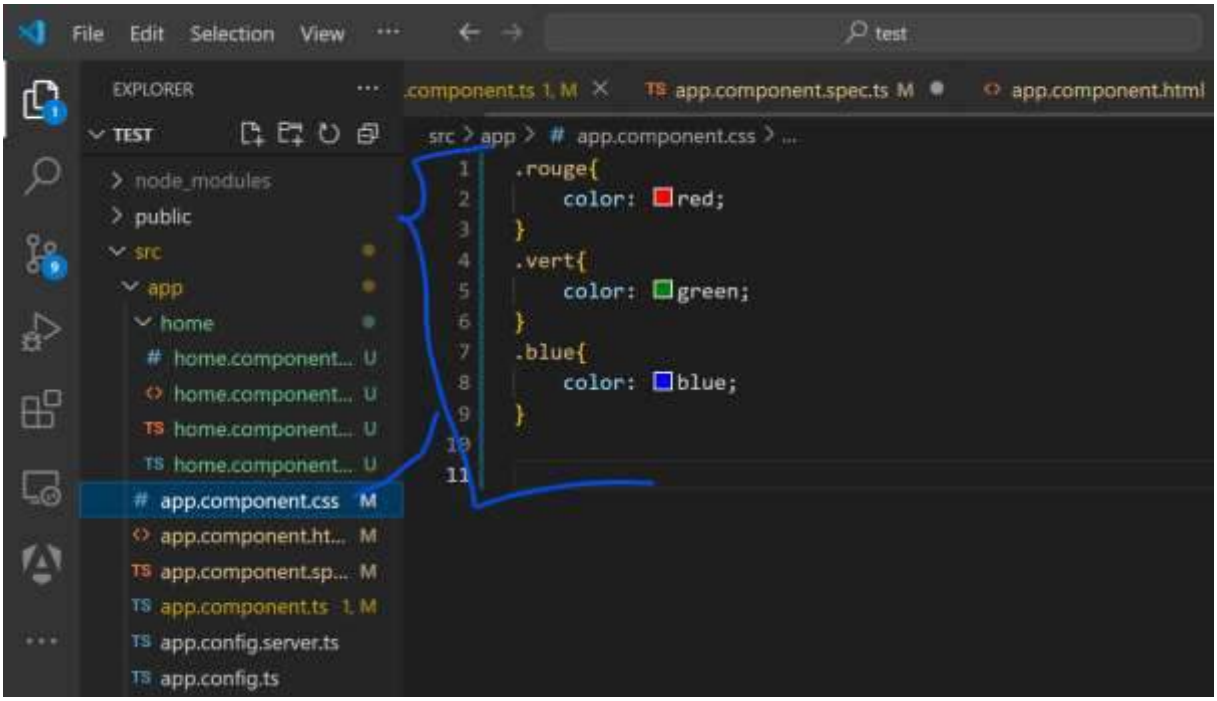


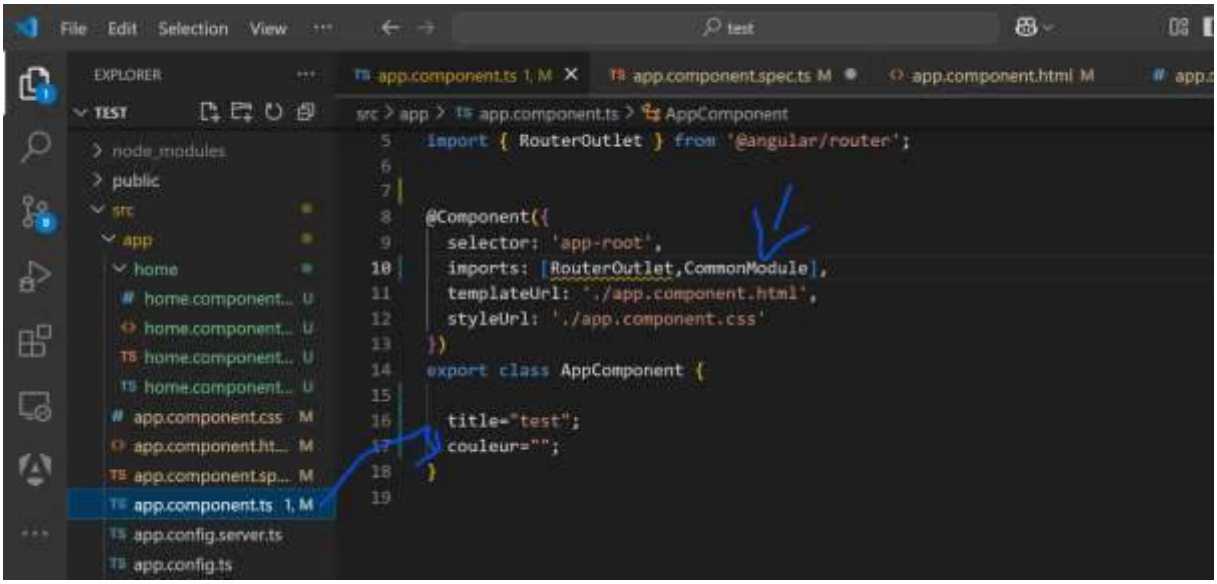

```
.active {
  background-color: #d4edda;
  border: 2px solid #28a745;
  color: #155724;
}

.disabled {
  background-color: #f8d7da;
```

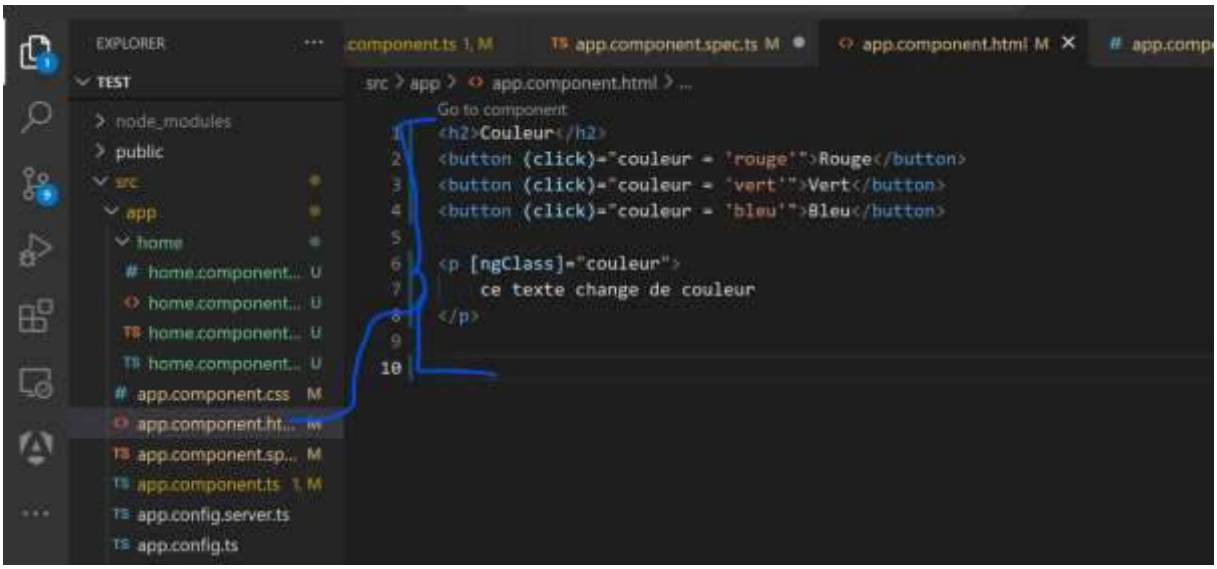
```
  border: 2px solid #dc3545;
  color: #721c24;
  opacity: 0.6;
}
```

Ex particulier :

Changer la **couleur du texte** d'un paragraphe quand on clique sur des boutons.







Donc si `couleur = 'bleu'`, Angular applique `.bleu` au paragraphe, et il devient bleu.

La variable `couleur` reçoit la valeur `'bleu'`

Le paragraphe applique automatiquement la classe CSS `.bleu`

⇨ Il existe déjà des **classes CSS "prêtes à l'emploi"**, appelées **classes utilitaires** ou **pré-définies**, que vous n'avez **pas besoin de créer vous-mêmes**.

Elles viennent généralement de **frameworks CSS** populaires, comme :

Bootstrap

Le plus connu. Il propose des classes toutes faites comme :

Classe Bootstrap	Rôle
`btn`	Style de base pour un bouton
`btn-primary`	Bouton bleu principal
`btn-danger`	Bouton rouge (souvent pour "supprimer")
`text-center`	Centrer le texte
`bg-success`	Fond vert
`d-flex`	Active Flexbox
`mt-3`, `mb-2`	Marges top/bottom (`mt` = margin-top)
`p-3`, `px-4`	Padding (`p` = padding, `x` = horizontal)

On peut les utiliser **directement avec `ngClass`** :

!!!Ces classes **ne fonctionnent que si tu as bien installé le framework CSS** (Bootstrap, Tailwind, etc.) dans ton projet Angular.

Avant de les utiliser, il faut d'abord ajouter bootstrap dans votre projet angular :

npm install bootstrap

Vérifie que Bootstrap est bien installé : npm list bootstrap

Si ça affiche une erreur ou rien → Bootstrap n'est **pas installé**.

- Et dans `angular.json` :

Ajouter dans la section principale `build.options.styles`,

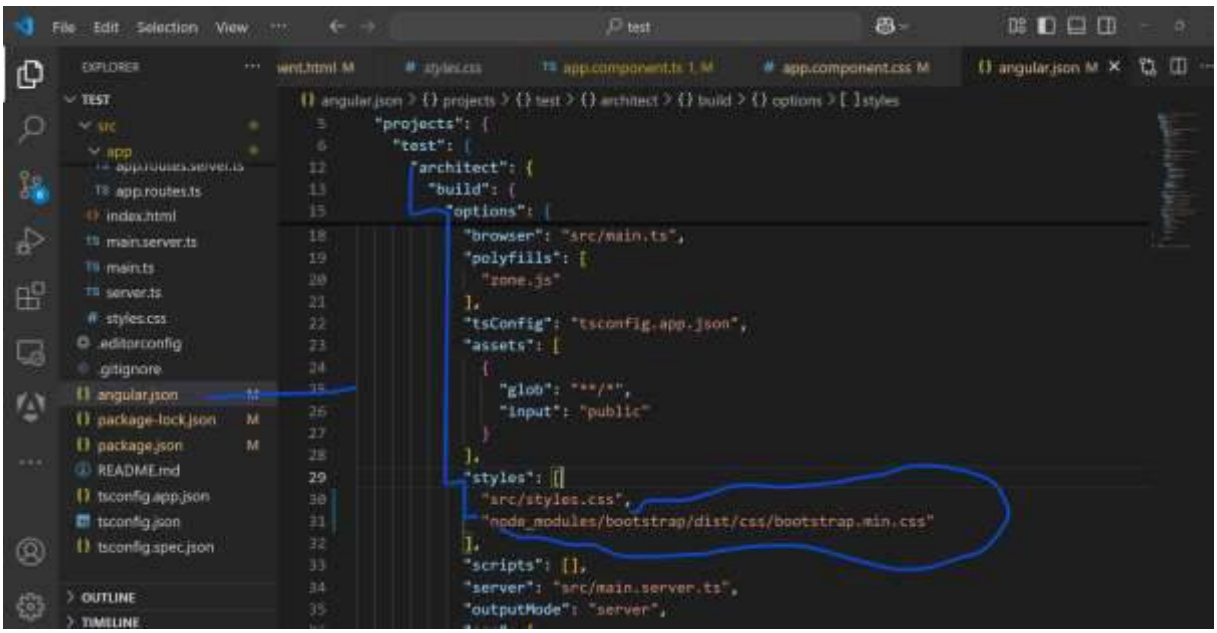
pas dans section : architect.test.options.styles

"styles": [

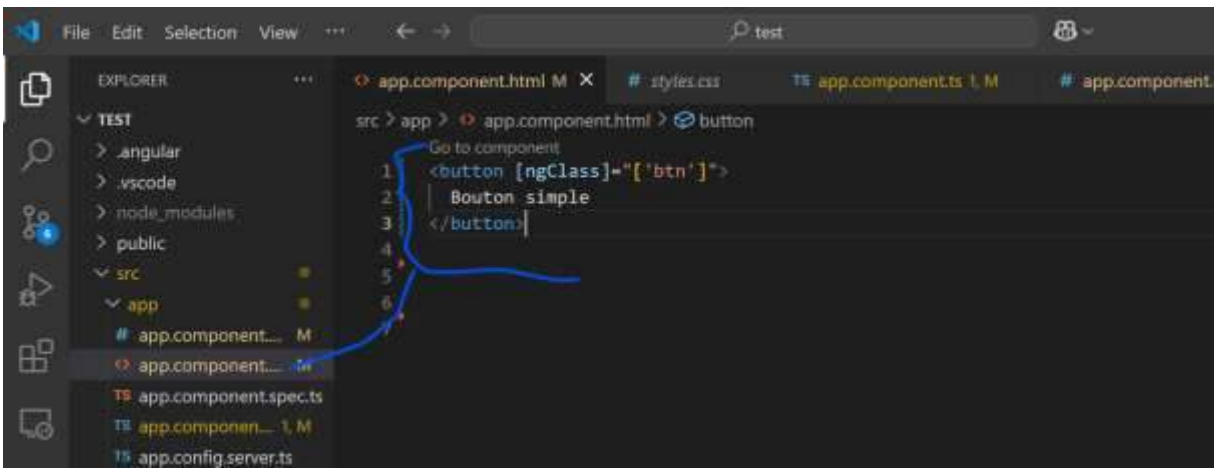
"node_modules/bootstrap/dist/css/bootstrap.min.css",

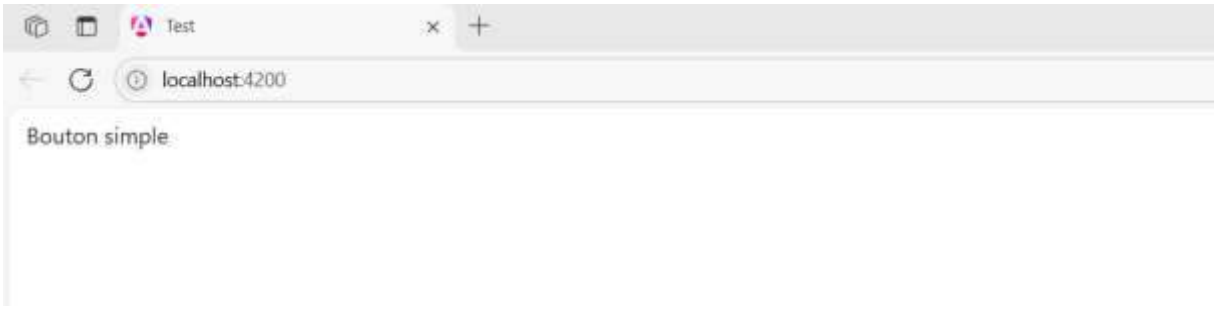
"src/styles.css"

]

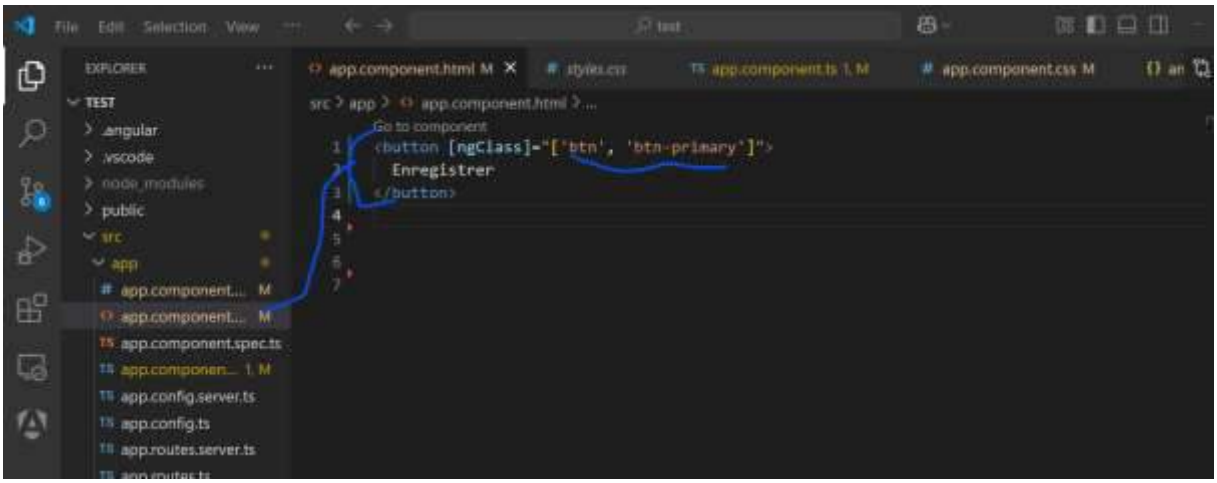


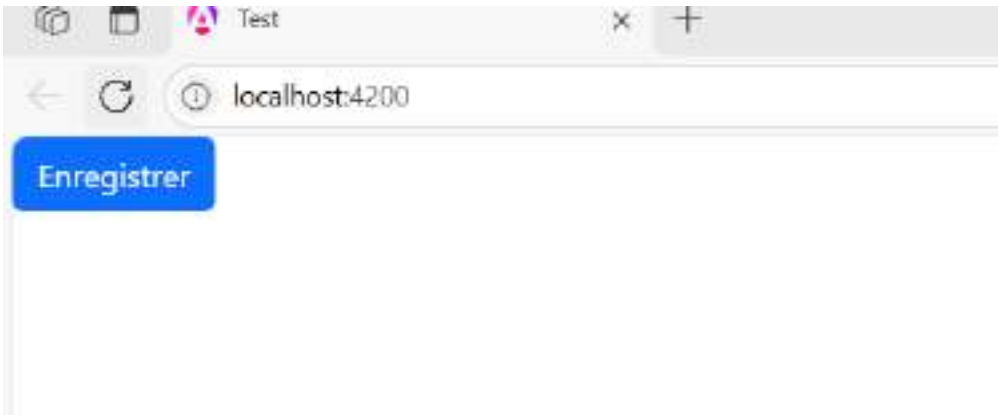
Ex1. `btn` – Style de base pour bouton



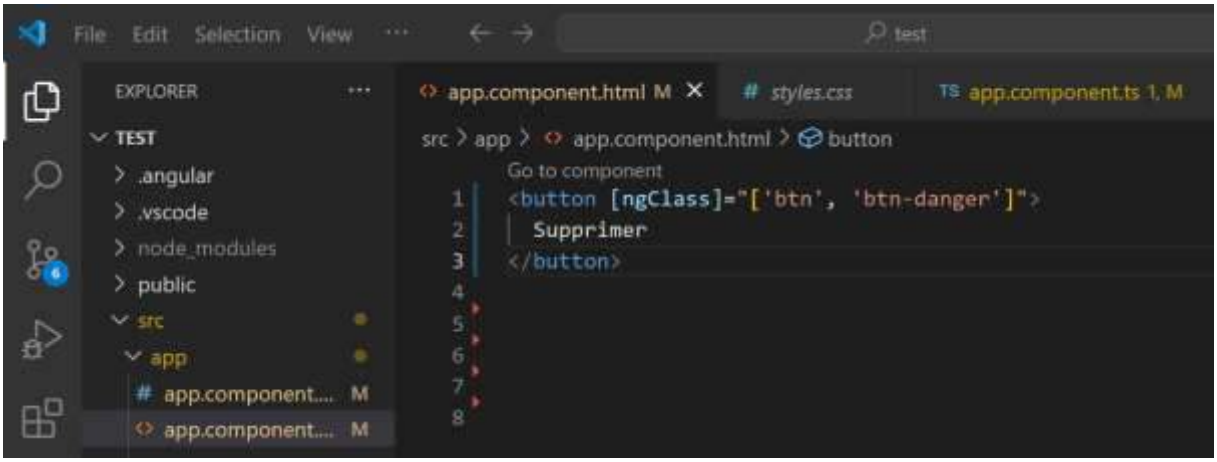


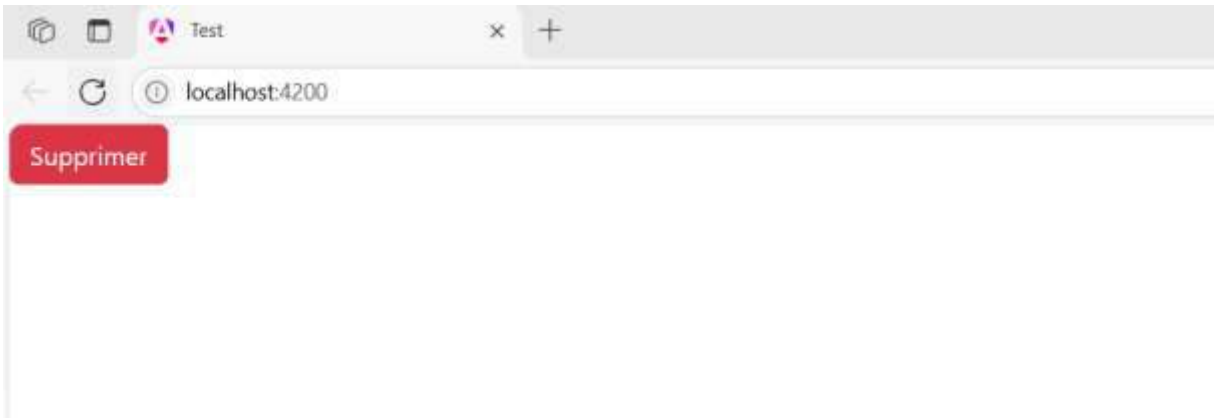
Ex2. `btn-primary` – Bouton bleu principal



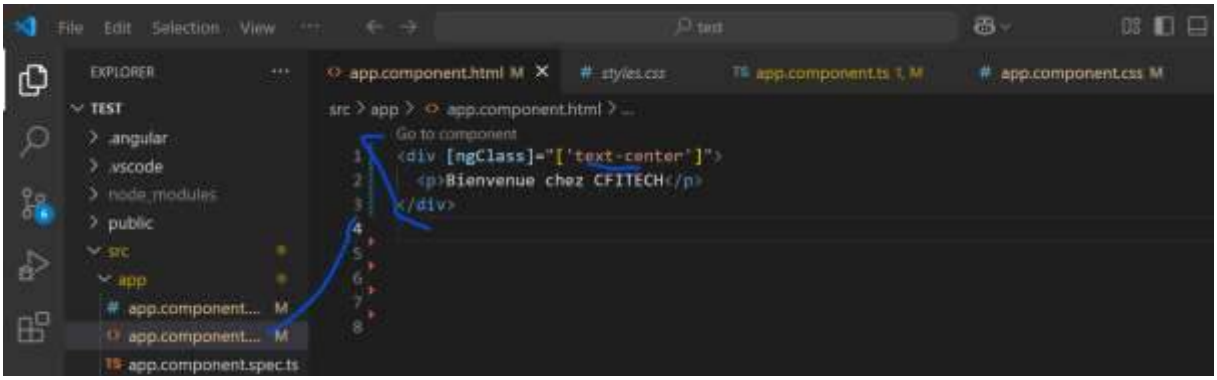


Ex3. `btn-danger` – Bouton rouge

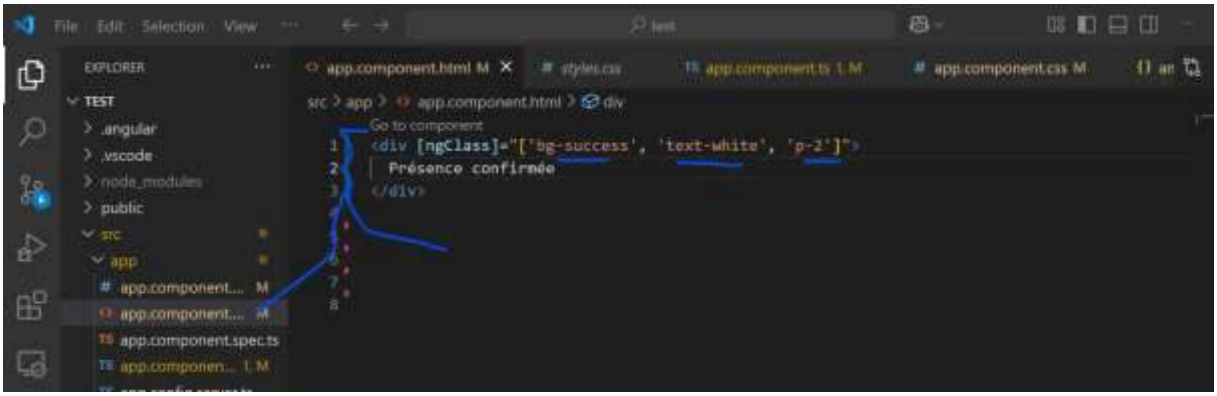


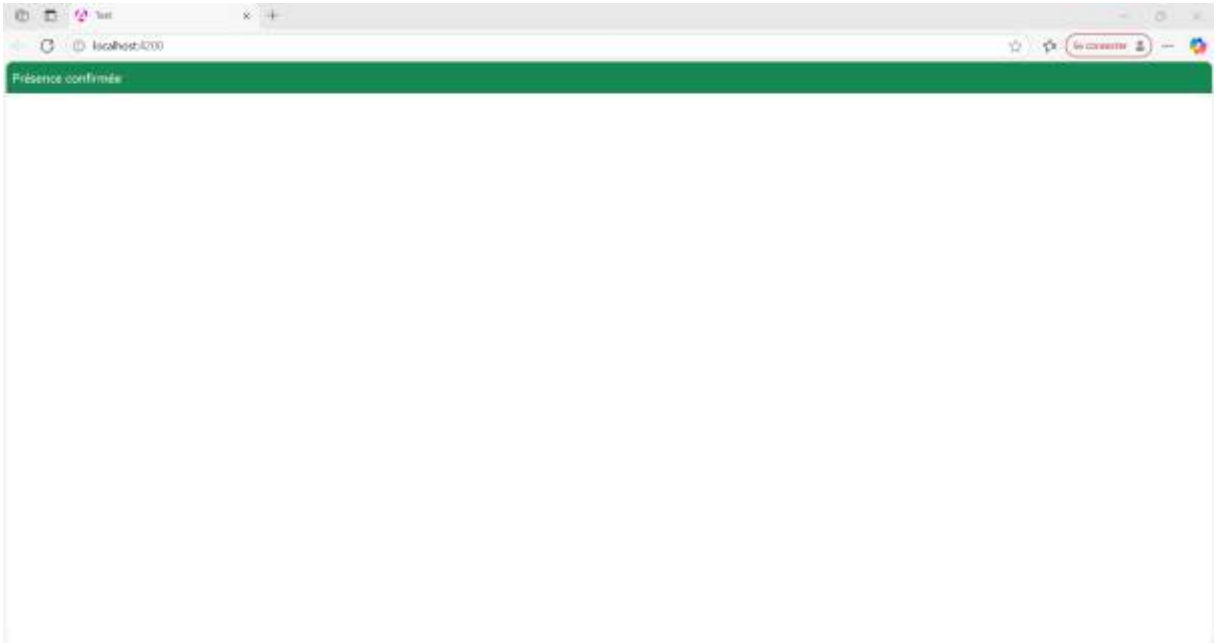


Ex4. `text-center` – Centrer le texte

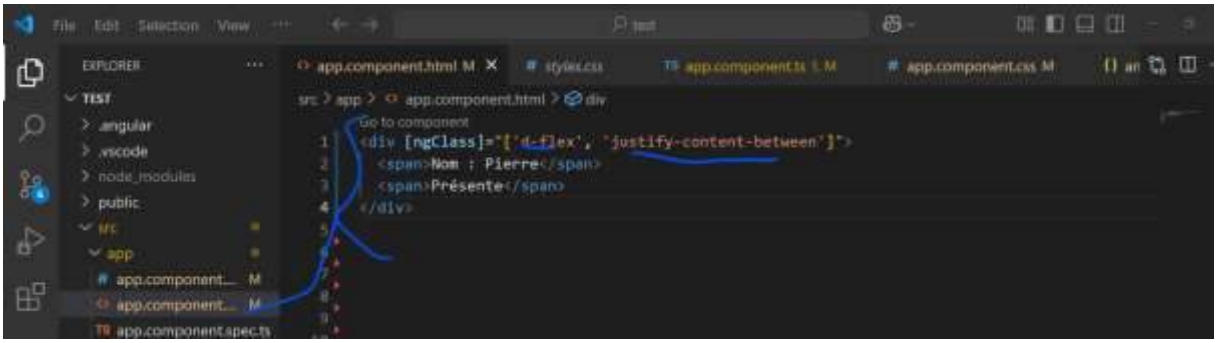


Ex5. `bg-success` – Fond vert (succès)





Ex6. `d-flex` – Activer Flexbox



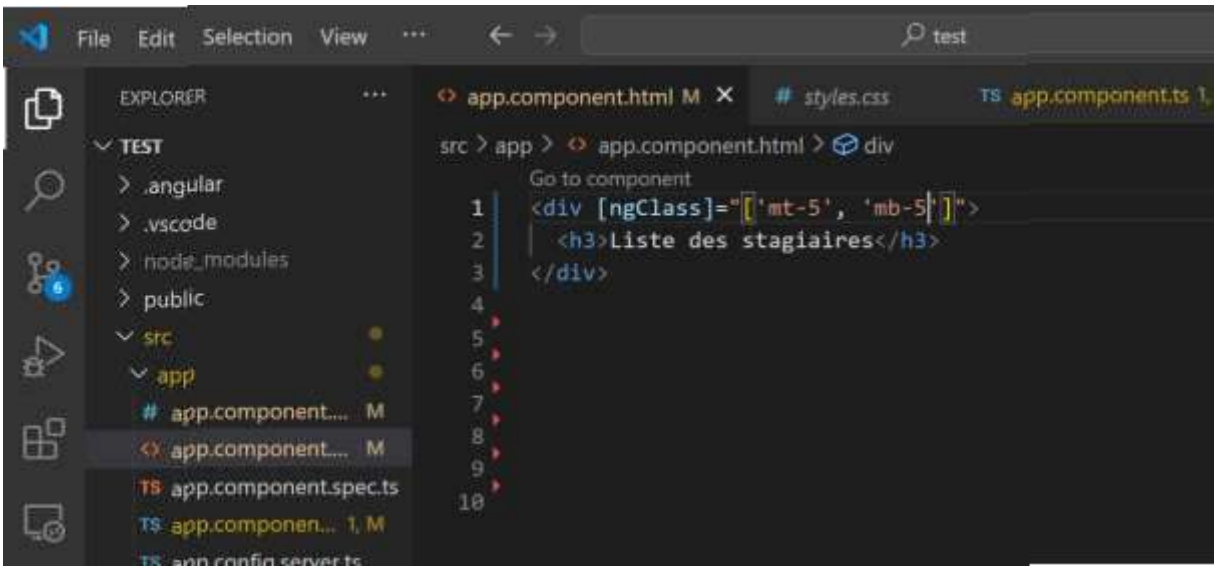
- **`d-flex`** :

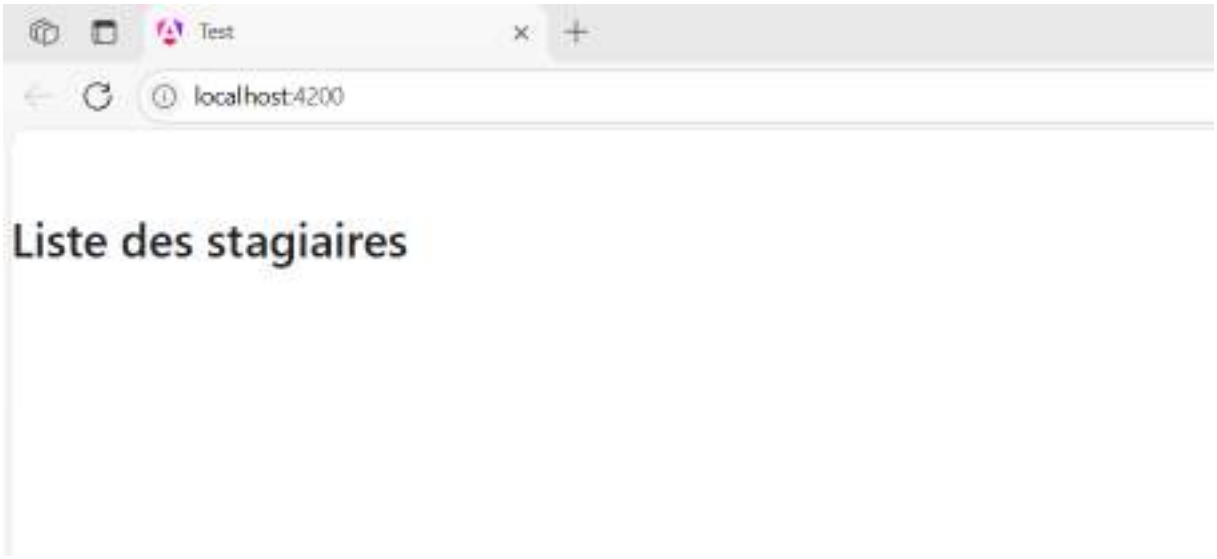
  - C'est une classe Bootstrap qui active **Flexbox** sur le `div`.
  - Cela transforme le conteneur en un **conteneur flex**, ce qui permet d'aligner les éléments enfants horizontalement ou verticalement de manière plus flexible.

- **`justify-content-between`** :

  - Une autre classe Flexbox de Bootstrap.
  - Elle positionne les **éléments enfants** (ici, les deux `<span>`) de manière à ce qu'il y ait **le plus grand espace possible entre eux** :
    - Le **premier élément** est aligné à gauche.
    - Le **deuxième élément** est aligné à droite.

Ex7. `mt-5`, `mb-5` – Marges (top et bottom)





Ex8. `p-3`, `px-4` – Padding global et horizontal

Classe 1 : `p-3`

- **`p`** = *padding* (espace **intérieur** autour du contenu)
- **`3`** = niveau d'espacement (de `0` à `5`)

Applique un **padding sur tous les côtés** (haut, bas, gauche, droite)

Classe 2 : `px-4`

- **`p`** = *padding*
- **`x`** = directions **horizontales** (gauche et droite uniquement)
- **`4`** = un peu plus d'espacement qu'avec `3`

Ajoute un **padding plus large** sur la gauche et la droite que celui de `p-3`

Classe 3 : `bg-light`

- **`bg`** = *background*
- **`light`** = couleur claire (généralement un gris très pâle)

Applique un **fond clair** à ton `div`

b. C’est quoi `ngStyle`

`ngStyle` permet d’ajouter ou de modifier des styles CSS **directement dans le HTML** via des expressions Angular.

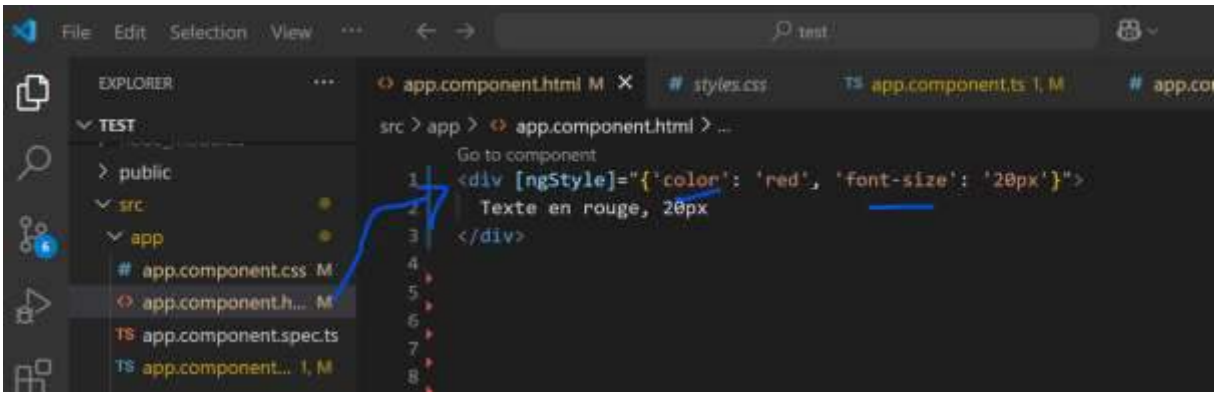
C’est l’équivalent du `[style]`, mais en version plus flexible

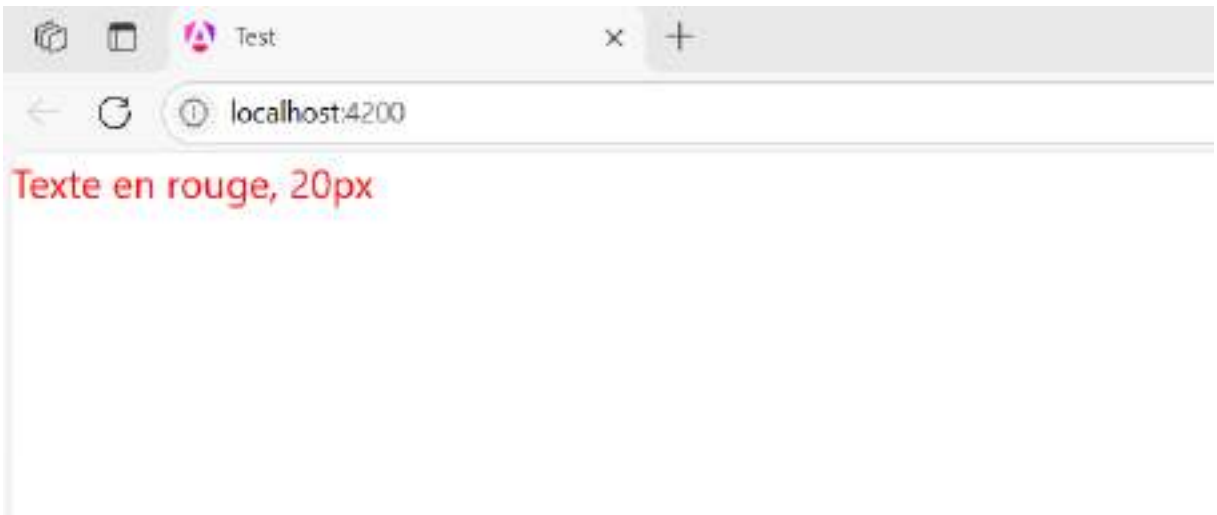
Syntaxes principales de ngStyle

Il existe plusieurs formes d’utilisation :

1. Objet JavaScript

Ex :

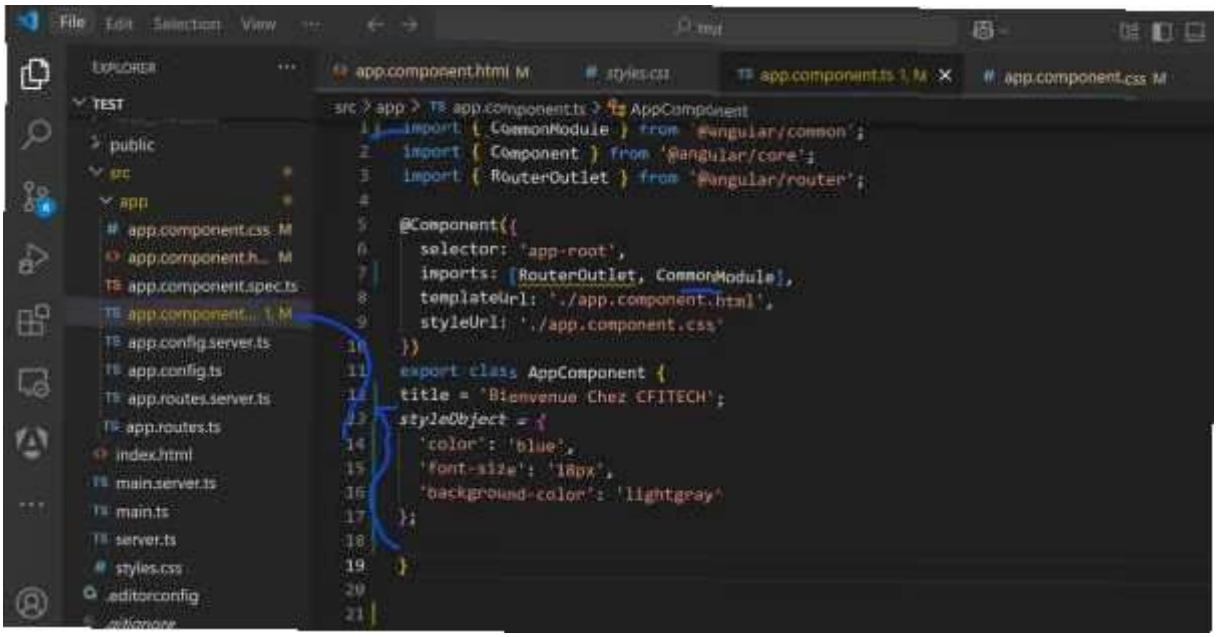




2. Expression liée à une variable (dans le component)

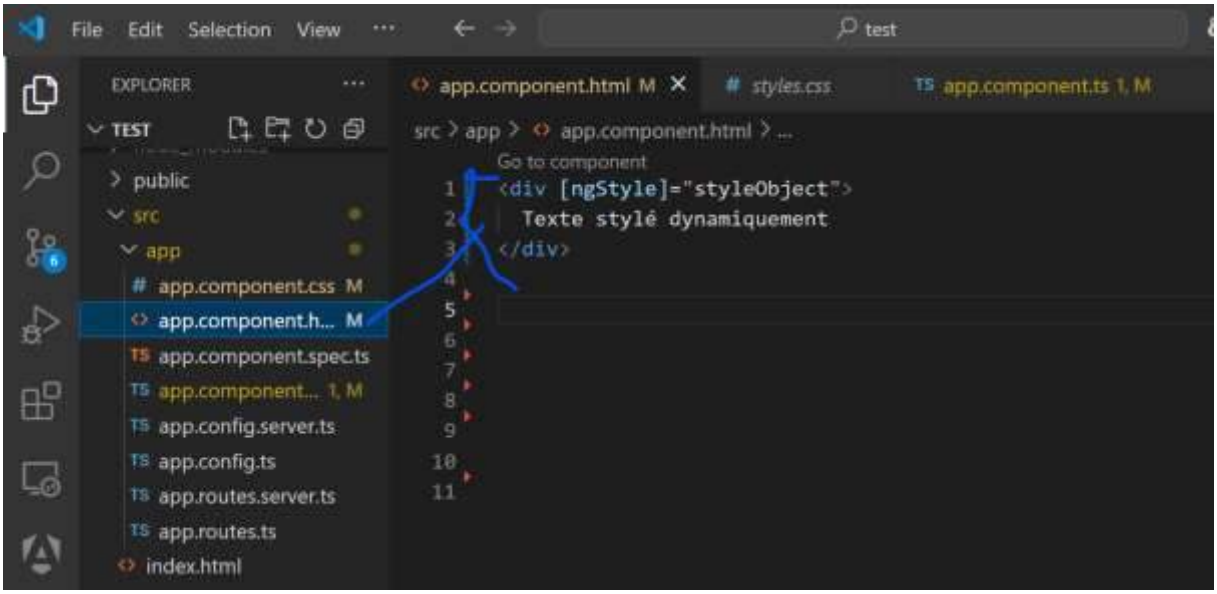
Ex :

Ts :

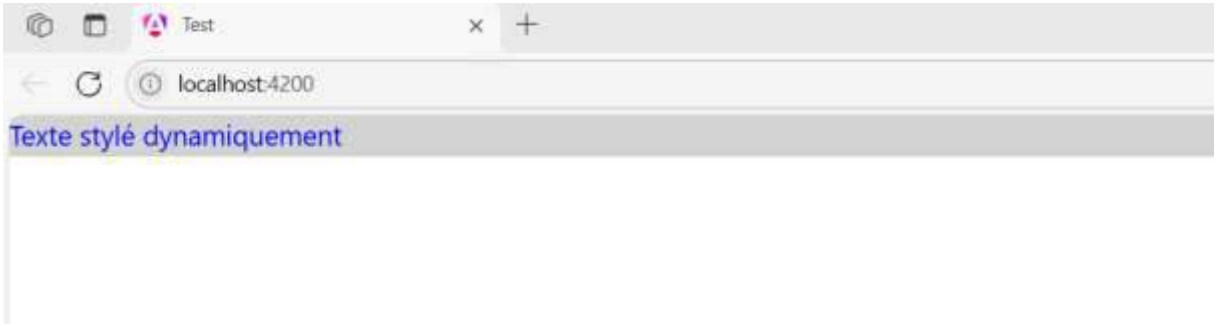



```ts
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet, CommonModule],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
title = 'Bienvenue Chez CFITECH';
styleObject = {
  'color': 'blue',
  'font-size': '18px',
  'background-color': 'lightgray'
};

}
```

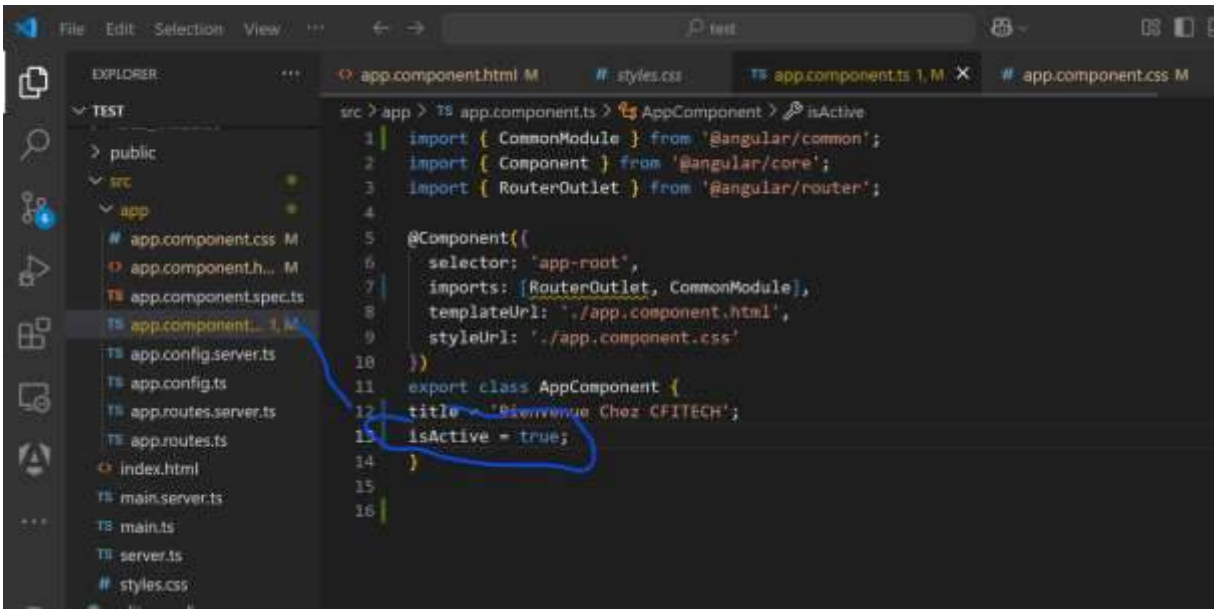
Html :


File Edit Selection View
test
EXPLORER
TEST
app.component.html M
styles.css
app.component.ts 1, M
src > app > app.component.html > ...
Go to component
<div [ngStyle]="styleObject">
Texte stylé dynamiquement

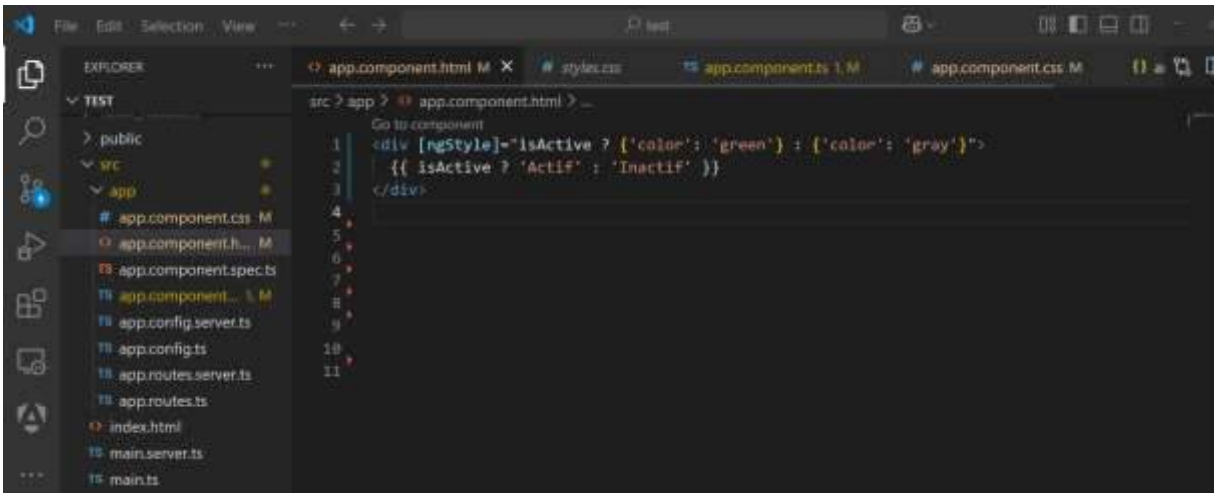
public
src
app
app.component.css M
app.component.h... M
app.component.spec.ts
app.component... 1, M
app.config.server.ts
app.config.ts
app.routes.server.ts
app.routes.ts
index.html


Test
localhost:4200
Texte stylé dynamiquement

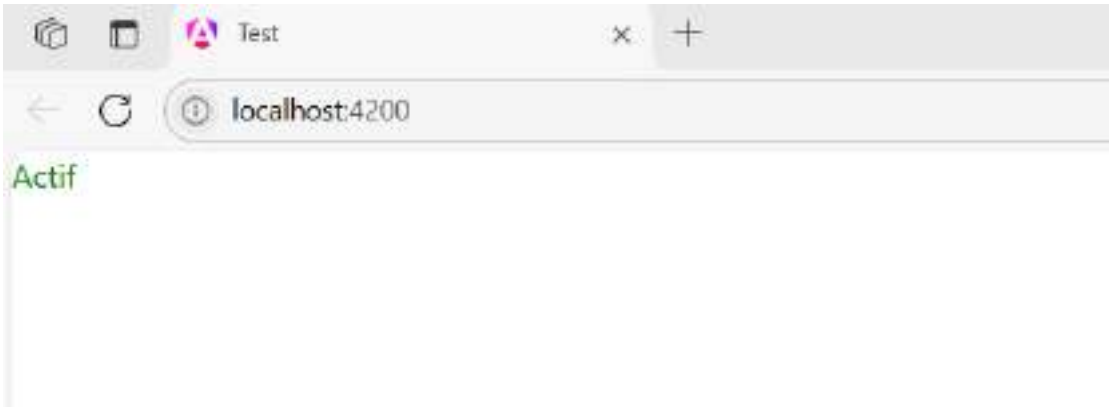
3. **Conditions dynamiques (ternaire ou booléen)**



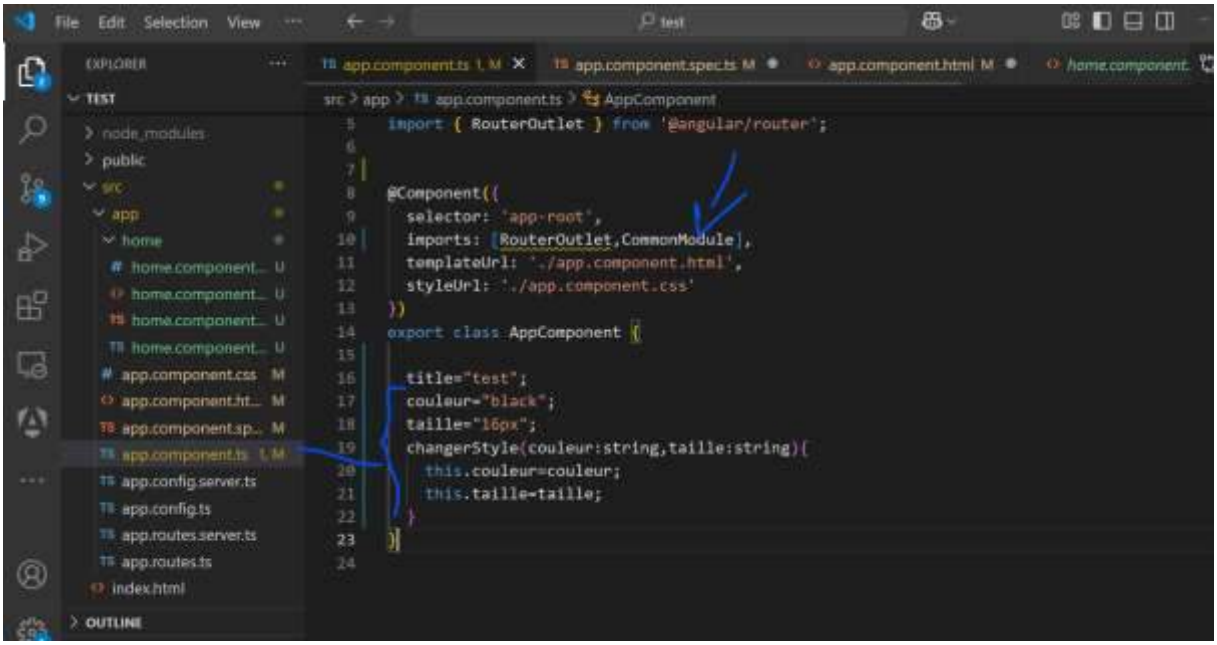
Html :

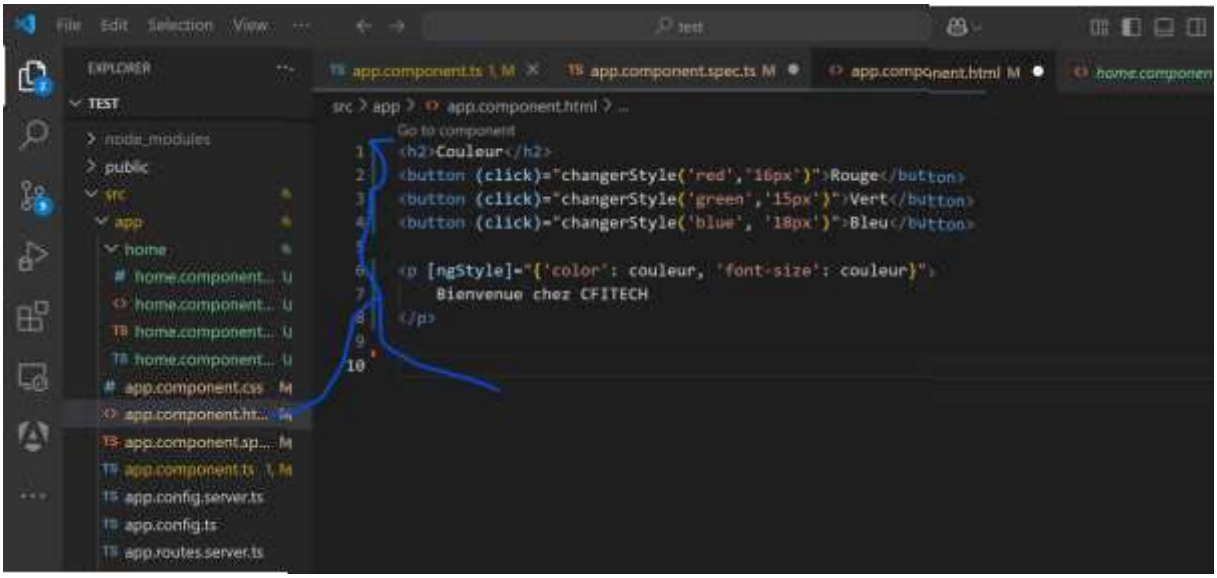



```
<div [ngStyle]="isActive ? {'color': 'green'} : {'color': 'gray'}">
  {{ isActive ? 'Actif' : 'Inactif' }}
</div>
```



Ex : Changer la taille et la couleur d'un texte selon un bouton





`[ngStyle]="{ 'color': couleur }"` Change la **couleur** du texte avec la variable

`'font-size': taille` Change la **taille** du texte avec la variable

`(click)="changerStyle(...)"`

`Exercices`

Exercice 1 – Affichage conditionnel avec @if

Le centre de formation CFITECH ouvre régulièrement les inscriptions pour ses nouvelles sessions de formation. Le responsable pédagogique souhaite afficher un message d'information aux visiteurs uniquement lorsque les inscriptions sont ouvertes.
Créez une variable permettant d'indiquer si les inscriptions sont ouvertes ou non. Lorsque les inscriptions sont ouvertes, le message suivant doit apparaître :
Les inscriptions sont ouvertes.
Si les inscriptions sont fermées, aucun message ne doit être affiché.

Réponse

TypeScript

```
inscriptionsOuvertes = true;
```

HTML

```
@if (inscriptionsOuvertes) {
  <p>Les inscriptions sont ouvertes.</p>
}
```

Exercice 2 – Affichage conditionnel avec @if et @else

Une formation Angular chez CFITECH possède un nombre limité de places. Lorsque des places sont encore disponibles, les visiteurs doivent voir le message :
Formation disponible
Dans le cas contraire, ils doivent voir :
Formation complète
Réalisez cette fonctionnalité à l'aide du nouveau système de contrôle de flux Angular.

Réponse

TypeScript

```
placesDisponibles = false;
```

HTML

```
@if (placesDisponibles) {
  <p>Formation disponible</p>
} @else {
  <p>Formation complète</p>
}
```

Exercice 3 – Afficher une liste de formations avec @for

CFITECH propose plusieurs formations en développement web. Le responsable du site souhaite afficher automatiquement la liste des formations disponibles à partir d'un tableau TypeScript.
Les formations proposées sont :

- Angular
- React
- Java Spring

Affichez cette liste à l'aide de la directive @for.

Réponse

TypeScript

```
formations = [
  'Angular',
  'React',
  'Java Spring'
];
```

HTML

```
@for (formation of formations; track formation) {
  <p>{{ formation }}</p>
}
```

Exercice 4 – Afficher les stagiaires avec track

Le secrétariat de CFITECH souhaite afficher la liste des stagiaires inscrits à une formation Angular. Chaque stagiaire possède un identifiant unique ainsi qu'un nom. Afin d'optimiser les performances de l'application, utilisez la bonne pratique Angular en spécifiant une clé unique avec track.

Réponse

TypeScript

```
stagiaires = [
 { id: 1, nom: 'Ali' },
 { id: 2, nom: 'Sara' },
 { id: 3, nom: 'Omar' }
];
```

HTML

```
@for (s of stagiaires; track s.id) {
 <p>{{ s.nom }}</p>
}
```

Exercice 5 – Gestion des niveaux avec @switch

Chaque stagiaire inscrit chez CFITECH possède un niveau :

- débutant
- intermédiaire
- avancé

Selon le niveau enregistré, un message spécifique doit être affiché à l'écran.
Utilisez la directive @switch pour gérer les différents cas possibles.

Réponse

TypeScript

```
niveau = 'intermediaire';
```

HTML

```
@switch (niveau) {

 @case ('debutant') {
  <p>Débutant</p>
 }

 @case ('intermediaire') {
  <p>Intermédiaire</p>
 }

 @case ('avance') {
  <p>Avancé</p>
 }

 @default {
  <p>Niveau inconnu</p>
 }

}
```

Exercice 6 – Mise en forme avec ngClass

Après chaque examen, CFITECH souhaite mettre en évidence les résultats des stagiaires.
Si le stagiaire a réussi son examen, son nom doit apparaître en vert.
S'il a échoué, son nom doit apparaître en rouge.

Utilisez ngClass pour appliquer dynamiquement les classes CSS appropriées.

Réponse

TypeScript

```
reussi = true;
```

HTML

```
<p [ngClass]="reussi ? 'succes' : 'echec'">
  Ali
</p>
```

CSS

```
.succes {
  color: green;
}

.echec {
  color: red;
}
```

Exercice 7 – Modifier dynamiquement un style avec ngStyle

Le responsable pédagogique souhaite mettre en avant certaines formations en leur attribuant une couleur particulière.
Créez une variable contenant une couleur et utilisez ngStyle pour appliquer cette couleur au titre de la formation Angular.

Réponse

TypeScript

```
couleur = 'blue';
```

HTML

```
<p [ngStyle]="{'color': couleur}">
  Formation Angular
</p>
```

Exercice 8 – Masquer un élément avec [hidden]

Lorsqu'un cours est terminé, le message :
Cours en cours...
ne doit plus être visible à l'écran.
Utilisez la propriété [hidden] afin de masquer ce message lorsque le cours est terminé.

Réponse

TypeScript

```
coursTermine = true;
```

HTML

```
<p [hidden]="coursTermine">
  Cours en cours...
</p>
```

Exercice 9 – Formulaire interactif avec ngModel

Lors de l'inscription à une formation, un stagiaire doit saisir son prénom dans un champ de saisie.
À mesure qu'il tape son prénom, un message de bienvenue doit être mis à jour automatiquement.
Exemple :
Bienvenue Ali
Utilisez ngModel pour synchroniser automatiquement le champ et la variable TypeScript.

Réponse

TypeScript

```
prenom = '';
```

HTML

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

<p>
  Bienvenue {{ prenom }}
</p>
```

Exercice 10 – Mini projet récapitulatif

CFITECH souhaite réaliser une petite interface d'inscription.
L'utilisateur doit pouvoir saisir son prénom dans un champ de formulaire.
Lorsque le champ contient une valeur :

- un message de bienvenue doit s'afficher ;
- le texte doit apparaître en vert ;
- un message indiquant que l'inscription est en cours doit être visible.

Lorsque le champ est vide :

- le message de bienvenue ne doit pas être affiché ;
- le message d'inscription doit être masqué.

Pour réaliser cet exercice, combinez :

- ngModel
- @if
- ngClass
- [hidden]

Réponse

TypeScript

```
nom = '';
```

HTML

```
<input
  type="text"
```

```
 [(ngModel)]="nom"
>

@if (nom) {

 <p [ngClass]="'valide'">
  Bienvenue {{ nom }}
 </p>

}

<p [hidden]="!nom">
 Votre inscription est en cours.
</p>
```

CSS

```
.valide {
 color: green;
}
```

Chapitre 7: Routing (Navigation moderne)

Objectifs

- Comprendre le rôle du Routing dans une application Angular
- Créer plusieurs pages dans une application
- Configurer des routes
- Utiliser router-outlet
- Naviguer avec routerLink
- Comprendre le fonctionnement du Routing avec les Standalone Components
- Construire une navigation professionnelle

1. **Pourquoi avons-nous besoin du Routing ?**

Jusqu'à présent, nos applications contenaient généralement un seul composant principal :
et tous les autres composants étaient affichés directement dans :
app.html
Par exemple :

```
<app-header></app-header>

<app-accueil></app-accueil>

<app-footer></app-footer>
```

Définition
Le Routing est le mécanisme qui permet à Angular d'afficher différents composants selon l'URL visitée.

Exemple
Lorsque l'utilisateur visite :
http://localhost:4200/

Angular affiche :

Accueil

Lorsque l'utilisateur visite :

http://localhost:4200/formations

Angular affiche :

Formations

Lorsque l'utilisateur visite :

http://localhost:4200/inscription

Angular affiche :

Inscription

2. Architecture du Routing

Sans Routing

App
├─ Header
├─ Accueil
├─ Formations
├─ Profil
├─ Inscription
└─ Footer
Tout est affiché.

Avec Routing :
App
├─ Header
├─ Router Outlet
└─ Footer
Le contenu central change selon l'URL.

3. Création des composants

Créons plusieurs pages :

ng generate component pages/accueil

ng generate component pages/formations

ng generate component pages/inscription

ng generate component pages/profil

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech>ng g c pages/accueil
CREATE src/app/pages/accueil/accueil.spec.ts (561 bytes)
CREATE src/app/pages/accueil/accueil.ts (201 bytes)
CREATE src/app/pages/accueil/accueil.css (0 bytes)
CREATE src/app/pages/accueil/accueil.html (23 bytes)

D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech>ng g c pages/formations
CREATE src/app/pages/formations/formations.spec.ts (582 bytes)
CREATE src/app/pages/formations/formations.ts (213 bytes)
CREATE src/app/pages/formations/formations.css (0 bytes)
CREATE src/app/pages/formations/formations.html (26 bytes)

D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech>ng g c pages/inscriptions
CREATE src/app/pages/inscriptions/inscriptions.spec.ts (596 bytes)
CREATE src/app/pages/inscriptions/inscriptions.ts (221 bytes)
CREATE src/app/pages/inscriptions/inscriptions.css (0 bytes)
CREATE src/app/pages/inscriptions/inscriptions.html (28 bytes)

D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech>ng g c pages/profil
CREATE src/app/pages/profil/profil.spec.ts (554 bytes)
CREATE src/app/pages/profil/profil.ts (197 bytes)
CREATE src/app/pages/profil/profil.css (0 bytes)
CREATE src/app/pages/profil/profil.html (22 bytes)

D:\CFITECH\Angular\Cours angular  2026\Exercices\cfitech>
```

4. Création des routes

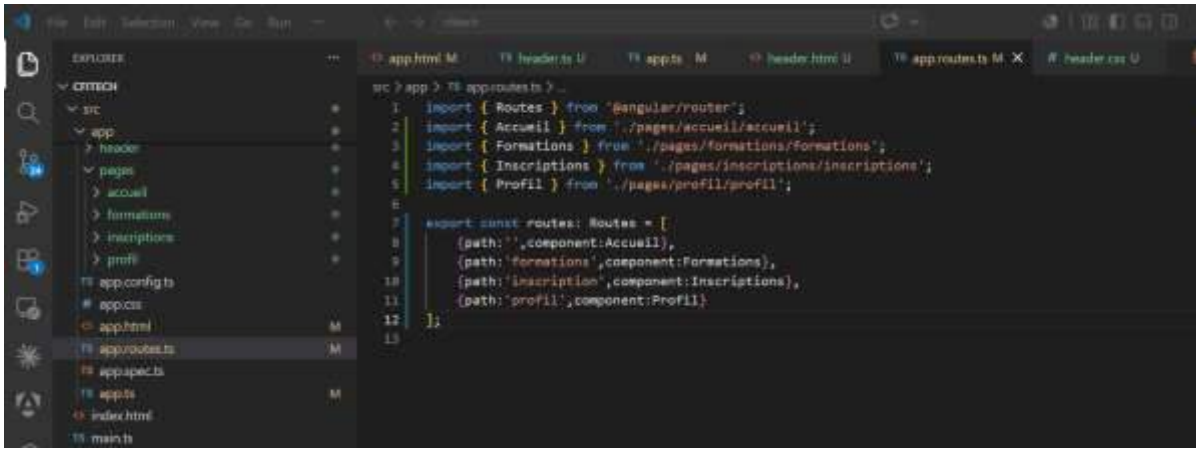
Dans Angular 21, les routes sont généralement créées dans le fichier :
src/app/app.routes.ts

C'est le fichier spécialement prévu pour centraliser toutes les routes de l'application.

Exemple :

```
import { Routes } from '@angular/router';

import { Accueil } from './pages/accueil/accueil';
import { Formations } from './pages/formations/formations';
import { Inscription } from './pages/inscription/inscription';
import { Profil } from './pages/profil/profil';

export const routes: Routes = [

 {
  path: '',
  component: Accueil
 },

 {
  path: 'formations',
  component: Formations
 },

 {
  path: 'inscription',
  component: Inscription
 },
```

```
 {
   path: 'profil',
   component: Profil
 }

];
```



Comprendre une route

Exemple :
{
 path: 'formations',
 component: Formations
}

Signifie :
URL :

/formations

↓

Afficher :

Formations Component

5. Le rôle de router-outlet

C'est l'élément le plus important du Routing.

Dans :

app.html
on écrit :

<app-header></app-header>

<router-outlet></router-outlet>

<app-footer></app-footer>

Que fait router-outlet ?

router-outlet est une zone réservée dans laquelle Angular affiche automatiquement le composant correspondant à la route active.
Autrement dit, c'est l'endroit où les pages vont apparaître.

Il agit comme une zone de remplacement dynamique.
Angular y injecte automatiquement le composant correspondant à la route.

RouterOutlet est un **composant natif fourni par Angular Router**. Il n'est pas créé par le développeur et il ne fait pas partie de HTML. C'est Angular qui le fournit pour permettre l'affichage dynamique des composants associés aux routes.

Exemple
URL :
/
Angular injecte :
<app-accueil>

URL :
/formations
Angular injecte :
<app-formations>

URL :
/profil
Angular injecte :
<app-profil>

6. Import du RouterOutlet
Dans App :

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

@Component({
  standalone: true,

  imports: [
    RouterOutlet
  ]
})
```

Où placer router-outlet ?

On place généralement :

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

dans le composant principal de l'application :

app.html

Exemple :

```
<app-header></app-header>

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

<app-footer></app-footer>

Où importer RouterOutlet ?

router-outlet est un composant Angular.
Comme tous les composants standalone, il doit être importé.
Dans :

app.ts

(selon le nom de ton projet)

Exemple :

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

@Component({
 selector: 'app-root',

 standalone: true,

 imports: [
  RouterOutlet
 ],

 templateUrl: './app.html'
})
export class App {

}
```

Dans les versions modernes d'Angular (Angular 17, 18, 19, 20, 21), **si tu crées ton projet avec le Routing activé**, Angular ajoute automatiquement les éléments nécessaires lors de la génération du projet.

Par exemple :

ng new mon-projet --routing

ou
lorsque Angular te pose la question :

Would you like to add Angular routing?

et que tu réponds :

Yes

Angular génère automatiquement :

app.config.ts

```
providers: [
  provideRouter(routes)
]
```

app.routes.ts

```
export const routes: Routes = [];
```

app.ts

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

@Component({
  standalone: true,
  imports: [RouterOutlet]
})
```

app.html

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

Donc dans le cas tout est déjà configuré, **tu n'as rien à faire**,.

Par contre,
si tu as créé ton projet **sans Routing** :

ng new mon-projet

ou si tu as répondu :

No

à la question concernant le Routing, alors Angular ne crée pas :

- app.routes.ts
- provideRouter()
- RouterOutlet
- <router-outlet>

Tu devras tout ajouter manuellement.

Lorsqu'un projet Angular est créé avec l'option Routing activée, Angular configure automatiquement les fichiers nécessaires à la navigation, notamment app.routes.ts, provideRouter() dans app.config.ts, l'import de RouterOutlet dans app.ts ainsi que la balise <router-outlet> dans le template principal. Si le projet est créé sans Routing, ces éléments ne sont pas générés et doivent être ajoutés manuellement par le développeur. C'est pourquoi il est important de vérifier dès la création du projet si le Routing a été activé ou non.

6. Navigation avec routerLink

routerLink est une directive native d'Angular Router qui permet de **naviguer d'une route à une autre sans recharger la page**.

Autrement dit, lorsqu'un utilisateur clique sur un lien utilisant routerLink, Angular change l'URL, recherche la route correspondante dans app.routes.ts et affiche le composant associé dans le router-outlet.

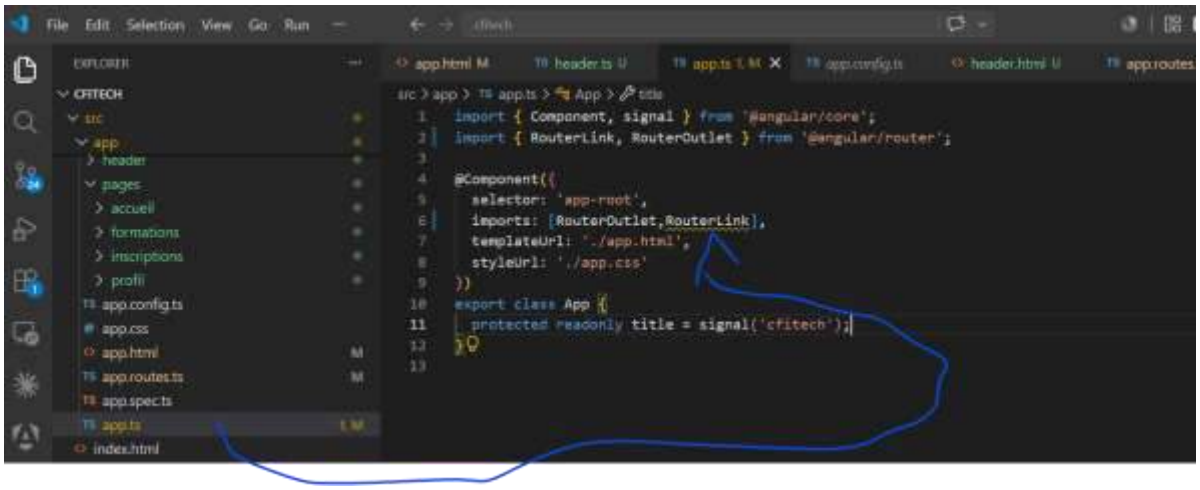
RouterLink est une directive native d'Angular Router qui permet de naviguer entre les différentes pages de l'application. Dans un projet Angular moderne basé sur les Standalone Components, cette directive n'est pas disponible automatiquement. Elle doit être importée dans chaque composant qui utilise routerLink. Ainsi, RouterLink sert à créer les liens de navigation, tandis que RouterOutlet sert à afficher le composant correspondant à la route sélectionnée.

Une fois les routes créées, il faut permettre à l'utilisateur de naviguer.

7.Import de RouterLink

Dans un composant standalone :
import { RouterLink } from '@angular/router';

```
@Component({
 standalone: true,

 imports: [
  RouterLink
 ]
})
```



Ancienne méthode HTML :

```
<a href="/formations">
 Formations
</a>
```

Ce n'est pas recommandé dans Angular.

Angular fournit :
routerLink

Exemple

```
<a routerLink="/">
 Accueil
</a>
```

```
<a routerLink="/formations">
  Formations
</a>

<a routerLink="/inscription">
  Inscription
</a>

<a routerLink="/profil">
  Profil
</a>
```

Les liens de navigation sont généralement créés dans le composant qui représente le **menu de l'application**.
Dans un projet professionnel, on ne met généralement pas les liens directement dans app.html. On crée plutôt un composant :

Par ex :Tu utilises routerLink dans Header

header.html

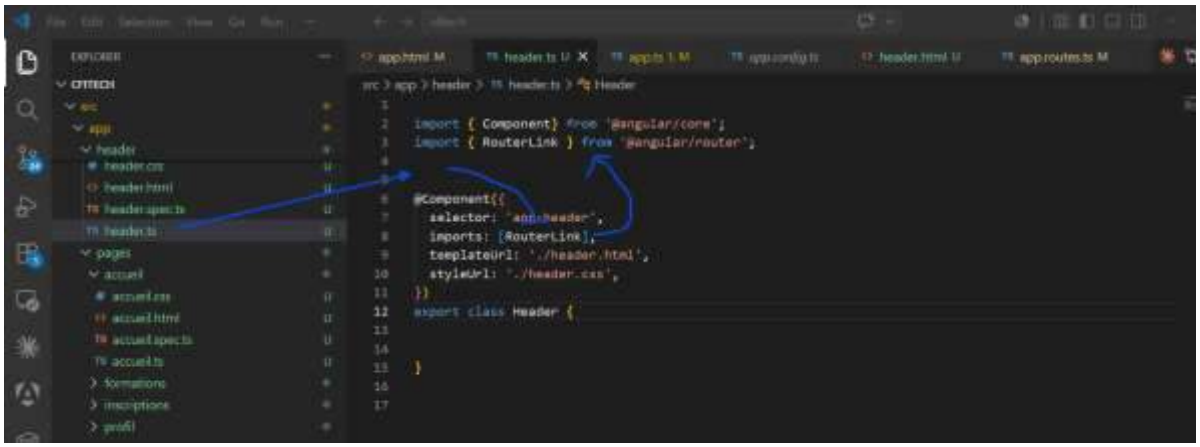
```
<a routerLink="/">Accueil</a>
<a routerLink="/formations">Formations</a>
<a routerLink="/profil">Profil</a>
```

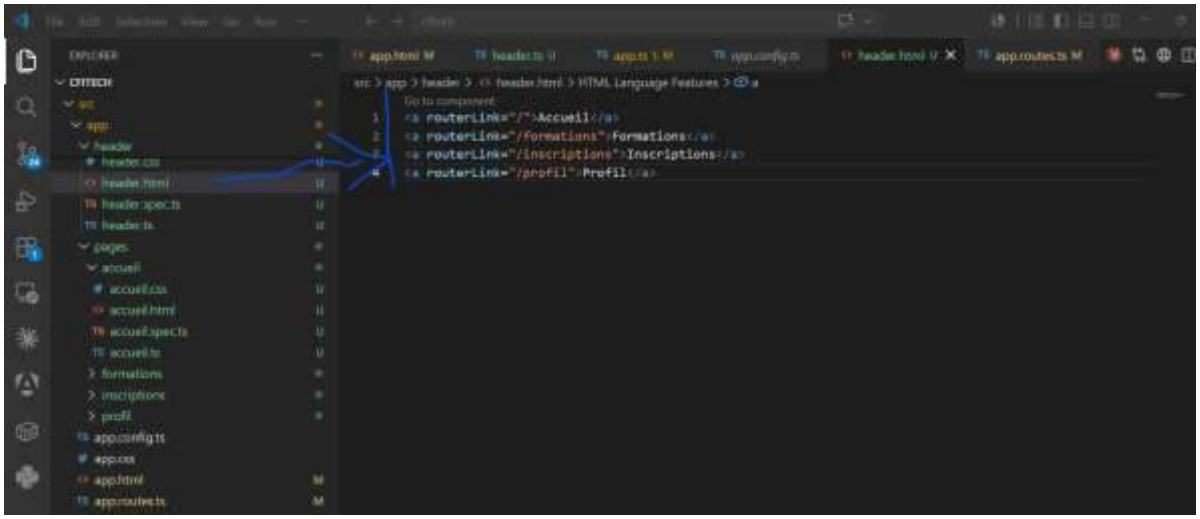
Dans ce cas, c'est **Header** qui doit importer RouterLink :

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

@Component({
  selector: 'app-header',
  standalone: true,

  imports: [
    RouterLink
  ],

  templateUrl: './header.html'
})
export class Header {

}
```

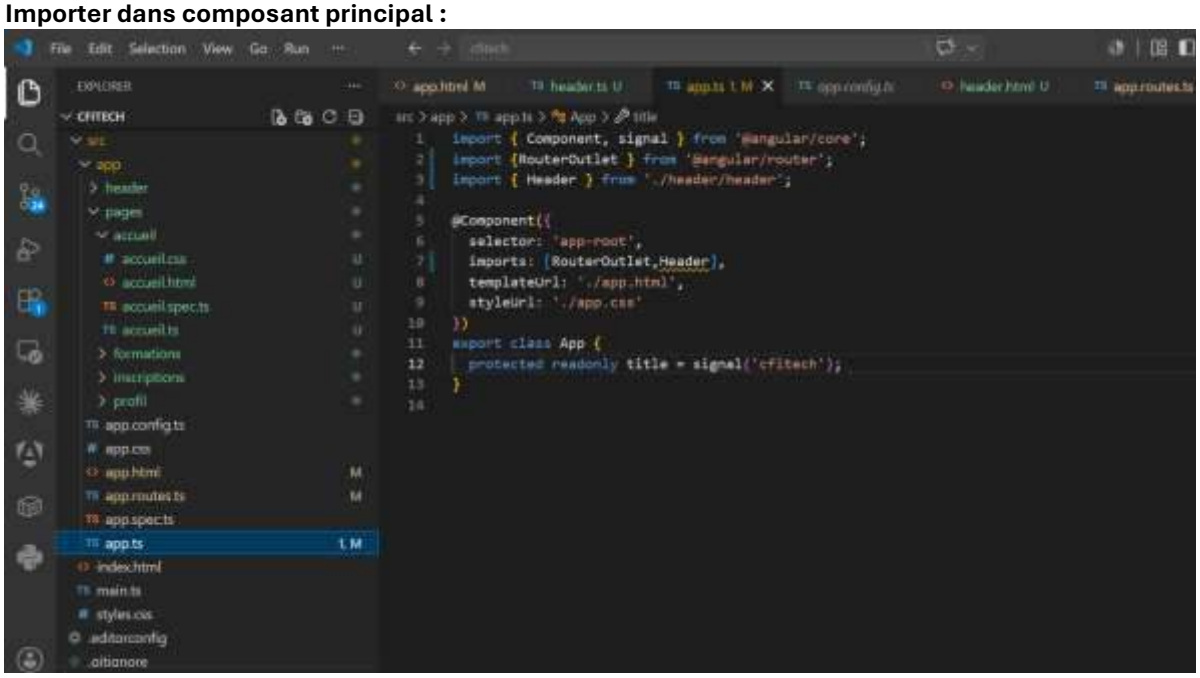
Et **App** n'a pas besoin d'importer RouterLink si App ne l'utilise pas.

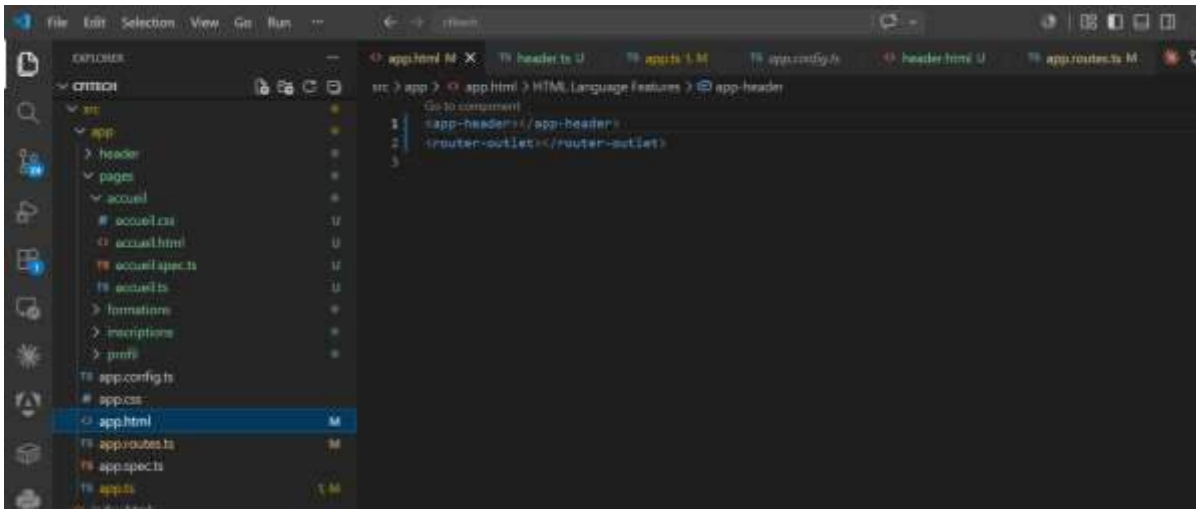
Dans les Standalone Components, chaque composant doit importer lui-même les directives et composants qu'il utilise dans son template. Si un composant contient des liens routerLink, il doit importer RouterLink. Si un composant contient <router-outlet>, il doit importer RouterOutlet. Les imports ne sont pas automatiquement hérités des autres composants. C'est l'un des principes fondamentaux de l'architecture standalone d'Angular moderne.
dans la plupart des applications Angular, RouterOutlet est importé dans le composant principal (App), car c'est généralement dans app.html que l'on place

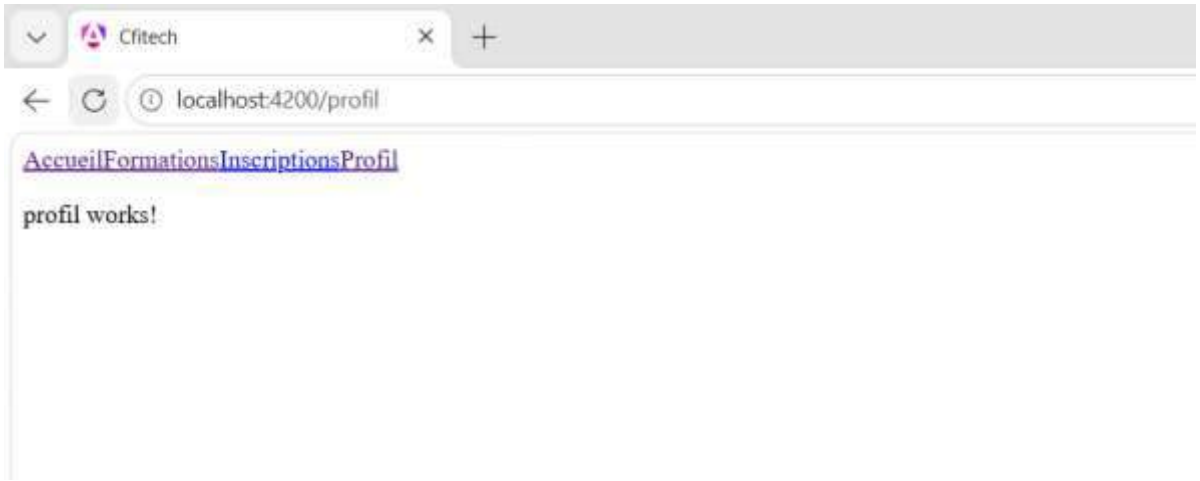




Importer dans composant principal :







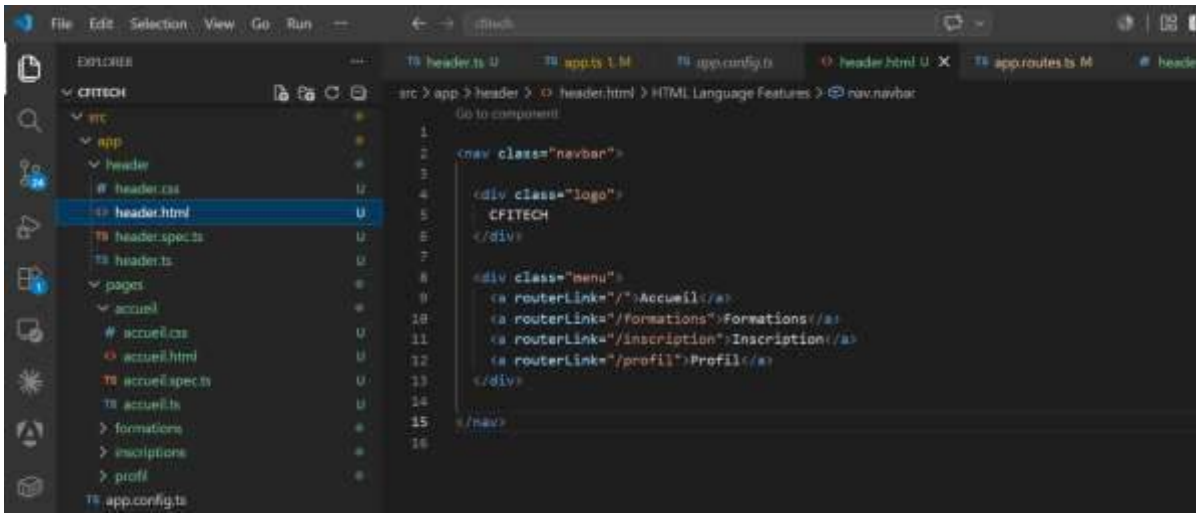
Pour un projet Angular moderne CFITECH, tu peux créer un menu professionnel simple avec Flexbox.

header.html

```
<nav class="navbar">

 <div class="logo">
  CFITECH
 </div>

 <div class="menu">
  <a routerLink="/">Accueil</a>
  <a routerLink="/formations">Formations</a>
  <a routerLink="/inscription">Inscription</a>
  <a routerLink="/profil">Profil</a>
 </div>

</nav>
```

header.css

```
/* Barre de navigation */

.navbar {
  display: flex;
  justify-content: space-between;
  align-items: center;

  background-color: #1e293b;
  padding: 15px 40px;

  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.2);
}

/* Logo */

.logo {
  color: white;
  font-size: 24px;
  font-weight: bold;
}

/* Conteneur des liens */

.menu {
  display: flex;
  gap: 25px;
}

/* Liens */

.menu a {
  color: white;
  text-decoration: none;

  font-size: 16px;
  font-weight: 500;

  padding: 8px 12px;
  border-radius: 5px;

  transition: 0.3s;
}

/* Hover */

.menu a:hover {
  background-color: #3b82f6;
}
```

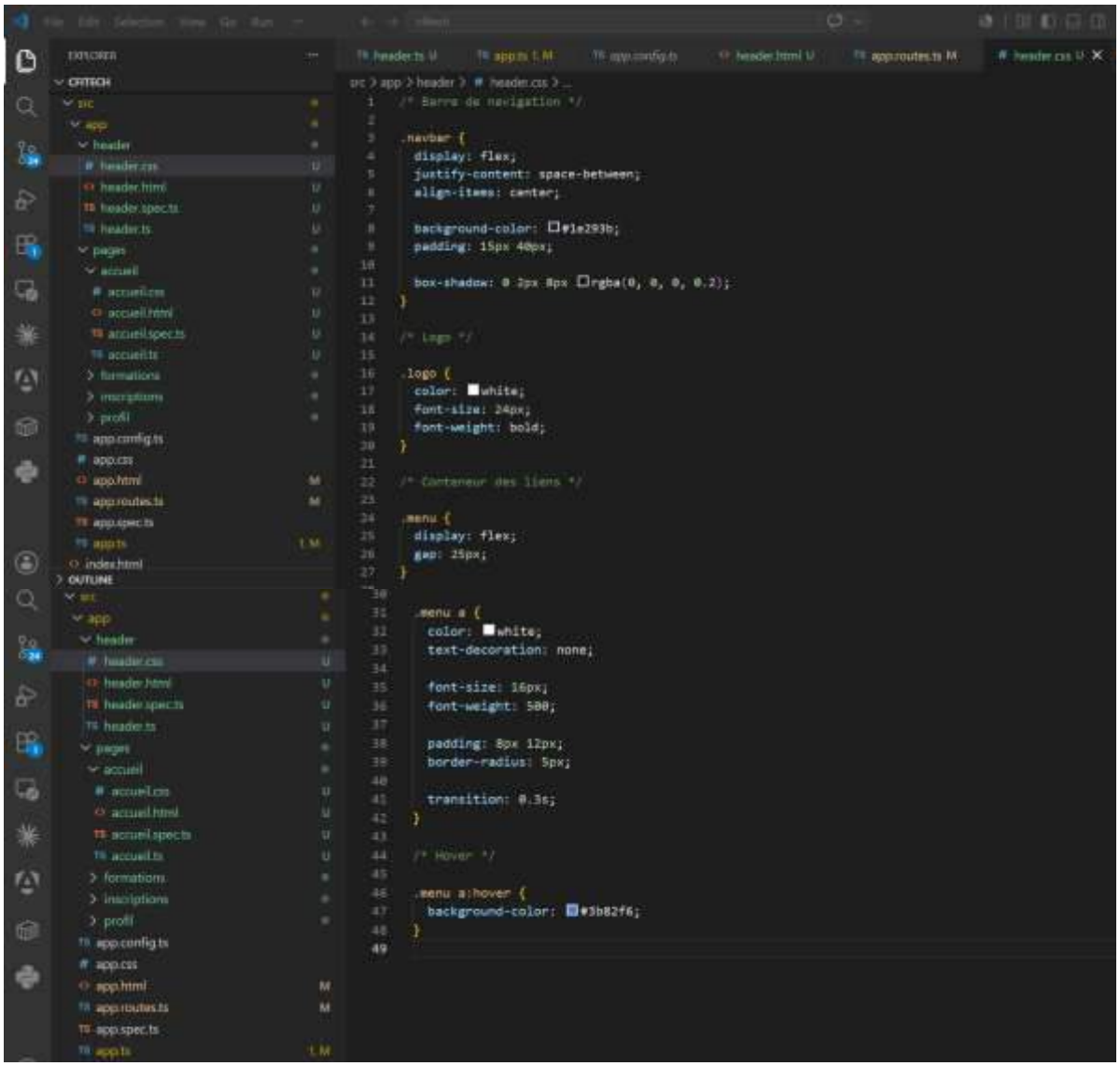

File Edit Selection View Go Run
EXPLORER
CFITECH
src
app
header
header.html
header.spec.ts
header.ts
pages
accueil
accueil.html
accueil.spec.ts
accueil.ts
formations
inscriptions
profil
app.config.ts
header.ts U
app.ts 1, M
app.config.ts
header.html U
app.routes.ts M
src > app > header > header.html > HTML Language Features > nav.navbar
Go to component
<nav class="navbar">

CFITECH

<a routerLink="/">Accueil
<a routerLink="/formations">Formations
<a routerLink="/inscription">Inscription
<a routerLink="/profil">Profil

</nav>

Header.css


EXPLORER
CFITECH
header.ts U
app.ts 1, M
app.config.ts
header.html U
app.routes.ts M
header.css U
src > app > header > header.css > ...
/* Barre de navigation */
.navbar {
display: flex;
justify-content: space-between;
align-items: center;
background-color: #1e293b;
padding: 15px 40px;
box-shadow: 0 2px 8px rgba(0, 0, 0, 0.2);
}
/* Logo */
.logo {
color: white;
font-size: 24px;
font-weight: bold;
}
/* Conteneur des liens */
.menu {
display: flex;
gap: 25px;
}
.menu a {
color: white;
text-decoration: none;
font-size: 16px;
font-weight: 500;
padding: 8px 12px;
border-radius: 5px;
transition: 0.3s;
}
/* Hover */
.menu a:hover {
background-color: #3b82f6;
}

Le Routing permet de créer des applications multi-pages tout en conservant le fonctionnement SPA d'Angular. Les routes sont définies dans app.routes.ts et associent une URL à un composant. Le composant spécial router-outlet agit comme une zone dynamique dans laquelle Angular affiche le composant correspondant à la route active. La navigation s'effectue grâce à routerLink, qui permet de changer de page sans recharger l'application. Dans Angular 21, les routes travaillent directement avec les Standalone Components, ce qui simplifie considérablement l'architecture des applications.

Chapitre 8 : Services & Injection de dépendances

Objectifs

- partager des données entre plusieurs composants ;
- comprendre l'injection de dépendances ;
- utiliser inject() (approche moderne Angular) ;
- communiquer entre composants grâce aux services.

1. Pourquoi les Services existent-ils ?

Jusqu'à présent, nos composants possèdent leurs propres données :

```
export class Inscription {

 nom = '';
 prenom = '';

}
```

Le problème est le suivant :

- Inscription possède ses données.
- Profil possède ses données.
- Dashboard possède ses données par exemple.

Les composants ne se connaissent pas.

Exemple

Page Inscription :

Ali
Angular

Page Profil :

???

Page Dashboard :

???

Chaque composant possède sa propre mémoire.

Définition

Un service est une classe Angular spécialisée qui permet de centraliser des données ou des traitements afin qu'ils puissent être utilisés par plusieurs composants.

Analogie CFITECH

Imaginez le secrétariat du centre CFITECH.

Stagiaire
↓
Secrétariat
↓
Accueil
Profil
Dashboard
Administration

Tous les services du centre consultent le même dossier.

Le service Angular joue exactement ce rôle.

Sans Service

Inscription
↓
données locales

Profil
↓

données locales

Dashboard
↓
données locales

Impossible de partager.

Avec Service

Inscription
↓
Service
↙ ↘
Profil Dashboard

Tout le monde utilise les mêmes données.

2. Création d'un Service

Commande :

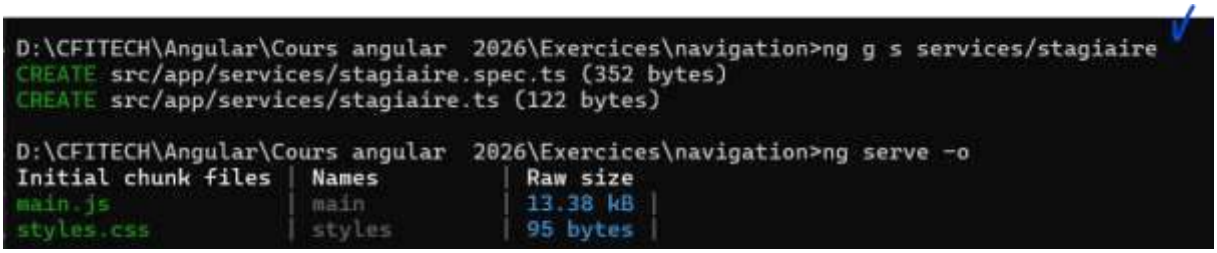
ng generate service services/stagiaire

ou :

ng g s services/stagiaire

Angular crée :

stagiaire.service.ts



Structure d'un service

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

@Injectable({
  providedIn: 'root'
})
```

```
export class StagiaireService {

}
```

Le rôle de @Injectable

Il indique à Angular :

Cette classe peut être injectée dans d'autres composants.

Que signifie :

providedIn: 'root'

Cela signifie :

Angular crée une seule instance du service pour toute l'application.

Tous les composants partageront donc les mêmes données.

Exemple simple

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

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

  nom = 'Ali';

}
```

Utilisation dans un composant

Ancienne méthode :

```
constructor(
  private service: StagiaireService
) {}
```

3. inject() : l'approche moderne

Depuis Angular 14+, Angular recommande :

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

service = inject(StagiaireService);

Exemple complet

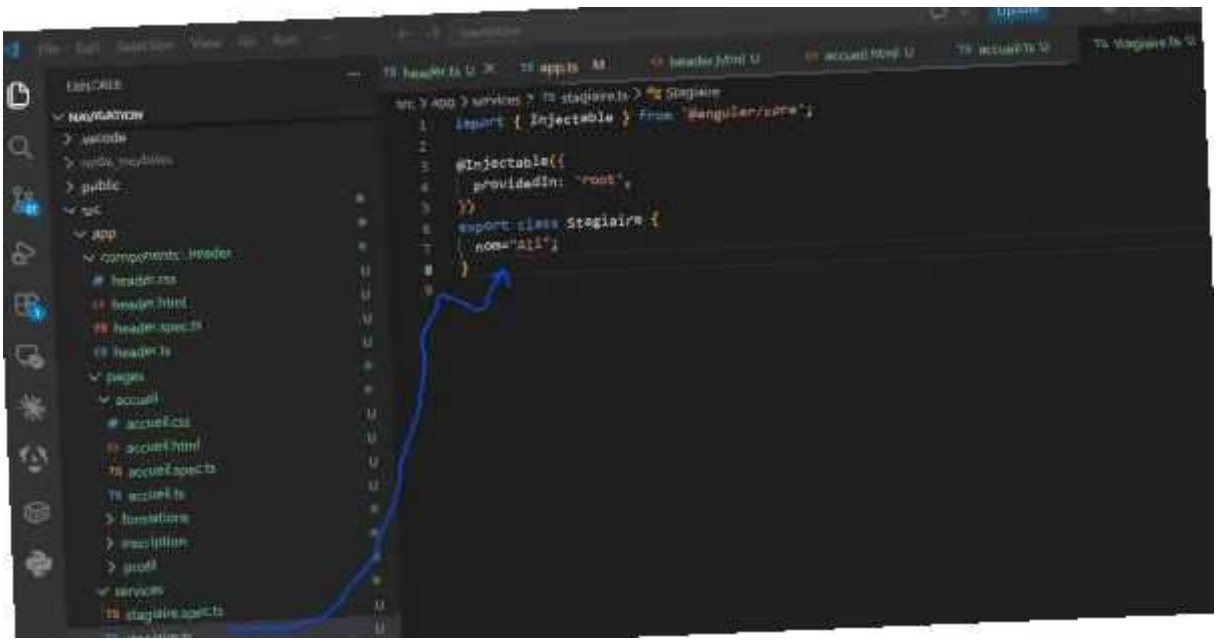
```
import { Component, inject } from '@angular/core';
import { StagiaireService } from '../services/stagiaire.service';

@Component({
  standalone: true
})
export class Profil {

  service = inject(StagiaireService);

}
```

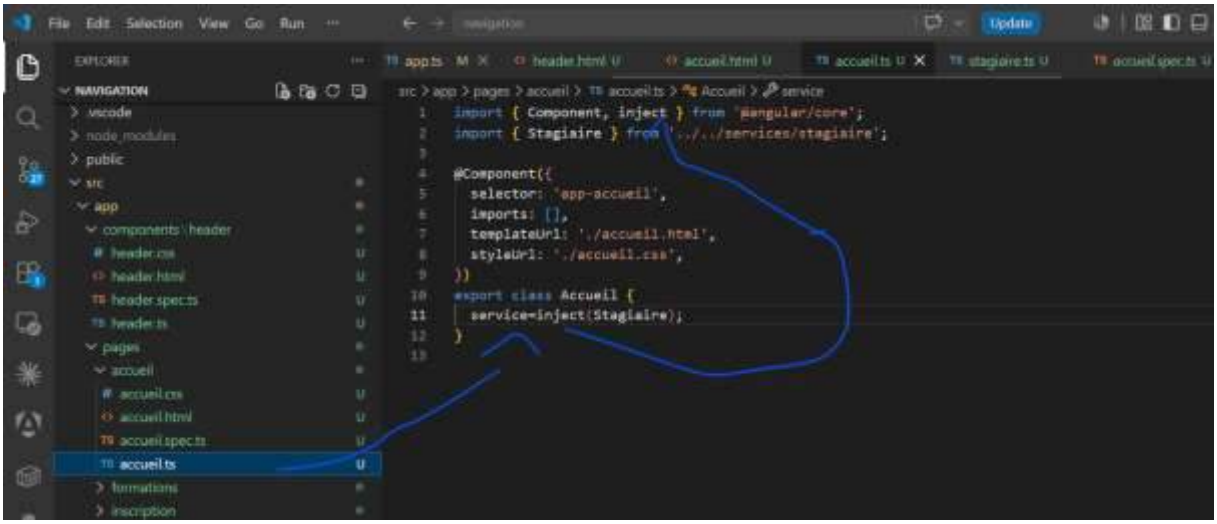
Utilisation dans le HTML

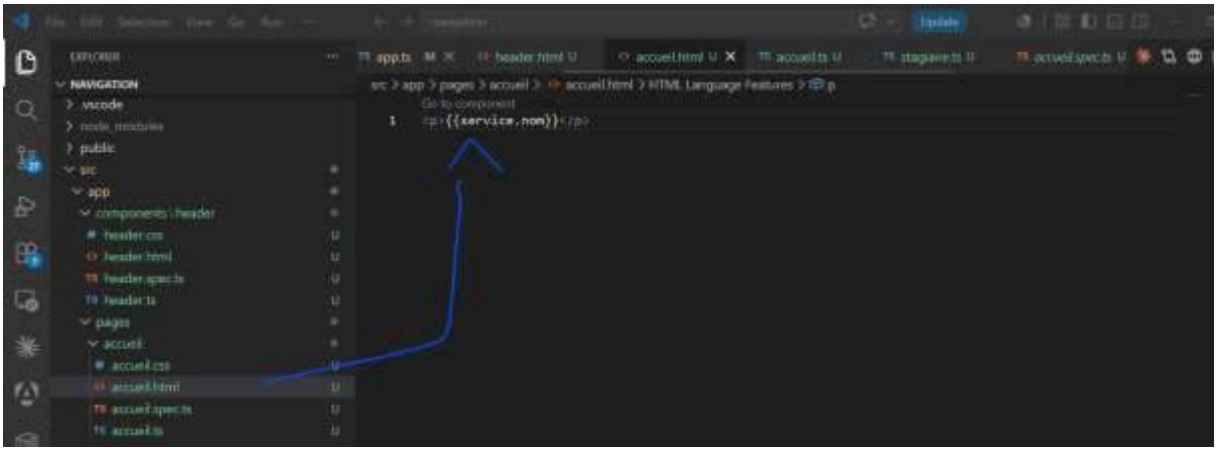
{{ service.nom }}

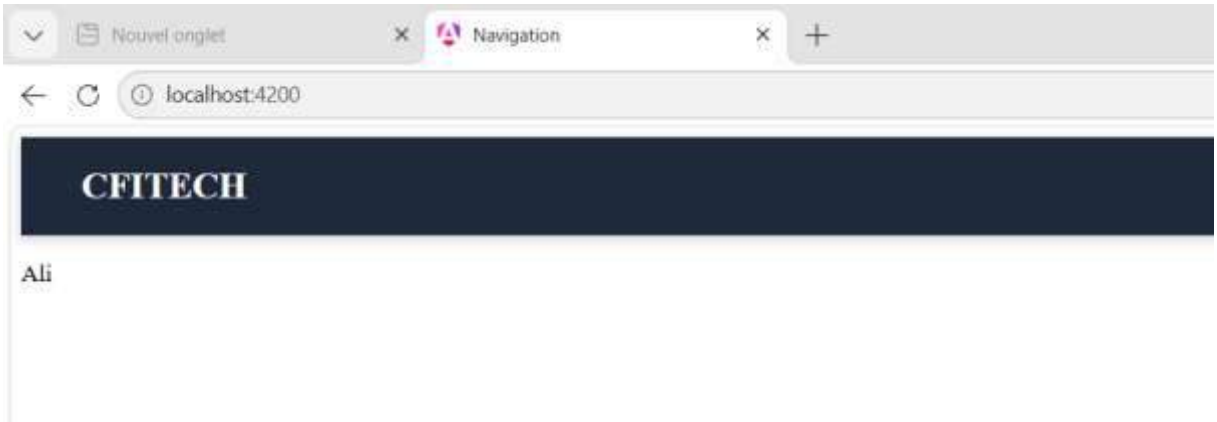
Résultat :

Ali









Pourquoi inject() ?

Ancienne méthode :

```
constructor(
  private service: StagiaireService
){}
```

Nouvelle méthode :

```
service = inject(StagiaireService);
```

Avantages :

- plus simple ;
- moins de code ;
- meilleure lisibilité ;
- compatible avec Angular moderne.

4. Communication entre composants

Imaginons :

Inscription

```
service.nom = 'Ali';
```

Profil

```
<p>{{ service.nom }}</p>
```

Dashboard

```
<p>{{ service.nom }}</p>
```

Résultat :

Inscription :
Ali

Profil :
Ali

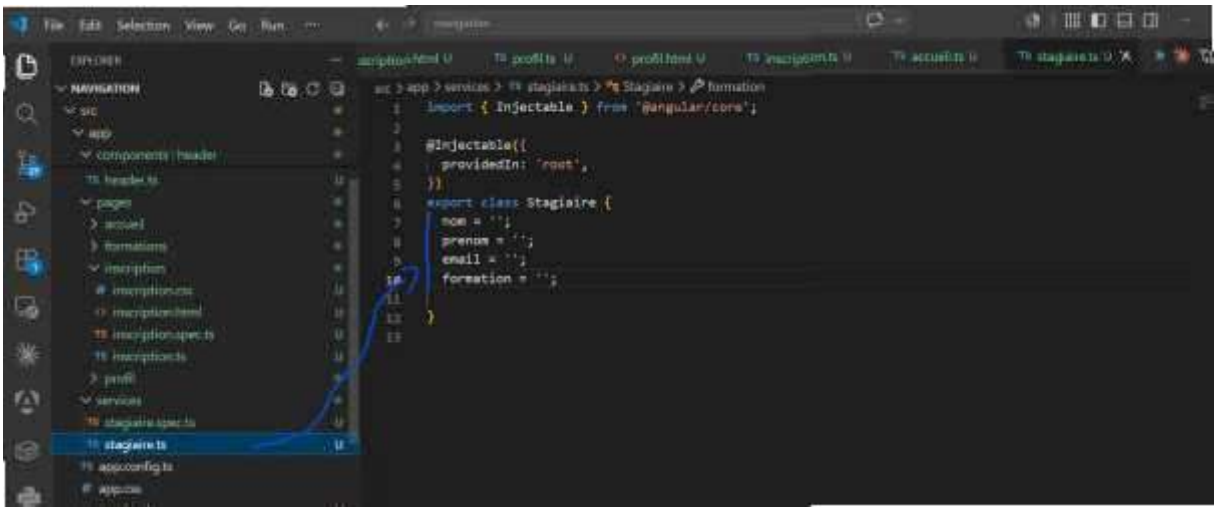
Dashboard :
Ali

Le service devient une mémoire commune.

Exemple CFITECH

stagiaire.service.ts

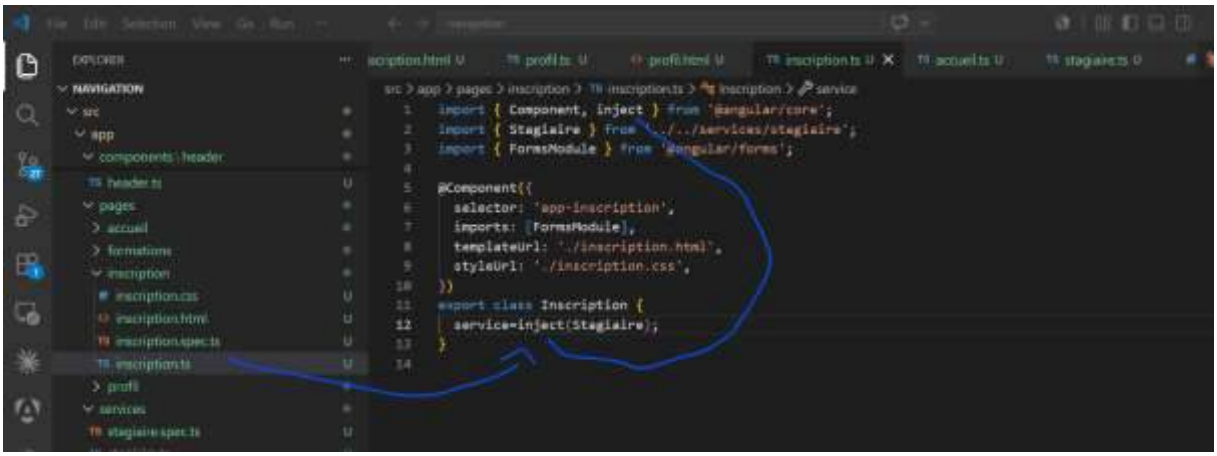
```
import { Injectable } from '@angular/core';

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

 nom = '';
 prenom = '';
 email = '';
 formation = '';

}
```



inscription.ts

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

service = inject(StagiaireService);

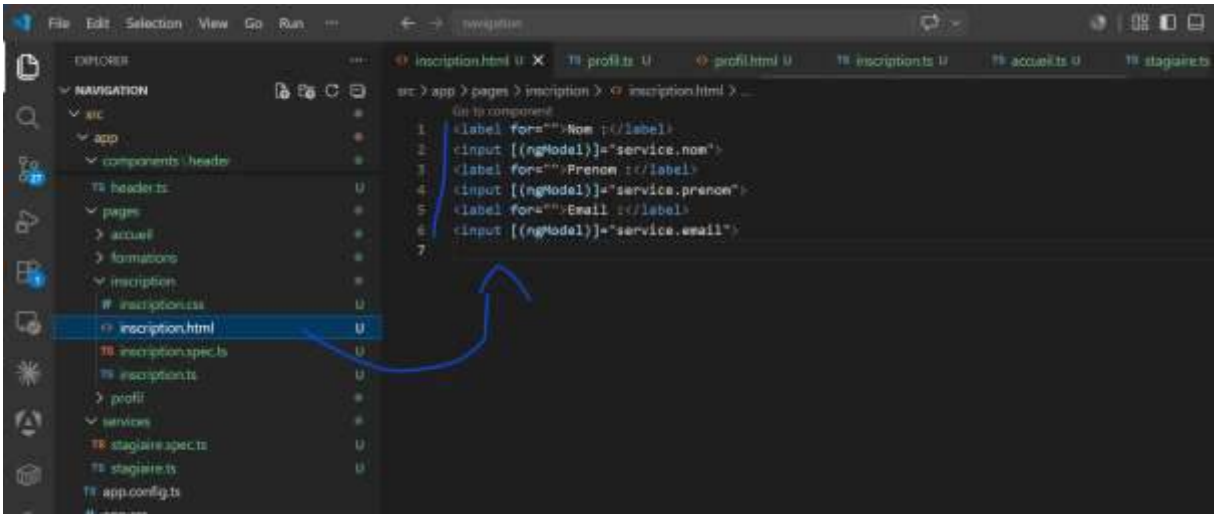


inscription.html

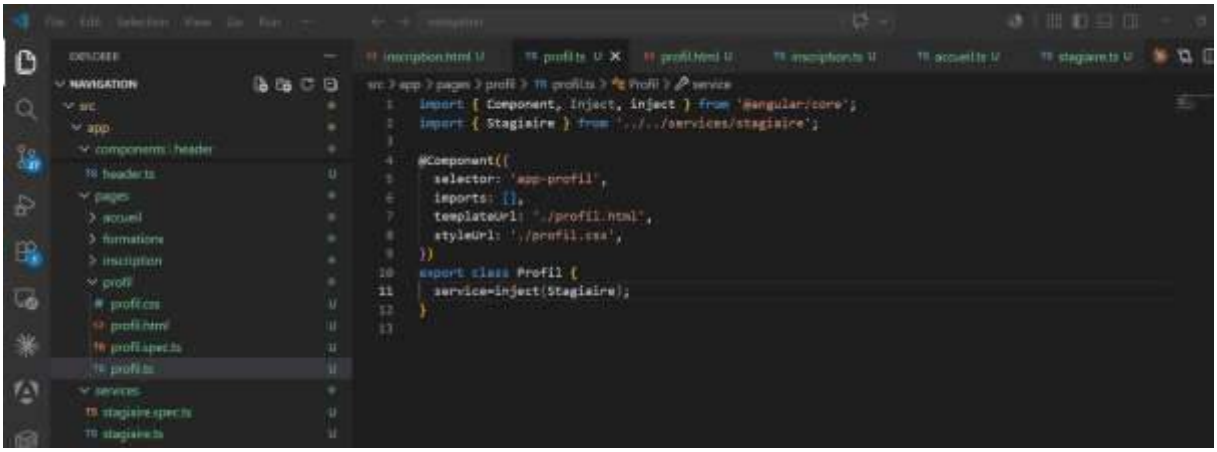
<input [(ngModel)]="service.nom">

<input [(ngModel)]="service.prenom">

<input [(ngModel)]="service.email">

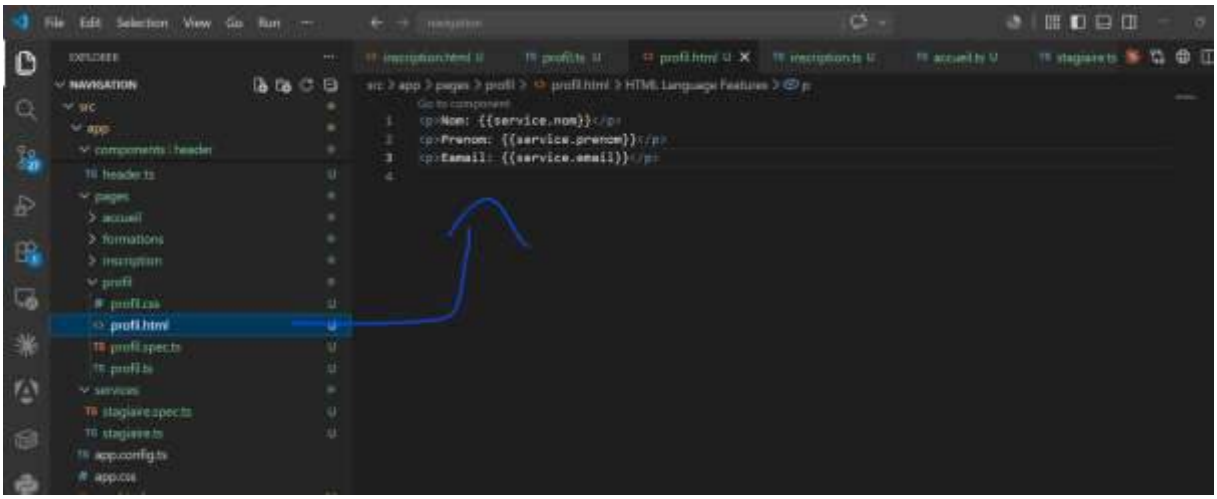


profil.ts



profil.html

```
<p>Nom : {{ service.nom }}</p>

<p>Prénom : {{ service.prenom }}</p>

<p>Email : {{ service.email }}</p>
```



dashboard.html

```
<h2>Tableau de bord</h2>

<p>{{ service.nom }}</p>

<p>{{ service.formation }}</p>
```

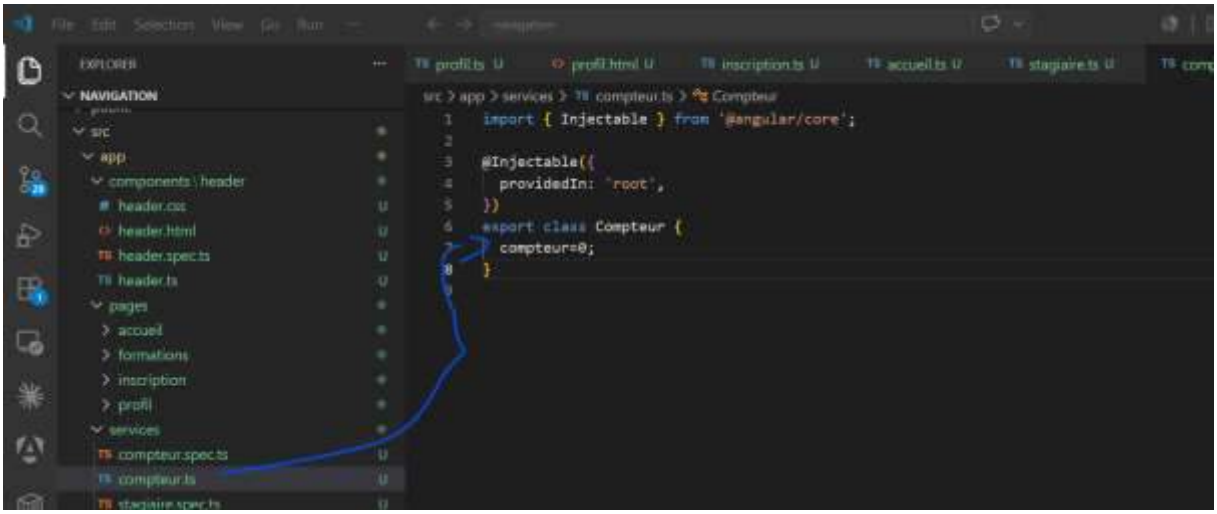
Services et Signal

Avant les Signals dans un service

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g s services/compteur
CREATE src/app/services/compteur.spec.ts (347 bytes)
CREATE src/app/services/compteur.ts (121 bytes)
```

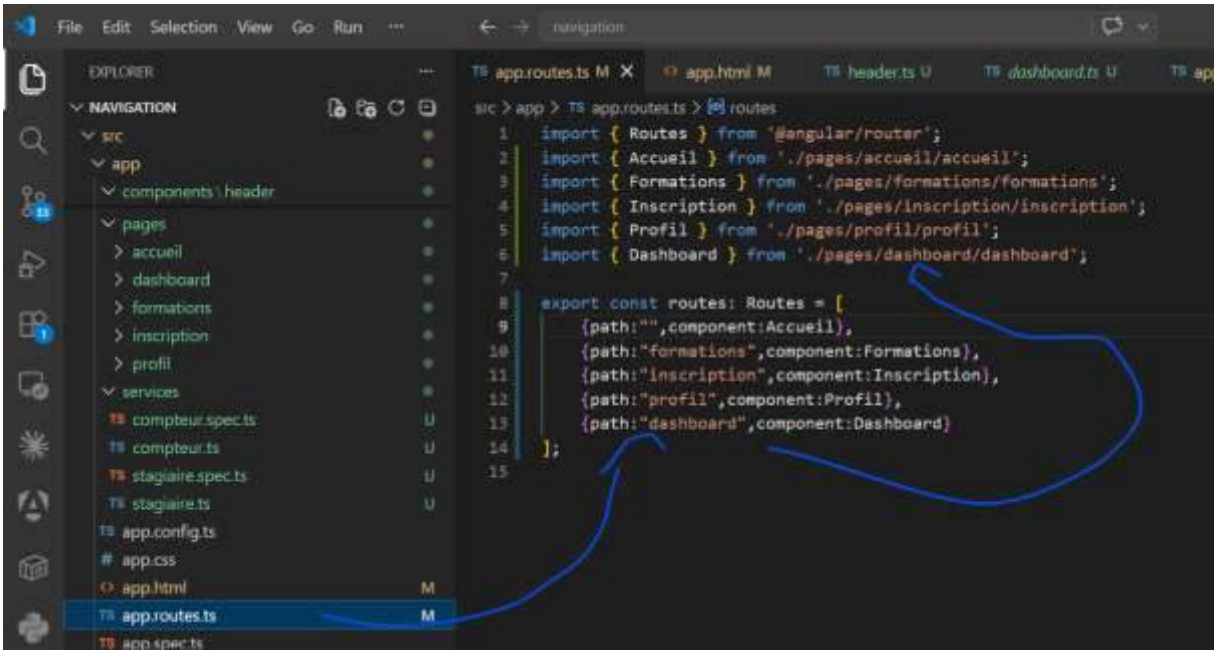
Supposons un service CFITECH :

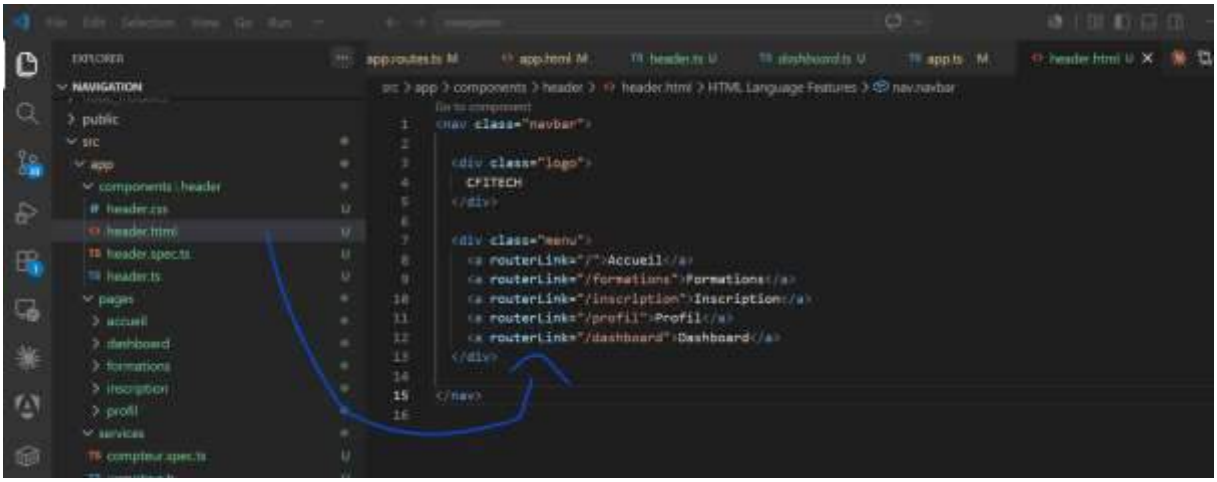
```
@Injectable({
  providedIn: 'root'
})
export class CompteurService {

  compteur = 0;

}
```



Composant Dashboard : Créer et ajouter le composant Dashboard sur le menu.

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g c pages/dashboard
CREATE src/app/pages/dashboard/dashboard.spec.ts (575 bytes)
CREATE src/app/pages/dashboard/dashboard.ts (209 bytes)
CREATE src/app/pages/dashboard/dashboard.css (0 bytes)
CREATE src/app/pages/dashboard/dashboard.html (25 bytes)
```

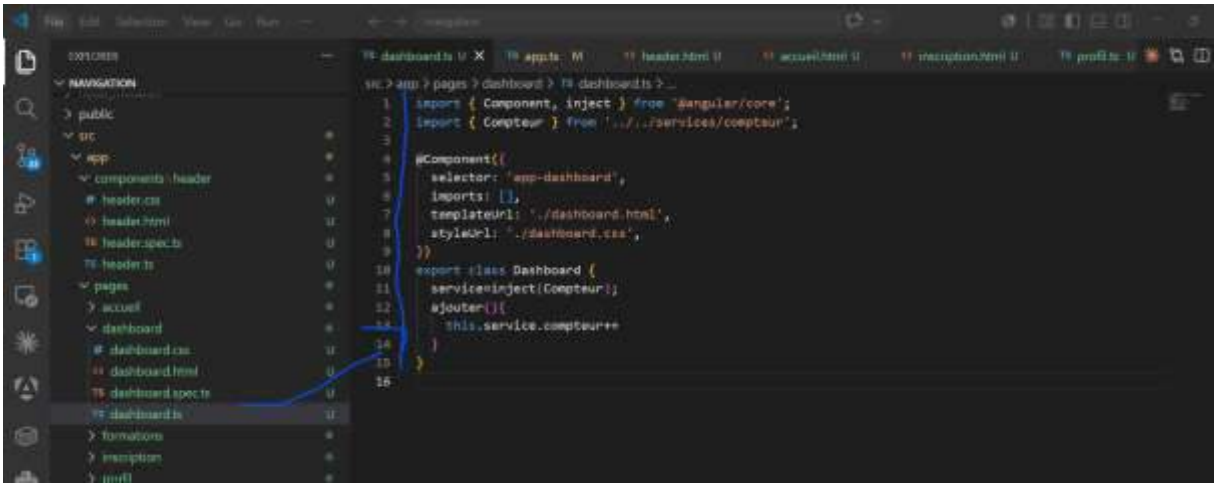




Dans dashboard.ts

service = inject(CompteurService);

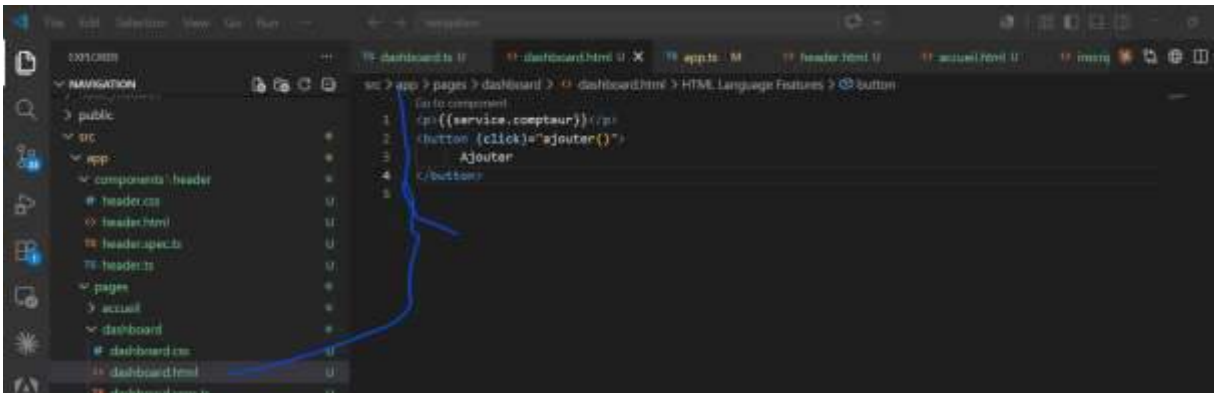
ajouter() {
 this.service.compteur++;
}



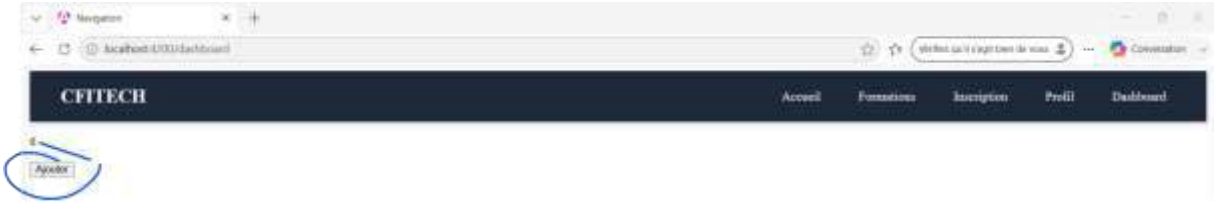
HTML :

{{ service.compteur }}

<button (click)="ajouter()">
Ajouter
</button>



Cela fonctionne.



Mais imaginons maintenant :

Dashboard
Profil
Accueil

Tous affichent :

{{ service.compteur }}

Si le compteur change dans Dashboard :

this.service.compteur++;

Angular doit lancer son système de détection des changements pour retrouver les composants concernés.

2. Le service devient réactif

Angular moderne propose :

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

  compteur = signal(0);

}
```

Le service ne stocke plus une variable.

Il stocke un Signal.

Analogie CFITECH

Ancien système

Le secrétariat annonce :

"Quelque chose a changé."

Tous les bureaux vérifient leurs dossiers.

Accueil
Profil
Dashboard
Administration

Tout le monde travaille.

Avec Signal

Le secrétariat annonce :

"Le compteur a changé."

Angular sait exactement :

Dashboard utilise compteur

Profil utilise compteur

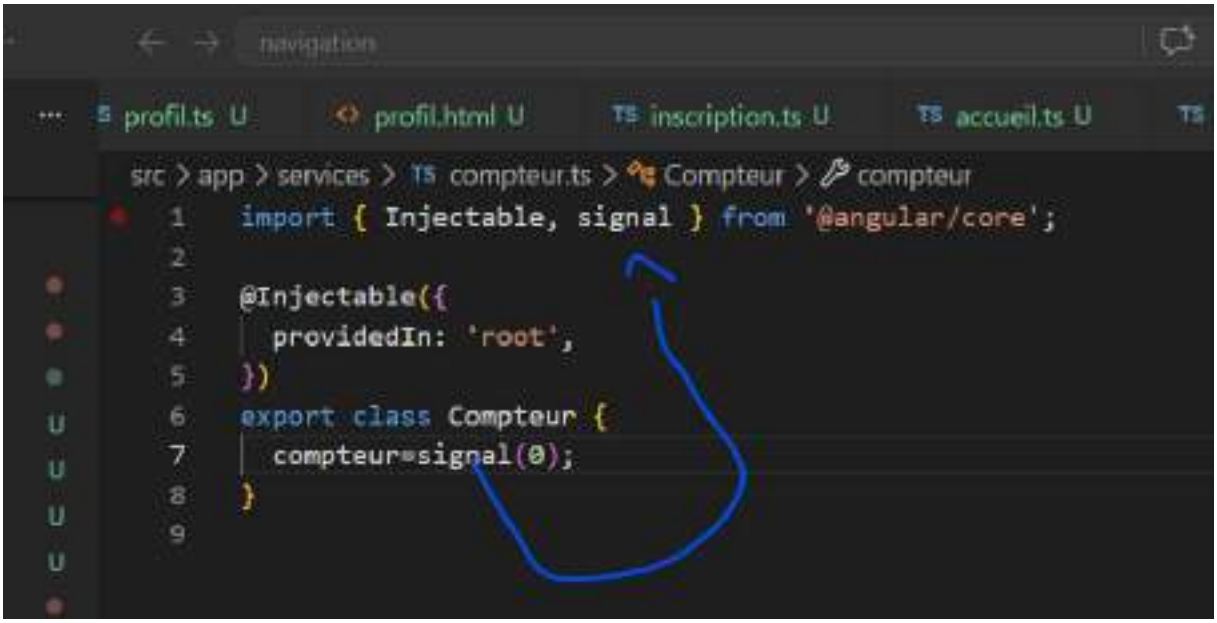
Accueil utilise compteur

Seuls ces écrans sont mis à jour.

Ex concret

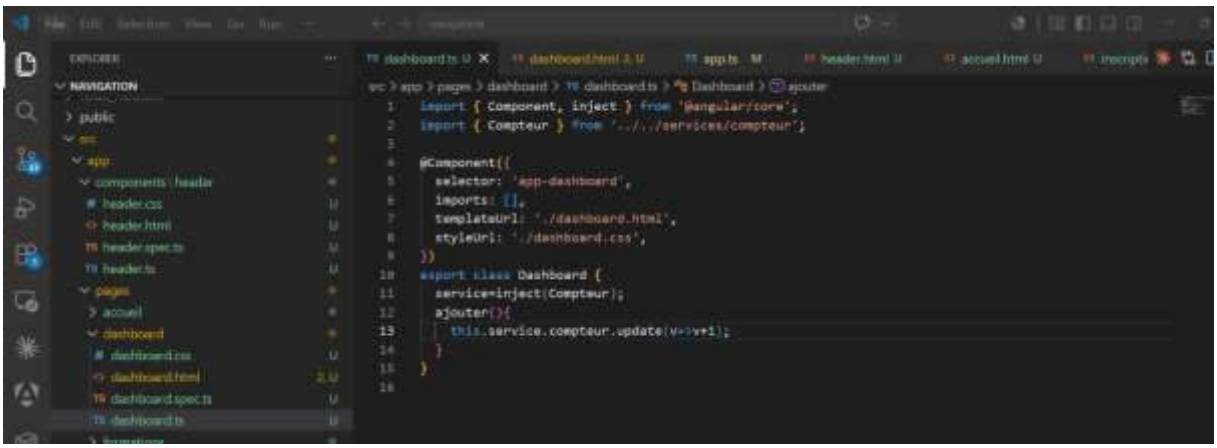
compteur.service.ts

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

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

  compteur = signal(0);

}
```



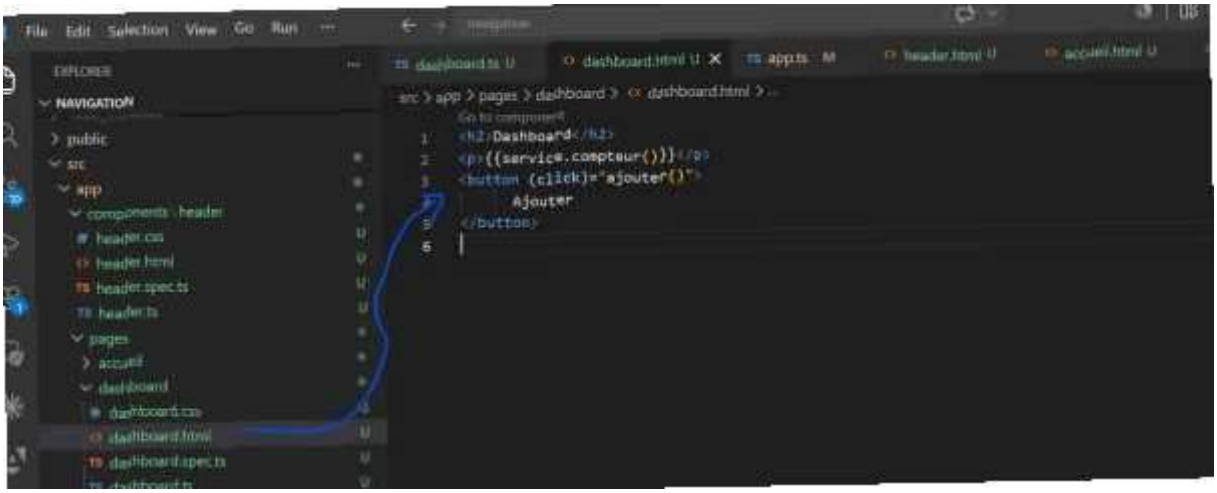
dashboard.ts

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

service = inject(CompteurService);

ajouter() {
 this.service.compteur.update(v => v + 1);
}
```



dashboard.html

```
<h2>Dashboard</h2>

<p>{{ service.compteur() }}</p>

<button (click)="ajouter()">
 Ajouter
</button>
```



profil.html

<h2>Profil</h2>

Nombre : {{ service.compteur() }}

accueil.html

<h2>Accueil</h2>

{{ service.compteur() }}

Ce qui se passe

Au départ :

Accueil : 0

Profil : 0

Dashboard : 0

Clic :

this.service.compteur.update(v => v + 1);

Résultat :

Accueil : 1

Profil : 1

Dashboard : 1

Tous les composants sont mis à jour automatiquement.

Pourquoi ?

Parce que :

compteur = signal(0);

Angular mémorise :

Accueil utilise compteur

Profil utilise compteur

Dashboard utilise compteur

Lorsque :

compteur.update(...)

Angular sait exactement :

"Je dois mettre à jour uniquement ces composants."

Pourquoi mettre le Signal dans le service ?

Car le service est partagé.

Service
↓
Signal
↙ ↓ ↘
Accueil
Profil
Dashboard

Tous les composants regardent la même donnée.

Ex CFITECH : Nombre d'inscriptions

Service :

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

  nombreInscriptions = signal(0);

}
```

Page Inscription :

```
valider() {
  this.service.nombreInscriptions.update(
    v => v + 1
  );
}
```

Dashboard :

```
<p>
Nombre d'inscriptions :
{{ service.nombreInscriptions() }}
</p>
```

Accueil :

```
<p>
Inscriptions :
{{ service.nombreInscriptions() }}
</p>
```

À chaque nouvelle inscription :

Accueil : 15

Dashboard : 15

mise à jour automatique.

Service	**Signal**
Partage les données	Rend les données réactives
Mémoire commune	Détecte les changements
Accessible partout	Met à jour automatiquement
Une seule instance	Réactivité précise

Un service permet de partager des données entre plusieurs composants de l'application. Lorsqu'une donnée du service est stockée dans un Signal, cette donnée devient réactive. Cela signifie qu'Angular sait exactement quels composants utilisent cette information et peut les mettre à jour automatiquement lorsqu'elle change. Le service joue donc le rôle de mémoire commune tandis que le Signal apporte la réactivité moderne d'Angular. Cette combinaison

constitue aujourd'hui l'une des approches recommandées dans Angular 21 pour partager des données entre plusieurs pages ou composants.

Exercice 1 : Création du premier service

Énoncé

Le centre CFITECH souhaite centraliser les informations du centre dans un service afin que plusieurs composants puissent les utiliser.

Créez un service nommé :

ng generate service services/centre

Ce service devra contenir :

- le nom du centre ;
- la ville ;
- le domaine de formation.

Affichez ces informations dans la page Accueil.

Solution

centre.service.ts

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

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

  nom = 'CFITECH';

  ville = 'Bruxelles';

  domaine = 'Développement Web';

}
```

accueil.ts

```
import { inject } from '@angular/core';
import { CentreService } from '../services/centre.service';

service = inject(CentreService);
```

accueil.html

```
<h2>{{ service.nom }}</h2>

<p>{{ service.ville }}</p>

<p>{{ service.domaine }}</p>
```

Exercice 2 : Partager une donnée entre deux composants

Énoncé

Le directeur de CFITECH souhaite afficher le nombre de stagiaires inscrits à la fois :

- sur la page Accueil ;
- sur la page Dashboard.

Créer un service contenant :

nombreStagiaires = 25;

Afficher cette valeur dans les deux composants.

Solution

stagiaire.service.ts

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

  nombreStagiaires = 25;

}
```

accueil.ts

```
service = inject(StagiaireService);
```

dashboard.ts

```
service = inject(StagiaireService);
```

HTML

```
<p>{{ service.nombreStagiaires }}</p>
```

Exercice 3 : Modifier une donnée du service

Énoncé

Ajouter un bouton :

Ajouter un stagiaire

À chaque clic, le nombre de stagiaires doit augmenter.

Solution

service

nombreStagiaires = 25;

composant

```
ajouter() {
  this.service.nombreStagiaires++;
}
```

HTML

```
<p>{{ service.nombreStagiaires }}</p>

<button (click)="ajouter()">
  Ajouter un stagiaire
</button>
```

Exercice 4 : Service d'inscription

Énoncé

Créer un service nommé :

InscriptionService

Ce service devra contenir :

- nom ;
- prénom ;
- email ;
- formation.

Créer une page Inscription permettant de remplir ces informations.

Créer une page Profil permettant d'afficher ces informations.

Solution

inscription.service.ts

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

  nom = '';

  prenom = '';

  email = '';

  formation = '';

}
```

inscription.ts

```
service = inject(InscriptionService);
```

inscription.html

```
<input [(ngModel)]="service.nom">

<input [(ngModel)]="service.prenom">

<input [(ngModel)]="service.email">
```

profil.html

```
<p>{{ service.nom }}</p>

<p>{{ service.prenom }}</p>

<p>{{ service.email }}</p>
```

Exercice 5 : Vérification du profil

Énoncé

Dans la page Profil, vérifier si toutes les informations ont été remplies.

Afficher :

Profil complet

ou

Profil incomplet

Solution

```
@if(
  service.nom &&
  service.prenom &&
  service.email &&
  service.formation
){

  <p>Profil complet</p>

} @else {

  <p>Profil incomplet</p>

}
```

Exercice 6 : Utilisation de Signal dans un service

Énoncé

Créer un service contenant :

compteur = signal(0);

Créer un bouton :

Nouvelle inscription

À chaque clic, le compteur doit augmenter.

Afficher le compteur dans deux composants différents.

Solution

compteur.service.ts

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

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

  compteur = signal(0);

}
```

composant

```
service = inject(CompteurService);

ajouter() {

  this.service.compteur.update(
    valeur => valeur + 1
  );

}
```

HTML

```
<p>{{ service.compteur() }}</p>

<button (click)="ajouter()">
  Ajouter
</button>
```

Exercice 7 : Dashboard CFITECH

Énoncé

Créer un service contenant :

- nom ;
- formation ;
- niveau.

Afficher ces informations dans :

- Profil ;
- Dashboard.

Modifier les données dans un composant.

Observer les changements dans les autres composants.

Solution

service

nom = 'Ali';

formation = 'Angular';

niveau = 'Débutant';

Tous les composants affichent :

{{ service.nom }}

{{ service.formation }}

{{ service.niveau }}

Exercice 8 : Signal et Dashboard

Énoncé

Créer :

nombreInscriptions = signal(0);

Créer un bouton :

Valider l'inscription

Chaque validation doit augmenter le compteur.

Afficher le résultat dans :

- Accueil ;
- Dashboard.

Solution

```
valider() {

 this.service.nombreInscriptions.update(
  valeur => valeur + 1
```

```
 );

}
```

HTML :

```
{{ service.nombreInscriptions() }}
```

Exercice 9 : Centre de formation

Énoncé

Créer un service :

FormationService

Contenant :

```
formations = [
 'Angular',
 'React',
 'Node.js'
];
```

Afficher cette liste dans deux composants différents.

Solution

```
formations = [
 'Angular',
 'React',
 'Node.js'
];
```

HTML :

```
@for(f of service.formations; track f){

 <p>{{ f }}</p>

}
```

Exercice 10 : Projet complet

Énoncé

Le centre CFITECH souhaite créer une application permettant :

- d'enregistrer un stagiaire ;
- d'afficher son profil ;
- d'afficher le nombre total d'inscriptions ;
- de consulter le tableau de bord.

Contraintes :

- utiliser un service ;
- utiliser inject() ;
- utiliser au moins un Signal ;
- partager les données entre plusieurs composants.

Solution attendue

Service

```
nom = '';

prenom = '';

formation = '';

compteur = signal(0);
```

Inscription

```
this.service.compteur.update(
  v => v + 1
);
```

Profil

```
{{ service.nom }}
```

Dashboard

```
{{ service.compteur() }}
```

Exercice : Comparaison Variable / Signal

Énoncé

Créer :

nombre1 = 0;

nombre2 = signal(0);

Créer deux boutons permettant d'incrémenter les valeurs.

Observer les différences dans le code.

Solution

nombre1++;

nombre2.update(v => v + 1);

HTML :

{{ nombre1 }}

{{ nombre2() }}

Recap :

Concept	**Rôle**
Service	Partager les données
Injectable	Rendre le service injectable
providedIn: 'root'	Une seule instance
inject()	Récupérer le service
Signal	Rendre les données réactives
update()	Modifier un Signal
set()	Affecter une valeur
Service + Signal	Partage + réactivité

Les Services Angular dans le monde professionnel

« Un service ne sert pas seulement à stocker quelques variables. »

En réalité, dans une application professionnelle Angular, **les services représentent le cœur de l'application**.

Les composants affichent l'interface.

Les services gèrent :

- les données ;
- la logique métier ;
- la communication ;
- les états ;
- les calculs ;
- les appels API ;
- l'authentification ;
- les paramètres de l'application.

1. Service de partage de données

Rôle

Partager des informations entre plusieurs composants.

Ex :

Le stagiaire s'inscrit sur une page.

Le profil et le dashboard doivent afficher les informations.

StagiaireService

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

  nom = '';
  prenom = '';
  formation = '';

}
```

Utilisé dans :

- Inscription
- Profil
- Dashboard

Exercice :

Une clinique médicale souhaite développer une application Angular permettant de gérer le dossier d'un patient.

Lorsqu'un patient arrive à l'accueil, ses informations sont enregistrées. Le médecin et le secrétariat doivent ensuite consulter ces informations sans avoir à les ressaisir.

Pour cela, l'application utilisera un service afin de partager les données entre plusieurs composants.

Travail demandé

Créer une application contenant trois composants :

- AccueilPatient
- DossierPatient
- Dashboard

Créer un service :

ng generate service services/patient

Le service devra contenir :

- nom ;
- prénom ;
- âge ;
- maladie.

Le composant **AccueilPatient** permettra de saisir les informations.

Le composant **DossierPatient** affichera le dossier médical.

Le composant **Dashboard** affichera un résumé du patient.

Reponse :

Étape 1 : Création du service

patient.service.ts

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

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

  nom = '';

  prenom = '';

  age = 0;

  maladie = '';

}
```

Étape 2 : Formulaire d'accueil

accueil-patient.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { PatientService } from '../services/patient.service';

@Component({
  standalone: true,
  imports: [FormsModule],
  templateUrl: './accueil-patient.html'
})
export class AccueilPatient {

  service = inject(PatientService);

}
```

accueil-patient.html

```
<h2>Enregistrement du patient</h2>

<input
```

```
 type="text"
 placeholder="Nom"
 [(ngModel)]="service.nom">

<br><br>

<input
 type="text"
 placeholder="Prénom"
 [(ngModel)]="service.prenom">

<br><br>

<input
 type="number"
 placeholder="Âge"
 [(ngModel)]="service.age">

<br><br>

<input
 type="text"
 placeholder="Maladie"
 [(ngModel)]="service.maladie">
```

Étape 3 : Dossier du patient

dossier-patient.ts

```
service = inject(PatientService);
```

dossier-patient.html

```
<h2>Dossier médical</h2>

<p>Nom : {{ service.nom }}</p>

<p>Prénom : {{ service.prenom }}</p>

<p>Âge : {{ service.age }}</p>

<p>Maladie : {{ service.maladie }}</p>
```

Étape 4 : Dashboard

dashboard.html

```
<h2>Patient actuel</h2>

<p>{{ service.prenom }} {{ service.nom }}</p>

<p>{{ service.age }} ans</p>

<p>{{ service.maladie }}</p>
```

Évolution moderne avec Signals

Le service pourrait devenir :

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

  nom = signal('');

  prenom = signal('');

  age = signal(0);

  maladie = signal('');

}
```

Modification :

```
this.service.nom.set('Martin');
```

Affichage :

```
{{ service.nom() }}
```

Ainsi, si les données changent, tous les composants sont automatiquement mis à jour.

2. Service d'état (State Management)

Rôle

Conserver l'état de l'application ou gérer l'état de l'application.

Exemple : mode sombre

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

  sombre = signal(false);

}
```

Composant :

```
changer() {
  this.service.sombre.update(v => !v);
}
```

Exercice

Le centre de formation **CFITECH** souhaite moderniser sa plateforme pédagogique.

L'utilisateur doit pouvoir changer le thème de l'application :

- Mode clair
- Mode sombre

Lorsque le thème change :

- le Header change ;
- le Dashboard change ;
- la page Profil change ;
- la page Accueil change.

Le changement doit rester actif lorsque l'on navigue entre les pages.

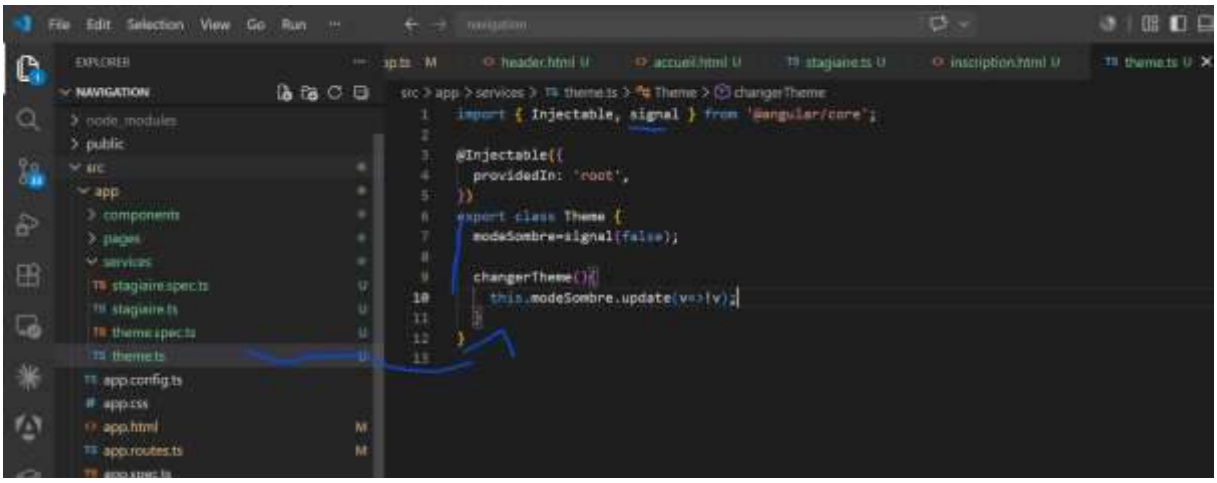
1. Création du service

ng generate service services/theme

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g s services/theme
CREATE src/app/services/theme.spec.ts (332 bytes)
CREATE src/app/services/theme.ts (118 bytes)
```

theme.service.ts

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

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

 modeSombre = signal(false);

 changerTheme() {
  this.modeSombre.update(v => !v);
 }

}
```

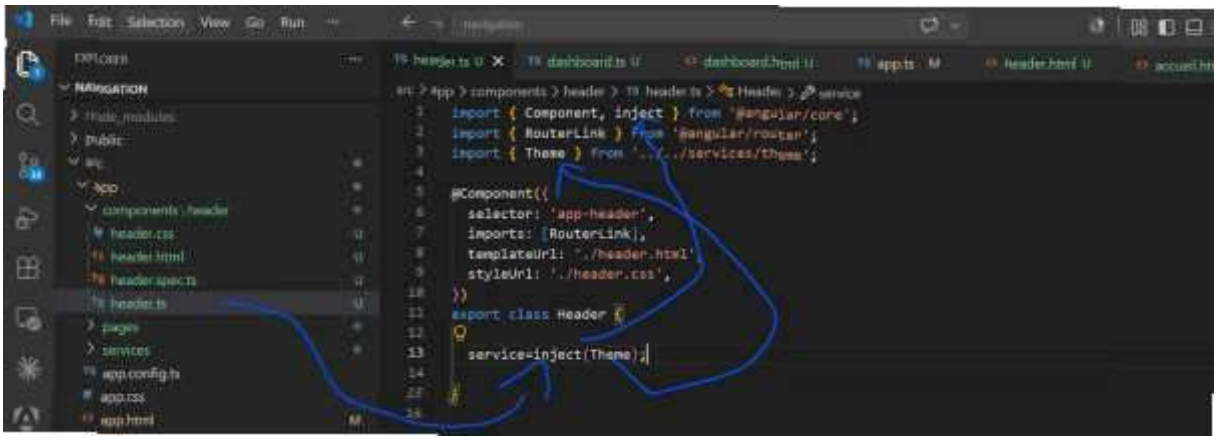


2. Header

header.ts

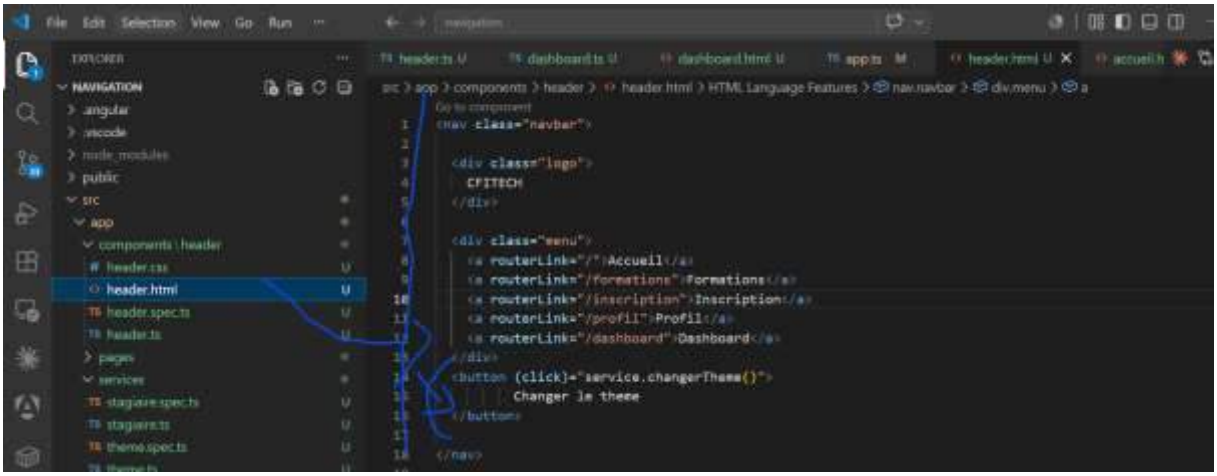
```
import { Component, inject } from '@angular/core';
import { RouterLink } from '@angular/router';
import { ThemeService } from '../services/theme.service';

@Component({
 selector: 'app-header',
 standalone: true,
 imports: [RouterLink],
```

```
 templateUrl: './header.html',
 styleUrl: './header.css'
})
export class HeaderComponent {

 service = inject(ThemeService);

}
```



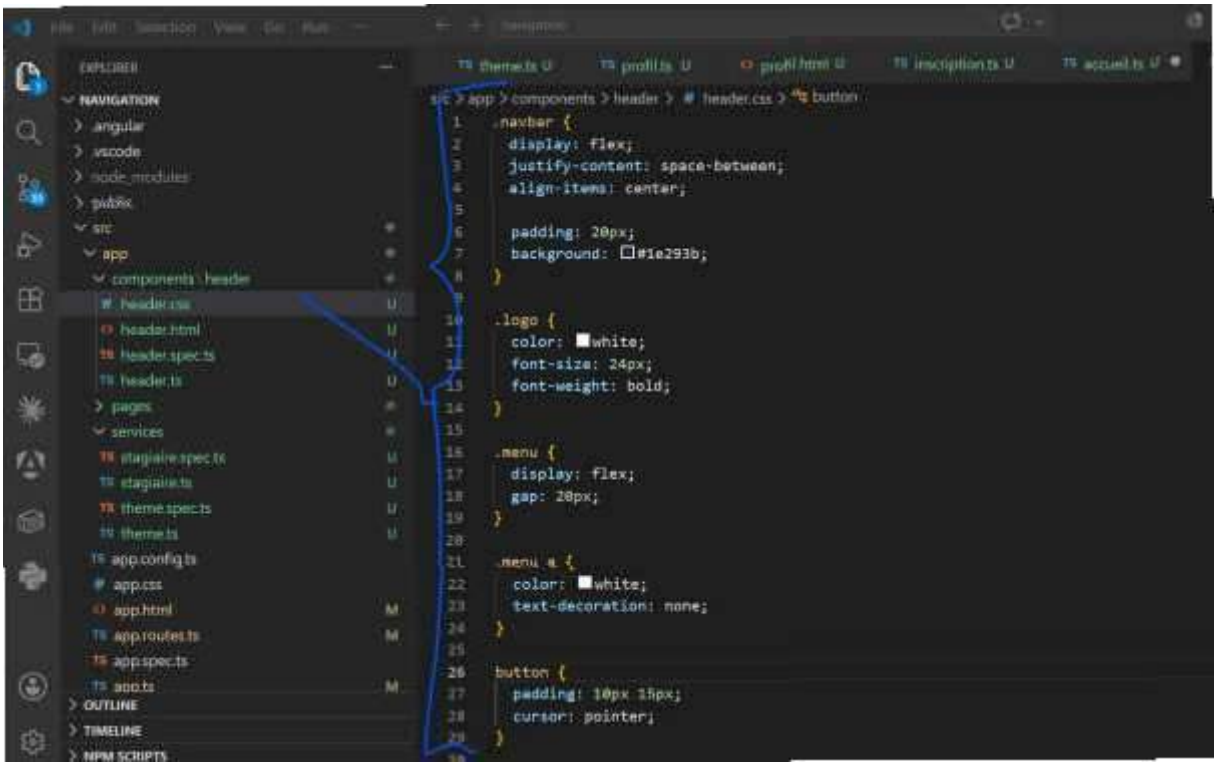
header.html

```
<nav class="navbar">

 <div class="logo">
  CFITECH
 </div>

 <div class="menu">
  <a routerLink="/">Accueil</a>
  <a routerLink="/profil">Profil</a>
  <a routerLink="/dashboard">Dashboard</a>
 </div>

 <button (click)="service.changerTheme()">
  Changer le thème
 </button>

</nav>
```



header.css

```
.navbar {
 display: flex;
 justify-content: space-between;
 align-items: center;

 padding: 20px;
 background: #1e293b;
}

.logo {
 color: white;
 font-size: 24px;
 font-weight: bold;
}

.menu {
 display: flex;
 gap: 20px;
}

.menu a {
 color: white;
 text-decoration: none;
}

button {
 padding: 10px 15px;
 cursor: pointer;
}
```

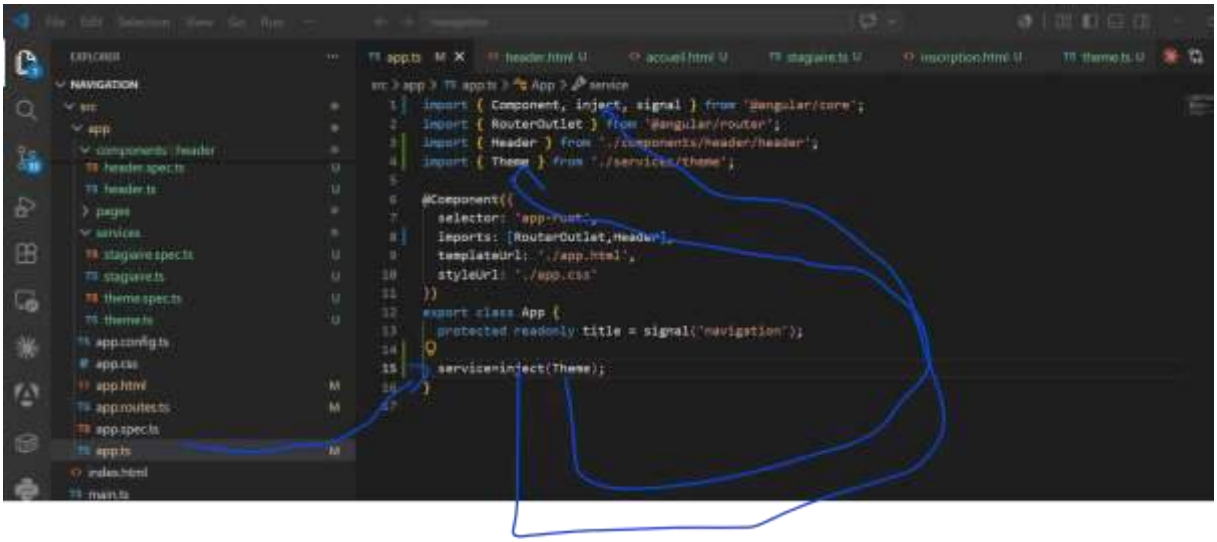


3. App Component

C'est lui qui va appliquer le thème à toute l'application.

app.ts

```
import { Component, inject } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ThemeService } from './services/theme.service';

@Component({
 selector: 'app-root',
 standalone: true,
 imports: [RouterOutlet],
 templateUrl: './app.html',
 styleUrl: './app.css'
})
export class AppComponent {

 service = inject(ThemeService);

}
```



app.html

```
<div
 [class.dark]="service.modeSombre()"
 [class.light]="!service.modeSombre()">

 <app-header></app-header>

 <router-outlet></router-outlet>

</div>
```

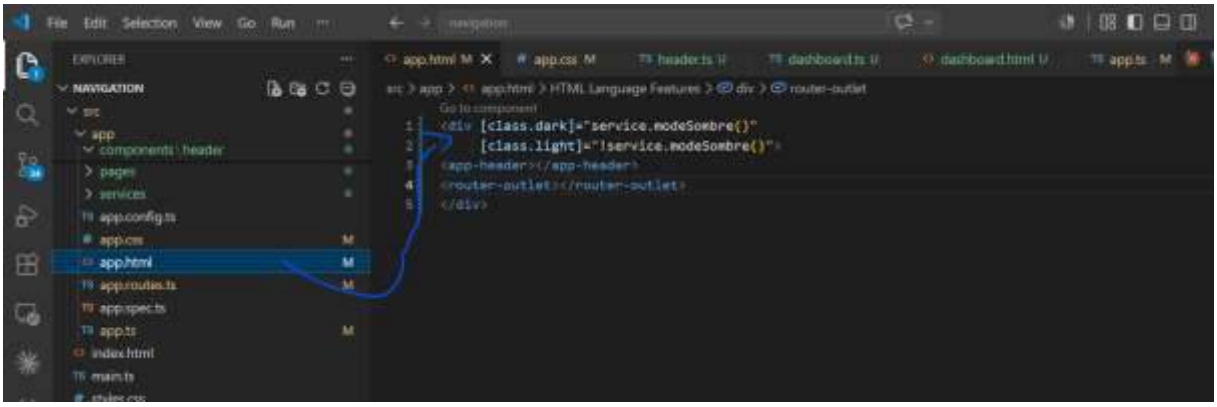
Qu'est-ce que [class.dark] ?

C'est ce qu'on appelle un **Class Binding**.

Il permet d'ajouter ou d'enlever une classe CSS selon une condition.

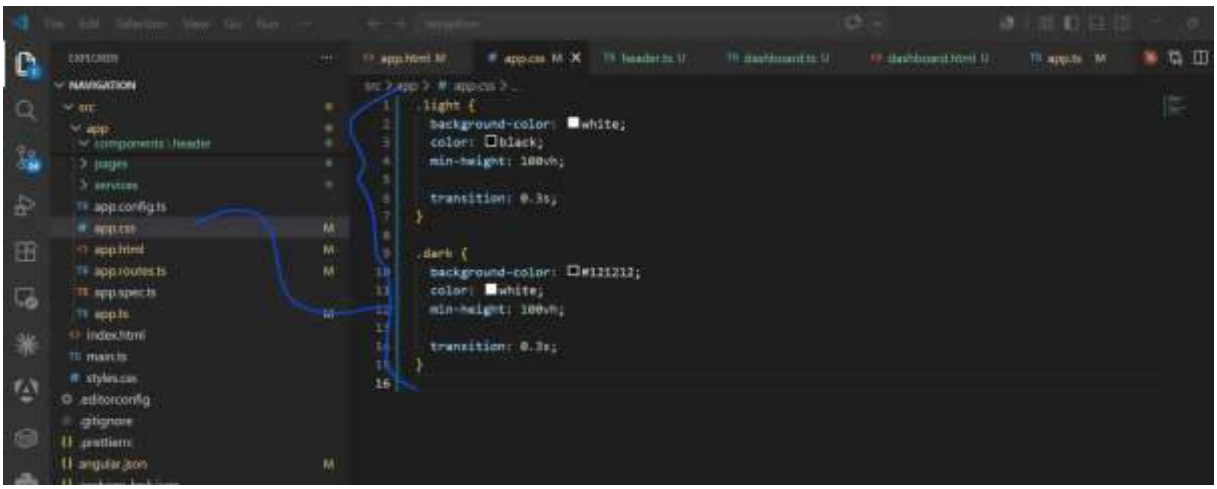
La syntaxe est :

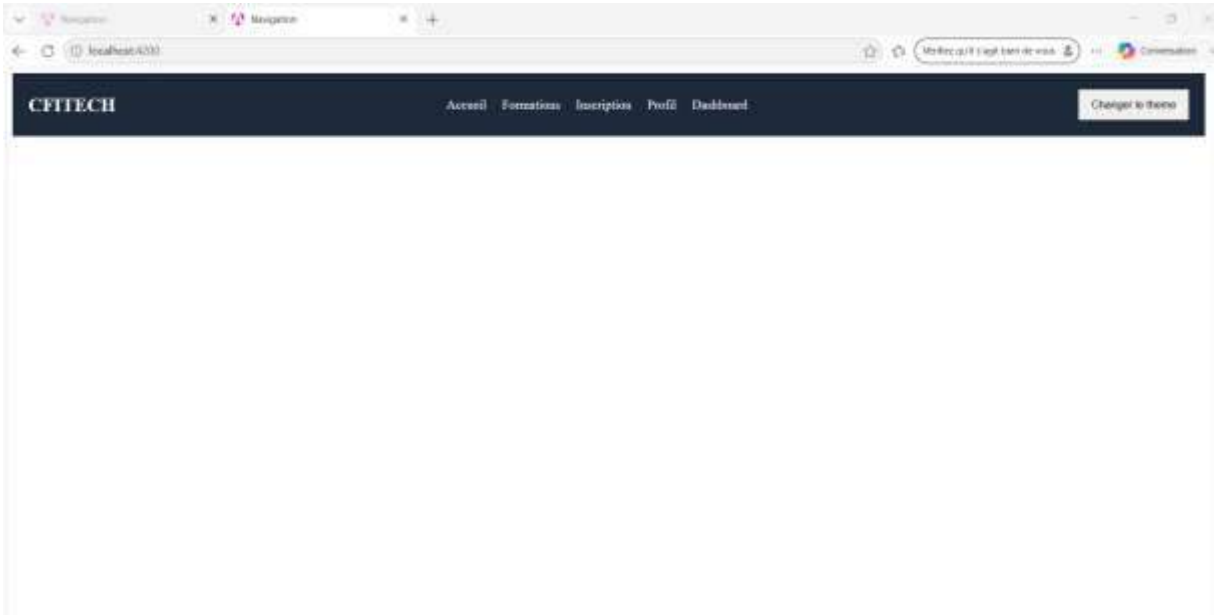
[class.nomClasse]="condition"

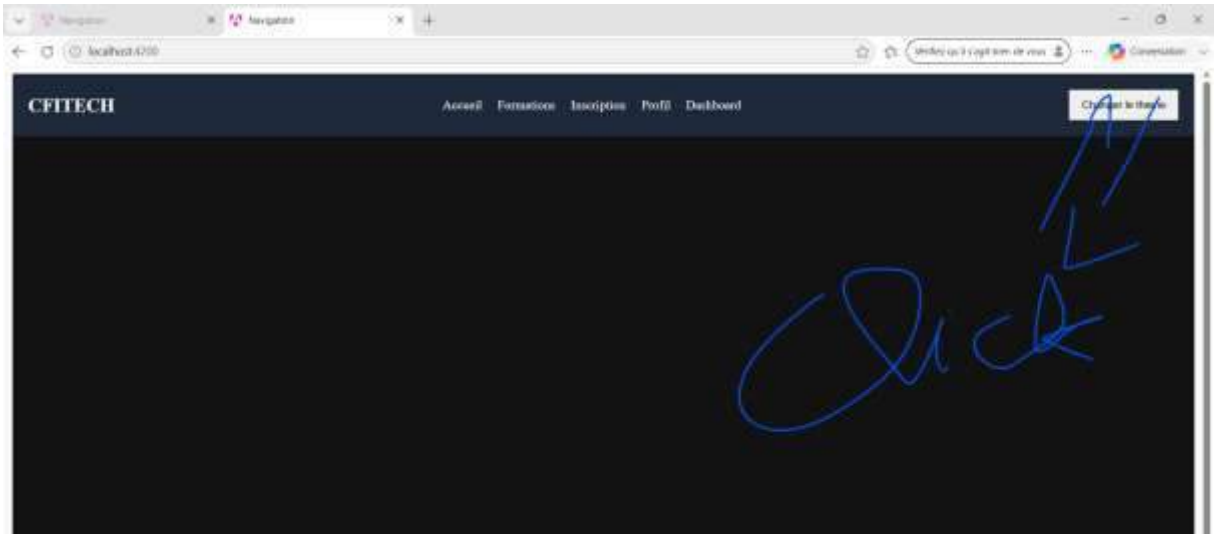


4. app.css

```
.light {
 background-color: white;
 color: black;
 min-height: 100vh;

 transition: 0.3s;
}

.dark {
 background-color: #121212;
 color: white;
 min-height: 100vh;

 transition: 0.3s;
}
```







3. Service de compteur

Exemple

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

  compteur = signal(0);

}
```

Utilisé pour :

- nombre de visiteurs
- nombre d'articles
- nombre d'inscriptions
- nombre de messages

4. Service de liste

Exemple : formations CFITECH

```
formations = [
  'Angular',
  'React',
  'Node'
];
```

Utilisé pour :

- produits
- employés
- étudiants
- commandes

Exercice :

Le centre de formation **CFITECH Bruxelles** souhaite développer une application permettant de gérer les formations proposées.

Les formations disponibles sont :

- Angular
- React
- Node.js

L'application doit permettre :

- d'afficher les formations ;
- d'ajouter une nouvelle formation ;
- de voir le nombre total de formations ;
- d'afficher la liste dans plusieurs composants.

Défi

Créer :

- Header
- ListeFormations
- Dashboard

Créer un service :

ng generate service services/formation

Le service devra stocker la liste des formations.

Le composant **ListeFormations** permettra d'ajouter une nouvelle formation.

Le Dashboard affichera le nombre total de formations.

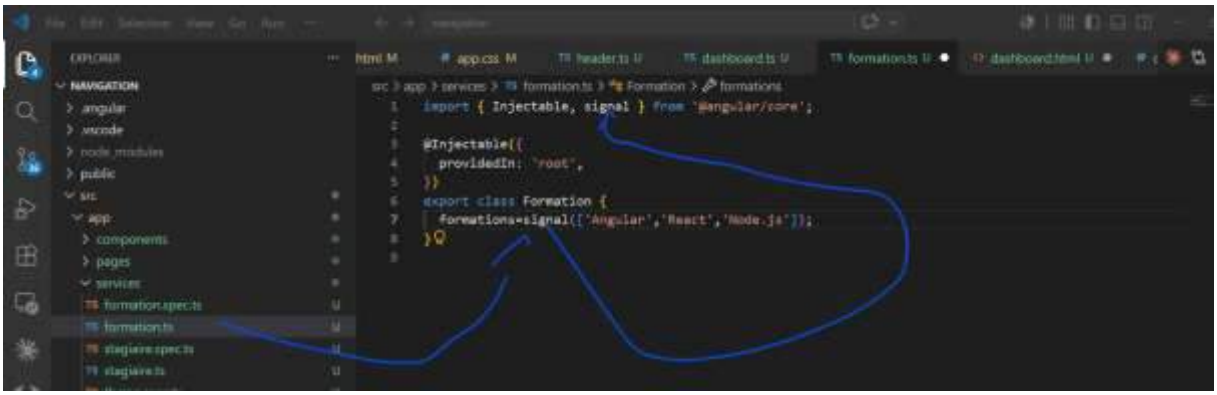
Réponse :

Étape 1 : Service

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g s services/formation
CREATE src/app/services/formation.spec.ts (352 bytes)
CREATE src/app/services/formation.ts (122 bytes)
```

formation.service.ts

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

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

  formations = signal([
    'Angular',
    'React',
    'Node.js'
  ]);

}
```



Étape 2 : Ajouter une méthode

```
ajouter(formation: string) {

//Le spread operator
  this.formations.update(
    liste => [...liste, formation]
  );

}
```

Le spread operator

[...liste, formation]

Le symbole :

...

s'appelle le **spread operator**.

Il signifie :

"Prends tous les éléments du tableau."

Donc :

...liste

devient :

Angular
React
Node

On ajoute ensuite :

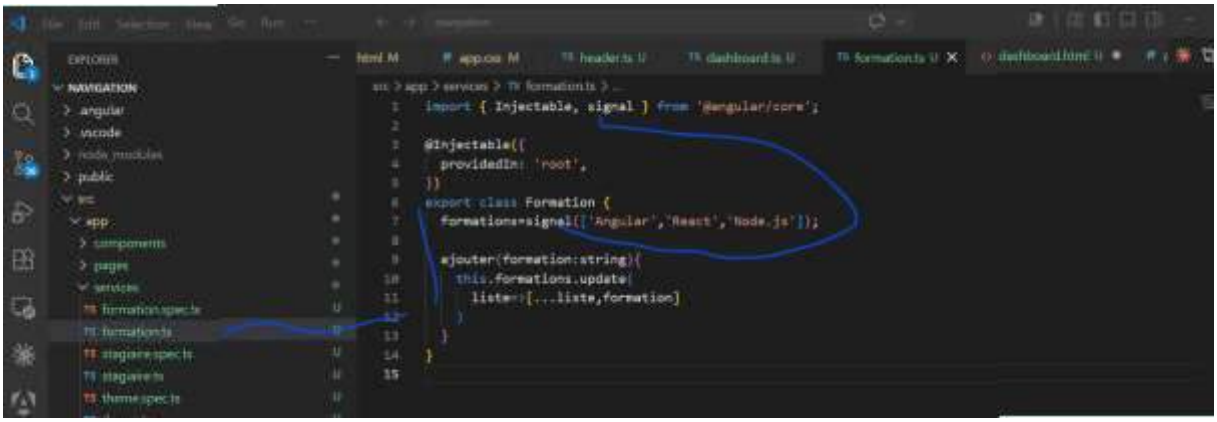
formation

qui vaut :

Java

Angular construit donc :

```
[
  'Angular',
  'React',
  'Node',
  'Java'
]
```



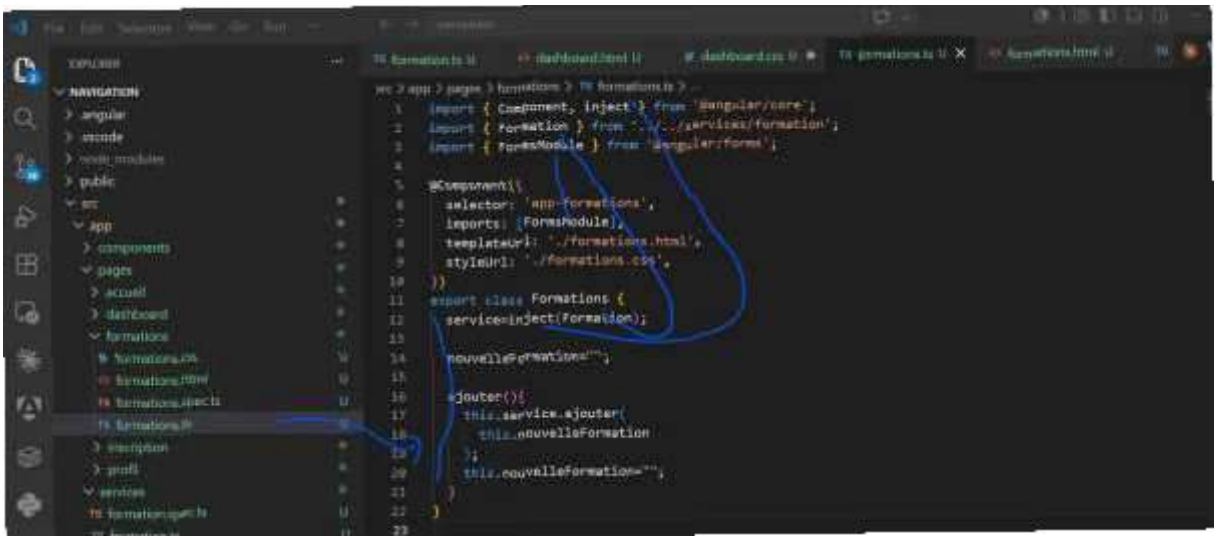
Service complet

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

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

 formations = signal([
  'Angular',
  'React',
  'Node.js'
 ]);

 ajouter(formation: string) {

  this.formations.update(
   liste => [...liste, formation]
  );

 }

}
```

Étape 3 : Composant ListeFormations

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { FormationService } from '../services/formation.service';

@Component({
 standalone: true,
 imports: [FormsModule],
 templateUrl: './liste-formations.html'
```

```
})
export class ListeFormations {

 service = inject(FormationService);

 nouvelleFormation = '';

 ajouter() {

  this.service.ajouter(
   this.nouvelleFormation
  );

  this.nouvelleFormation = '';

 }

}
```

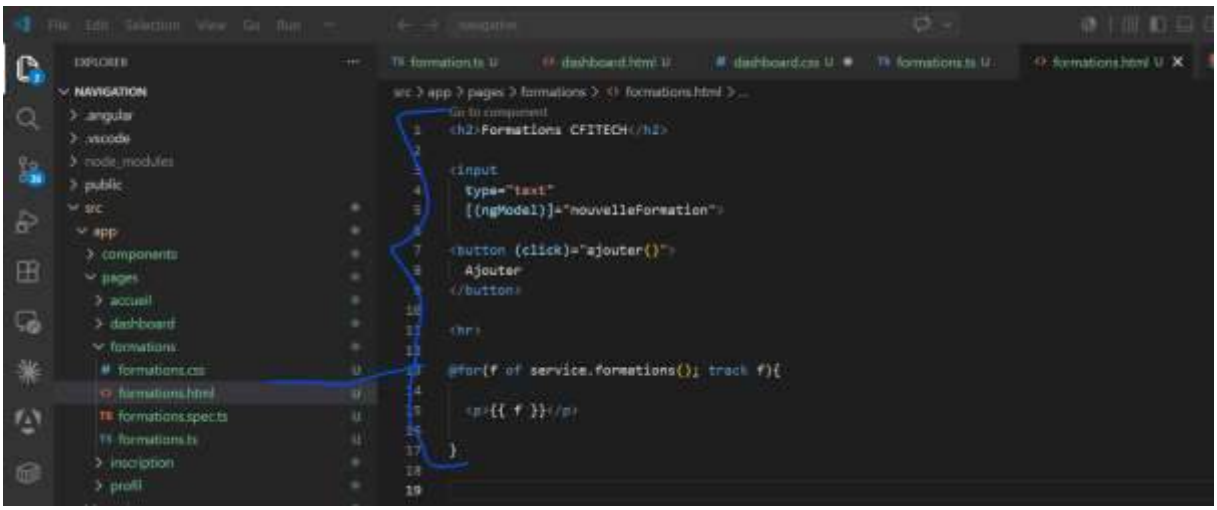


liste-formations.html

```
<h2>Formations CFITECH</h2>

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

<button (click)="ajouter()">
 Ajouter
</button>

<hr>

@for(f of service.formations(); track f){
```

{{ f }}

}

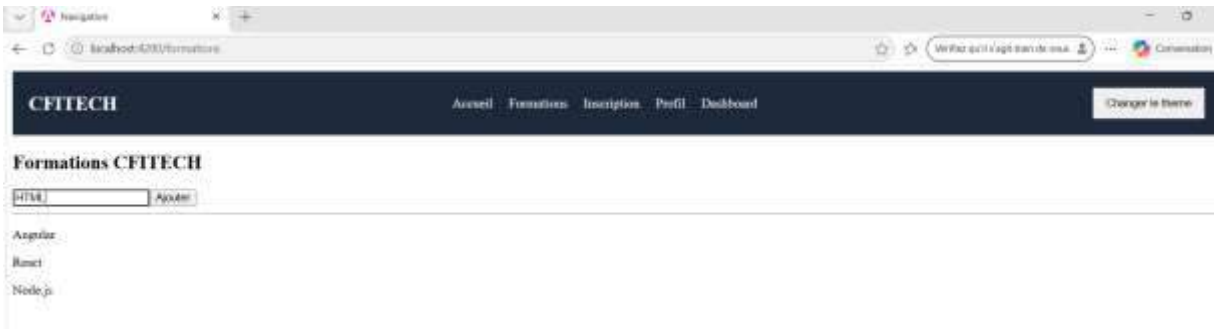


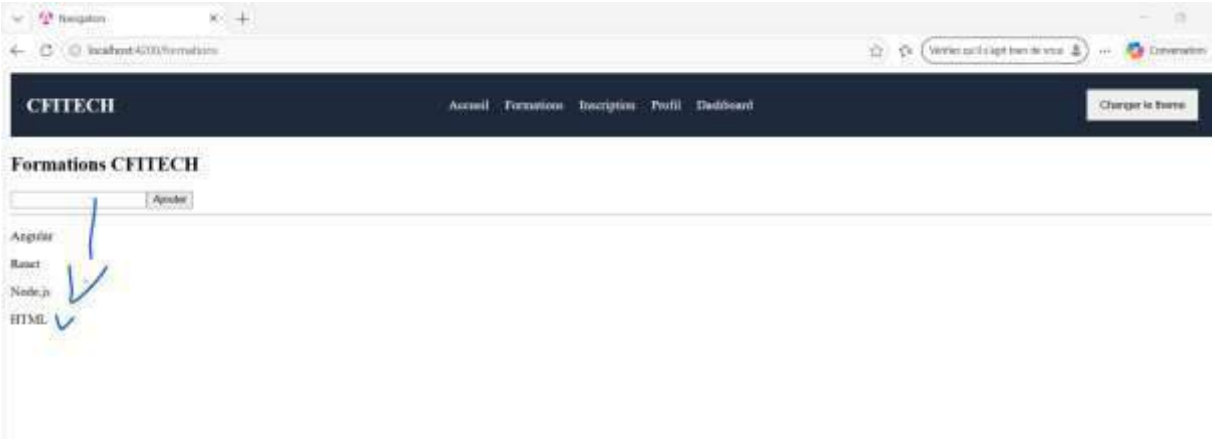
Dashboard

service = inject(FormationService);

<h2>Dashboard</h2>

Nombre de formations :
{{ service.formations().length }}





Pourquoi avons-nous utilisé :

```
this.formations.update(
  liste => [...liste, formation]
);
```

et non :

```
this.formations().push(formation);
```

Réponse

push()

modifie directement le tableau.

Le Signal ne détecte pas toujours cette modification.

update()

crée un nouveau tableau.

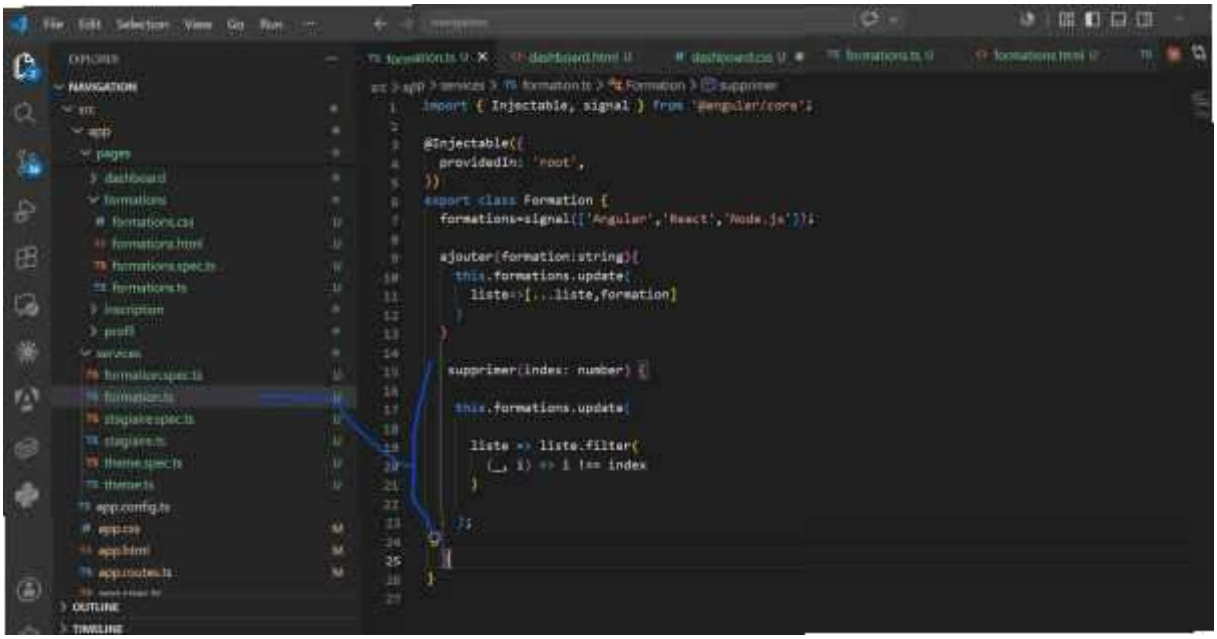
Angular détecte immédiatement le changement.

Suppression:

formation.service.ts

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

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

  formations = signal([
    'Angular',
    'React',
    'Node.js'
  ]);

  ajouter(formation: string) {

    this.formations.update(
      liste => [...liste, formation]
    );

  }


//Conserver tous les éléments dont l'index est différent de celui à supprimer.


supprimer(index: number) {

    this.formations.update(

      liste => liste.filter(
        (_, i) => i !== index
      )

    );

  }

}
```



(_, i) => i !== index

Ici :

- _ = l'élément du tableau
- i = sa position (index)

Angular parcourt :

0 → Angular
1 → React
2 → Node.js

Premier passage

_ = Angular
i = 0

On calcule :

0 !== 1

Résultat :

true

Angular garde Angular.

Deuxième passage

_ = React
i = 1

On calcule :

1 !== 1

Résultat :

false

React est supprimé.

Troisième passage

_ = Node.js
i = 2

On calcule :

2 !== 1

Résultat :

true

Node.js reste.

Rés

Avant :

```
[
  'Angular',
  'React',
  'Node.js'
]
```

Après :

```
[
  'Angular',
  'Node.js'
]
```

Pourquoi le _ ?

Normalement on pourrait écrire :

formation, i

comme ceci :

```
liste.filter(
  (formation, i) => i !== index
)
```

Mais ici on n'utilise pas :

formation

On utilise seulement :

i

Par convention, on écrit :

_

pour dire :

"Je reçois cette variable, mais je ne l'utilise pas."

Version très détaillée

```
liste.filter(
  function(formation, i) {
    return i !== index;
  }
)
```

Supposons :

index = 1;

Angular fait :

return 0 !== 1; // true
return 1 !== 1; // false
return 2 !== 1; // true

Pourquoi utiliser filter ?

Parce que filter() crée un nouveau tableau.

Avant :

Angular
React
Node.js

Après :

Angular
Node.js

Le Signal reçoit donc une nouvelle valeur.

Angular détecte immédiatement le changement.

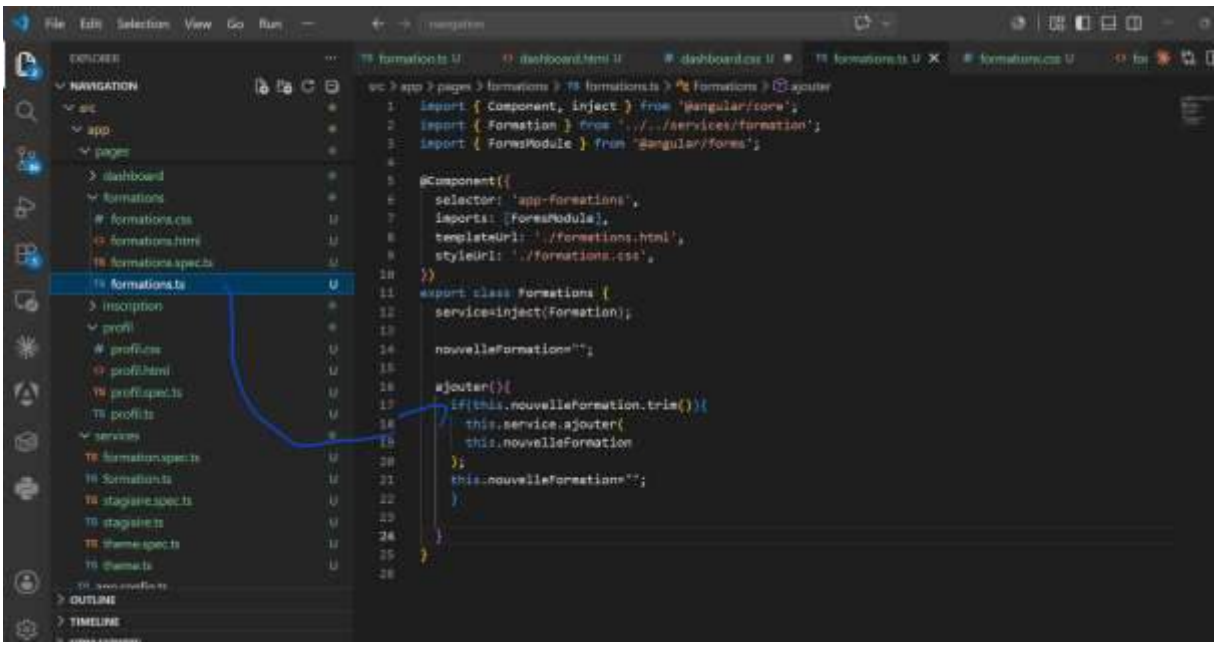
La méthode filter() parcourt tous les éléments du tableau et décide lesquels doivent être conservés. La variable i représente la position de chaque élément dans la liste. L'expression i !== index signifie : "conserver tous les éléments dont la position est différente de celle que l'on souhaite supprimer". Un nouveau tableau est alors créé, ce qui permet au Signal de détecter immédiatement le changement et de mettre automatiquement à jour l'interface.

2. Composant Formations

formations.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { FormationService } from '../services/formation.service';
```

```
@Component({
 standalone: true,
 imports: [FormsModule],
 templateUrl: './formations.html',
 styleUrl: './formations.css'
})
export class FormationsComponent {

 service = inject(FormationService);

 nouvelleFormation = '';

 ajouter() {

  if(this.nouvelleFormation.trim()) {

   this.service.ajouter(
    this.nouvelleFormation
   );

   this.nouvelleFormation = '';

  }

 }

}
```

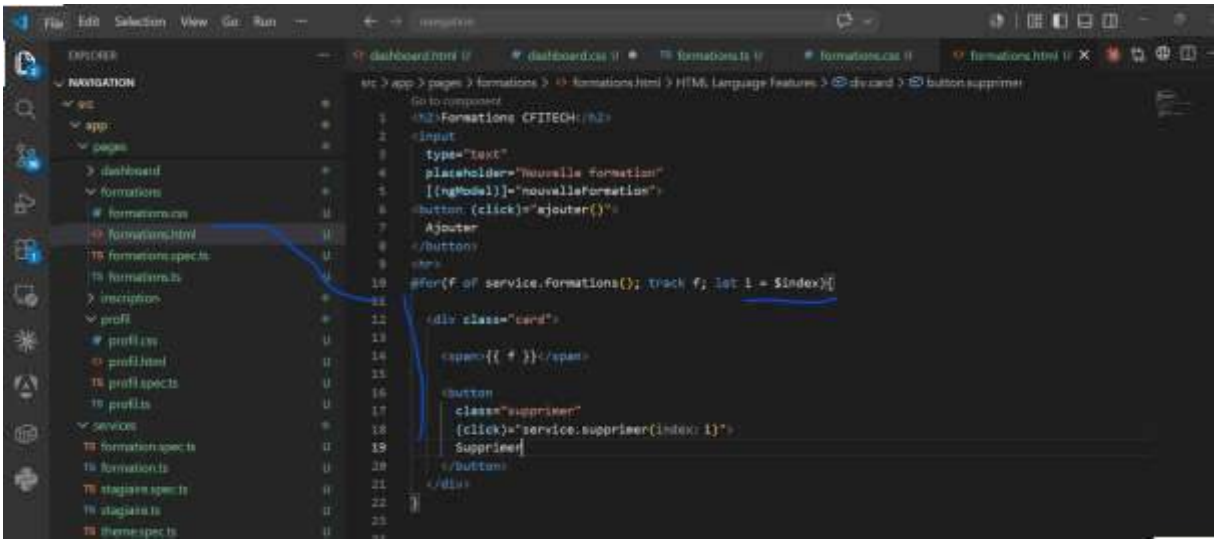


La méthode trim() supprime les espaces situés au début et à la fin d'une chaîne de caractères. Elle est utilisée ici pour éviter qu'un utilisateur ajoute une valeur vide ou composée uniquement d'espaces. Le if vérifie qu'il reste réellement du texte après cette suppression des espaces. Si la condition est vraie, la formation est ajoutée puis le champ est réinitialisé. Il s'agit d'une bonne

pratique très utilisée dans les applications professionnelles afin de garantir la qualité des données saisies par l'utilisateur.

formations.html

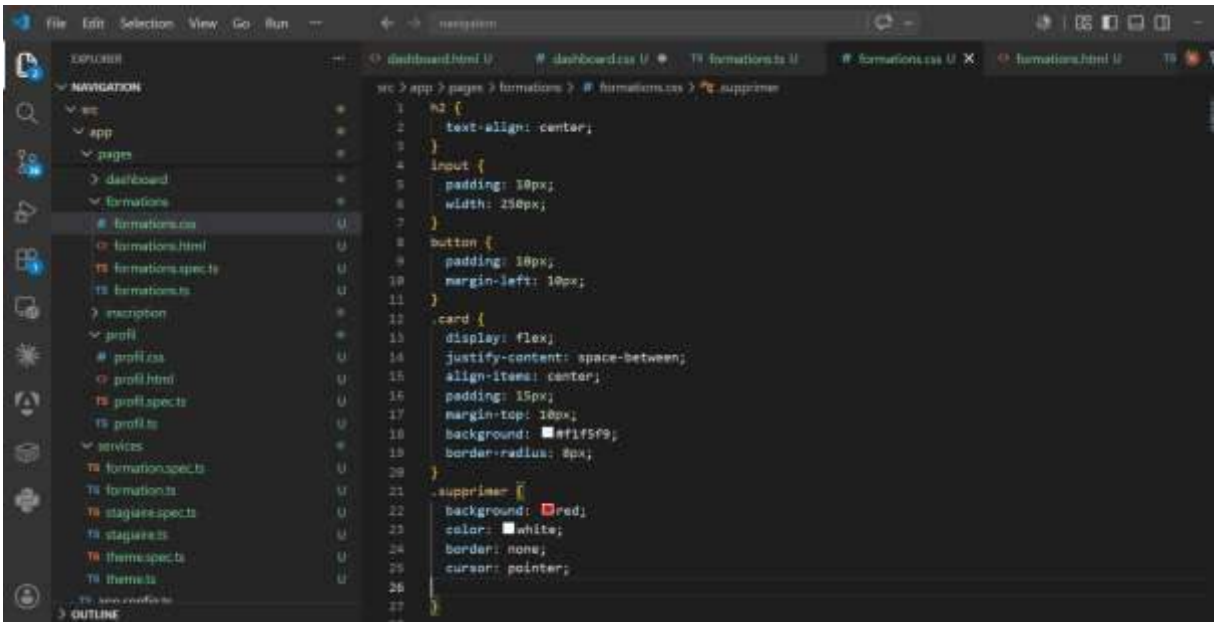
```
<h2>Formations CFITECH</h2>

<input
 type="text"
 placeholder="Nouvelle formation"
 [(ngModel)]="nouvelleFormation">

<button (click)="ajouter()">
 Ajouter
</button>

<hr>

@for(f of service.formations(); track f; let i = $index){

 <div class="card">

  <span>{{ f }}</span>

  <button
   class="supprimer"
   (click)="service.supprimer(i)">

   Supprimer

  </button>

 </div>

}
```

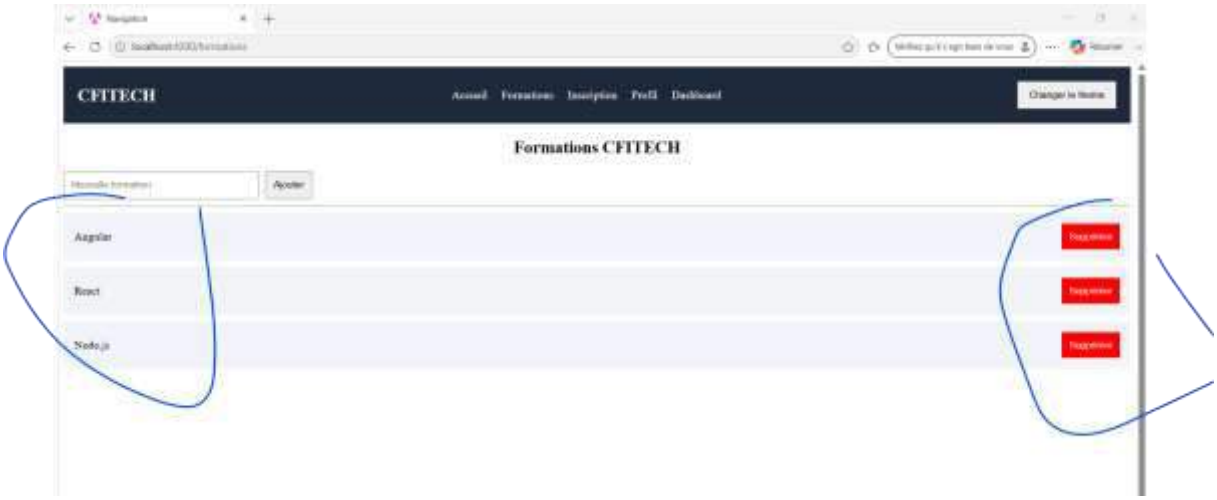


formations.css

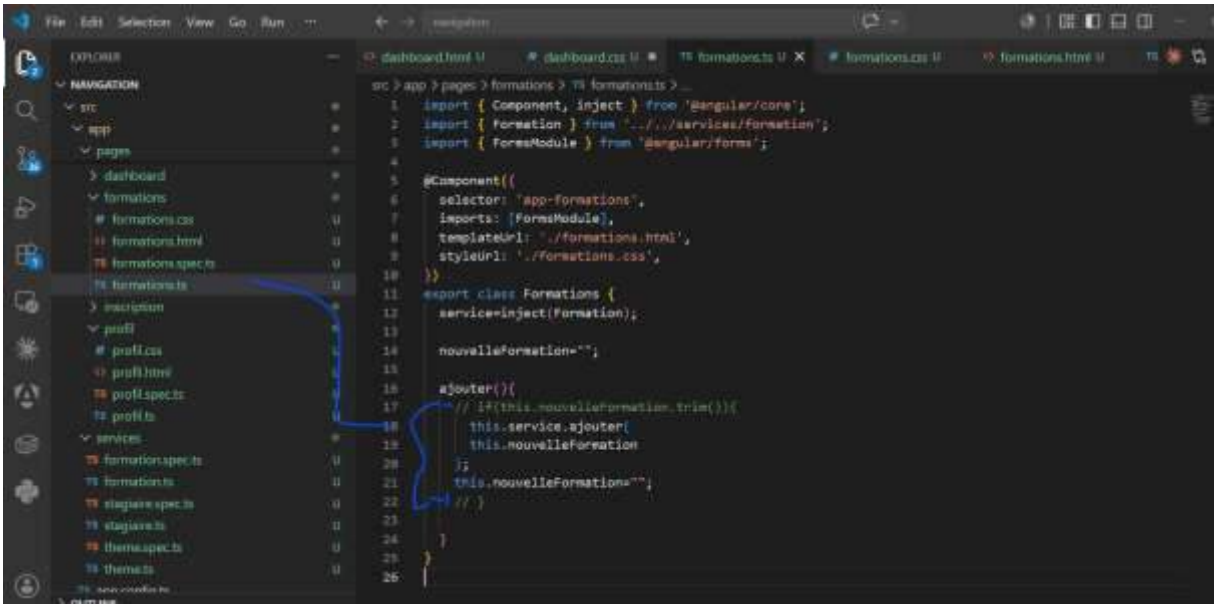
```
h2 {
 text-align: center;
}

input {
 padding: 10px;
 width: 250px;
}

button {
 padding: 10px;
 margin-left: 10px;
}

.card {

 display: flex;

 justify-content: space-between;

 align-items: center;

 padding: 15px;

 margin-top: 10px;

 background: #f1f5f9;

 border-radius: 8px;
```

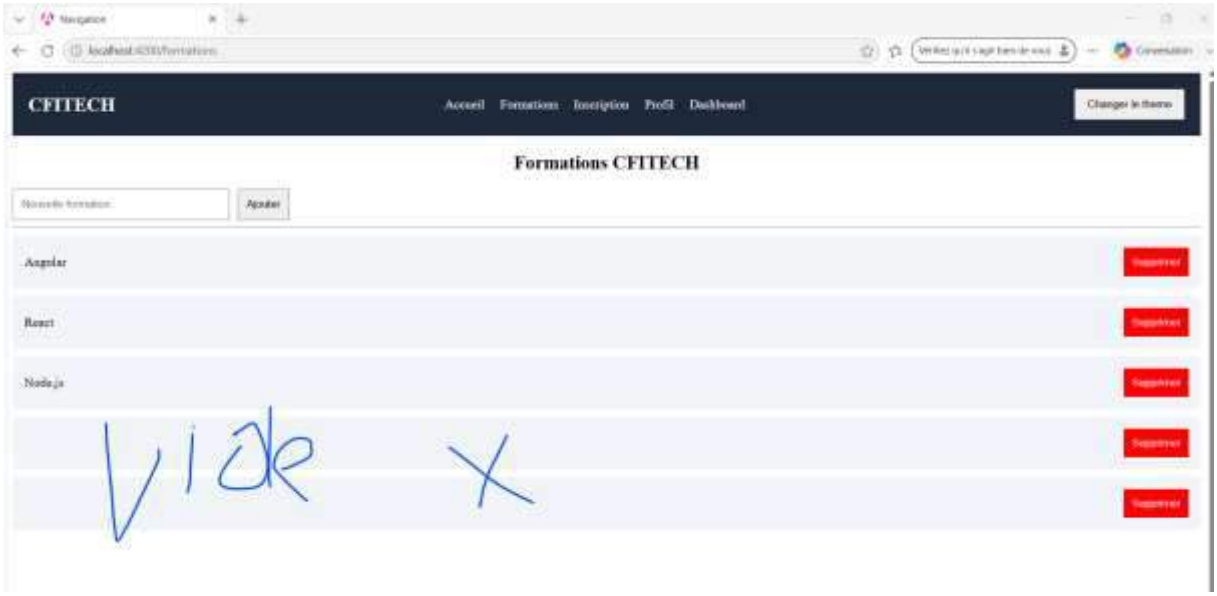
```
}

.supprimer {

 background: red;

 color: white;

 border: none;

 cursor: pointer;

}
```





Si l'on supprime la condition if ainsi que la méthode trim(), il devient possible d'ajouter des valeurs vides ou contenant uniquement des espaces. Cela peut entraîner des données incorrectes dans l'application, ce qui n'est pas recommandé dans le développement professionnel.



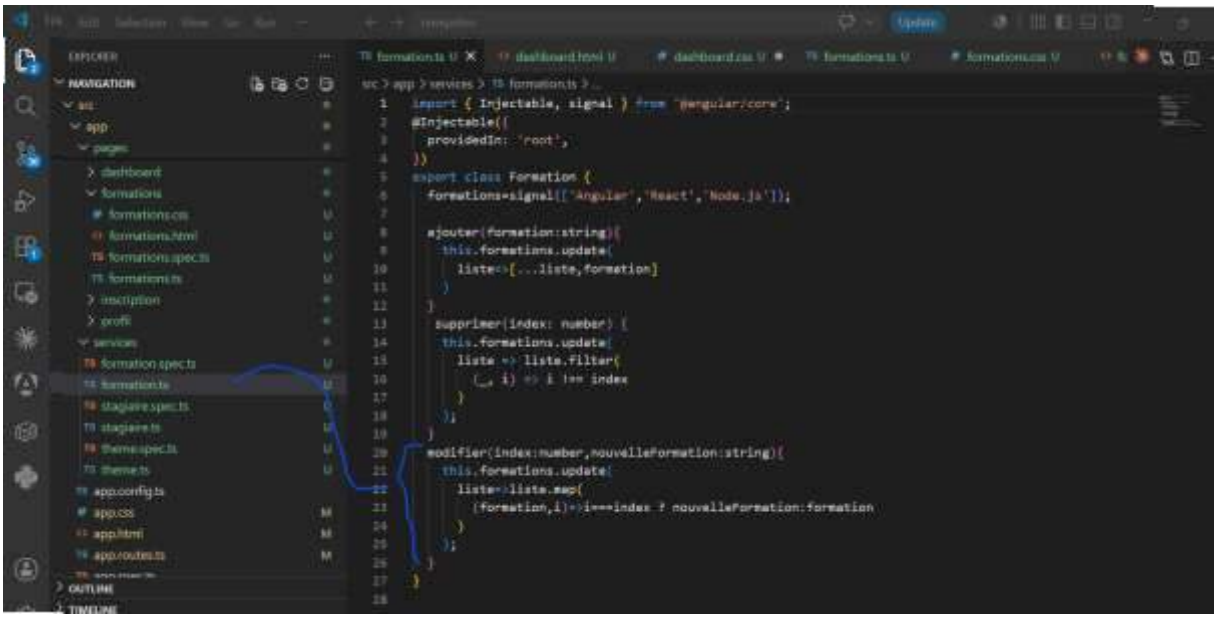


Modifier :

1. Service

Ajoutons une méthode :

```
modifier(index: number, nouvelleFormation: string) {
 this.formations.update(
  liste => liste.map(
   (formation, i) =>i === index ? nouvelleFormation : formation
  )
 );
}
```



La méthode map() parcourt tous les éléments du tableau un par un. Pour chaque élément, Angular vérifie si sa position correspond à celle de l'élément que l'on souhaite modifier. Cette vérification est réalisée grâce à l'opérateur ternaire :

```
i === index
 ? nouvelleFormation
 : formation
```

Le fonctionnement du ternaire est le suivant :

- La partie avant le ? représente la condition.

- La partie située entre ? et : est la valeur retournée si la condition est vraie.
- La partie située après : est la valeur retournée si la condition est fausse.

Dans cet exemple, si la position courante i est égale à l'index recherché, Angular remplace l'ancienne valeur par nouvelleFormation. Dans le cas contraire, l'élément d'origine est conservé.

Cette écriture est équivalente à :

```
if (i === index) {
  return nouvelleFormation;
} else {
  return formation;
}
```

À la fin du parcours, map() crée un nouveau tableau contenant les valeurs conservées et la valeur modifiée. Ce nouveau tableau est ensuite transmis au Signal grâce à la méthode update(). Angular détecte immédiatement cette nouvelle valeur et met automatiquement l'interface à jour. Cette approche respecte le principe d'immutabilité, très utilisé dans le développement moderne, qui consiste à créer une nouvelle version des données plutôt que de modifier directement les anciennes.

2. Composant

```
service = inject(FormationService);

nouvelleFormation = '';

indexModification: number | null = null;
```

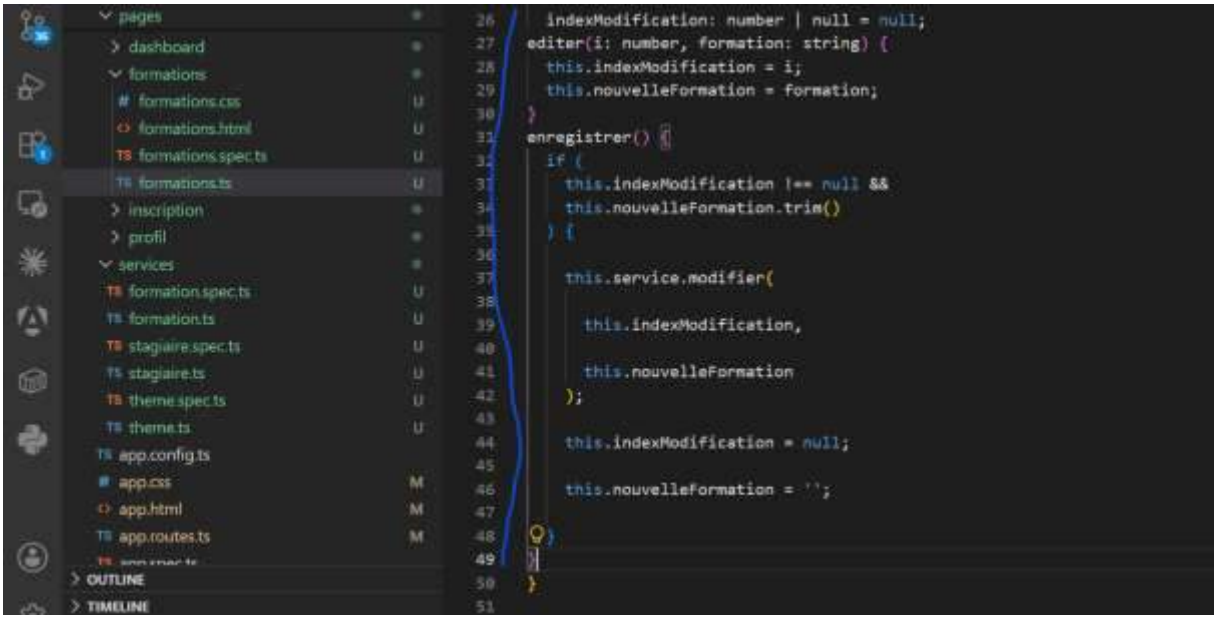
Méthode éditer

```
editer(i: number, formation: string) {

  this.indexModification = i;

  this.nouvelleFormation = formation;

}
```

Méthode enregistrer

```
enregistrer() {

  if (
    this.indexModification !== null &&
    this.nouvelleFormation.trim()
```

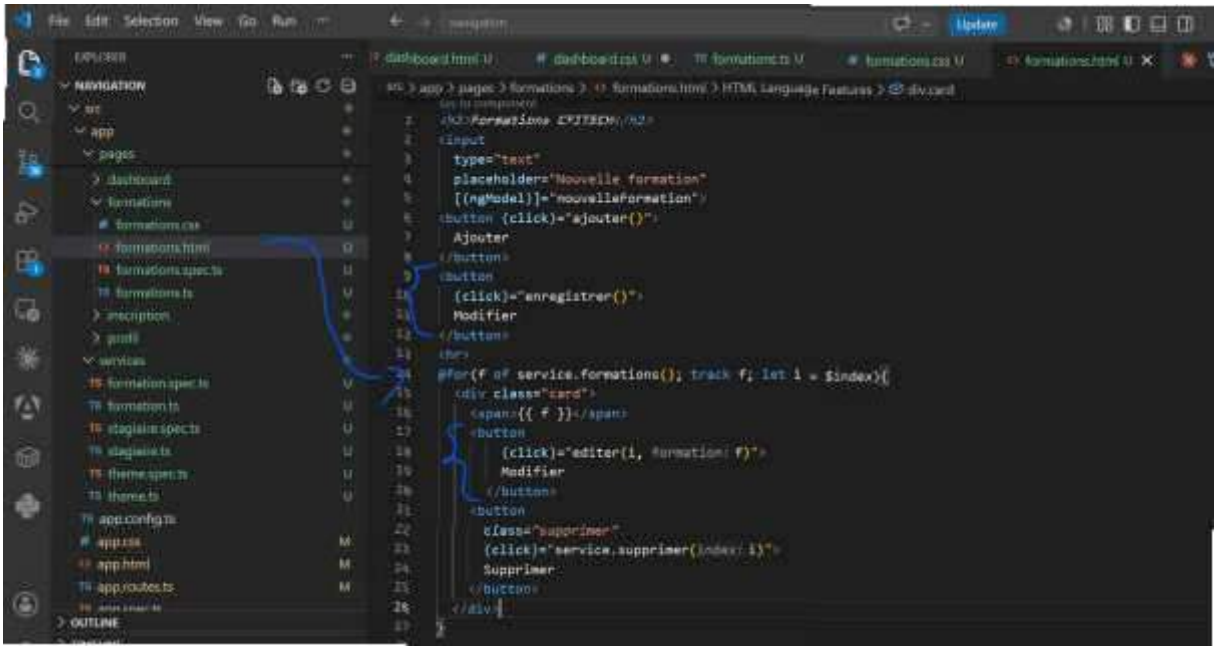
```
  ) {
    this.service.modifier(
      this.indexModification,
      this.nouvelleFormation
    );
    this.indexModification = null;
    this.nouvelleFormation = '';
  }
}
```



La déclaration indexModification: number | null = null signifie que la variable peut contenir soit un nombre, soit la valeur null. Le nombre représente la position de l'élément actuellement en cours de modification, tandis que null indique qu'aucune modification n'est en cours. Au démarrage du composant, la variable vaut null, car aucun élément n'est encore sélectionné. Lorsqu'un utilisateur clique sur le bouton **Modifier**, cette variable reçoit l'index de l'élément concerné. Une fois la modification enregistrée, elle est remise à null, ce qui signifie que la modification est terminée et qu'aucun élément n'est désormais sélectionné. Cette approche est très utilisée en TypeScript pour représenter un état « vide » ou « non sélectionné » de manière claire et sécurisée.

HTML

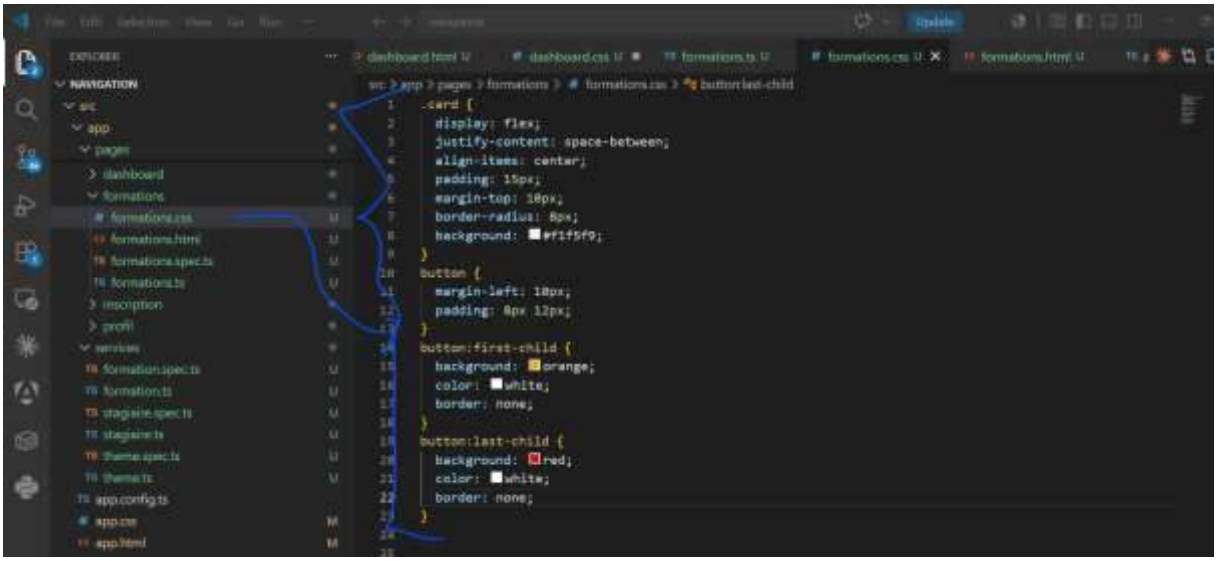
```
<input
 type="text"
 [(ngModel)]="nouvelleFormation">

<button (click)="ajouter()">
 Ajouter
</button>

<button
 (click)="enregistrer()">

 Modifier

</button>

<hr>

@for(f of service.formations(); track f; let i = $index){

 <div class="card">

  {{ f }}

  <div>

   <button
    (click)="editer(i, f)">

    Modifier

   </button>

   <button
    (click)="supprimer(i)">

    Supprimer

   </button>

  </div>

 </div>

}
```

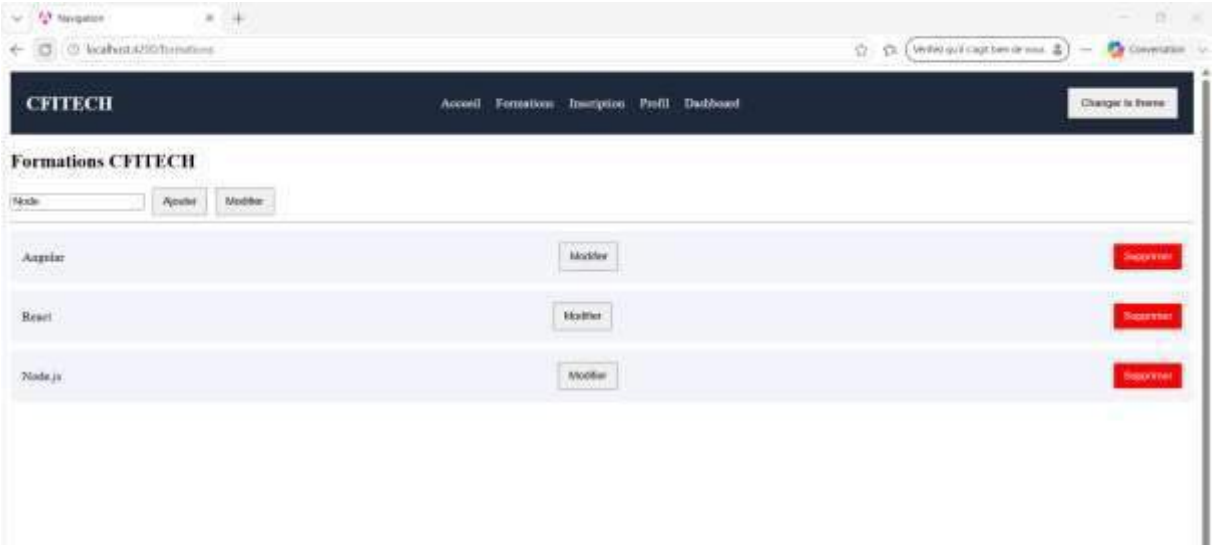


CSS

```
.card {
 display: flex;
 justify-content: space-between;
 align-items: center;
 padding: 15px;
 margin-top: 10px;
 border-radius: 8px;
 background: #f1f5f9;
}
button {
 margin-left: 10px;
 padding: 8px 12px;
```

```
}

button:first-child {
 background: orange;
 color: white;
 border: none;
}

button:last-child {
 background: red;
 color: white;
 border: none;
}
```





L'application réalisée met en œuvre les quatre opérations essentielles du développement d'applications, appelées **CRUD** : créer, afficher, modifier et supprimer des données. Ces opérations sont présentes dans la majorité des applications professionnelles et constituent une compétence fondamentale à maîtriser en Angular.

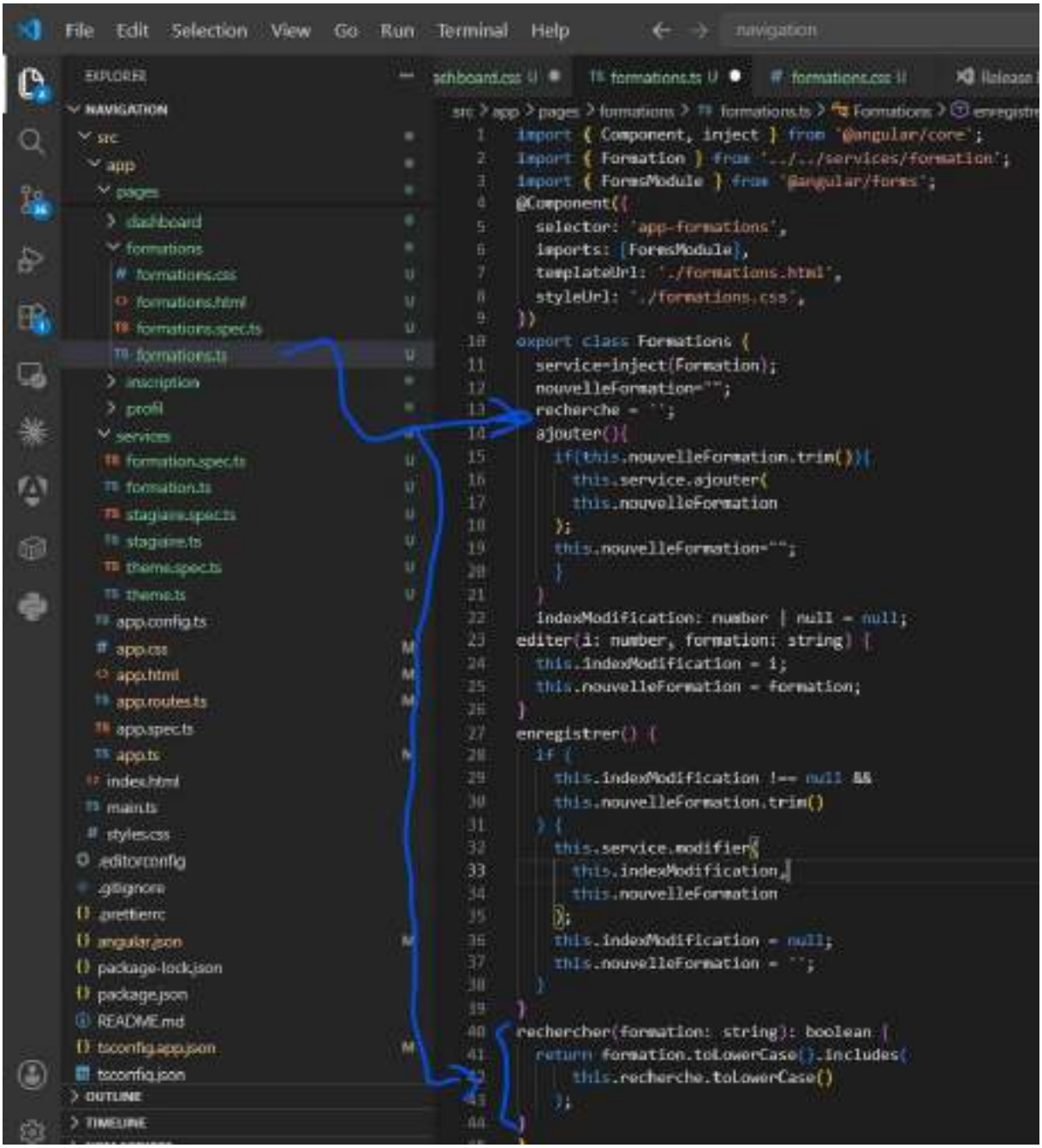
Rerchercher :

Formation.ts :

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

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

import { FormationService } from '../services/formation.service';

@Component({

  selector: 'app-formation',

  standalone: true,
```

```
 imports: [FormsModule],
 templateUrl: './formation.html',
 styleUrl: './formation.css'
})
export class Formation {
 service = inject(FormationService);
 nouvelleFormation = '';
 recherche = '';
 // Index de la formation à modifier
 // null = aucune modification en cours
 indexModification: number | null = null;
 ajouter() {
  if (this.nouvelleFormation.trim()) {
   this.service.ajouter(
    this.nouvelleFormation
   );
   this.nouvelleFormation = '';
  }

 }
 supprimer(i: number) {
  this.service.supprimer(i);

 }
 editer(i: number, formation: string) {
  this.indexModification = i;
```

```
    this.nouvelleFormation = formation;
  }
  enregistrer() {
    if (
      this.indexModification !== null &&
      this.nouvelleFormation.trim()
    ) {
      this.service.modifier(
        this.indexModification,
        this.nouvelleFormation
      );
      this.indexModification = null;
      this.nouvelleFormation = '';
    }
  }
  rechercher(formation: string): boolean {
    return formation
      .toLowerCase()
      .includes(
        this.recherche.toLowerCase()
      );
  }
}
```

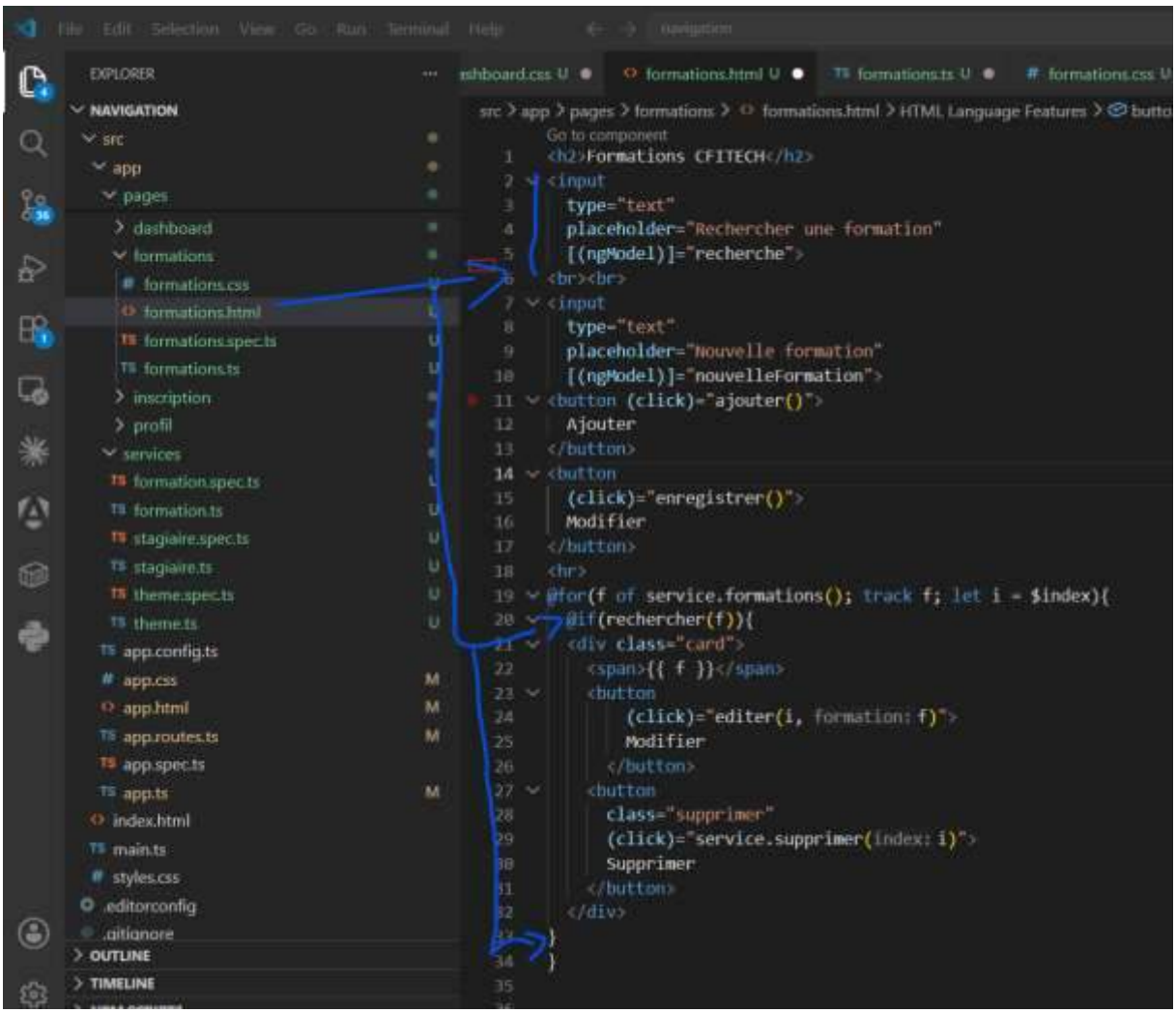


La méthode rechercher() permet de vérifier si une formation correspond au texte saisi par l'utilisateur. Pour rendre la recherche insensible aux majuscules et aux minuscules, la formation ainsi que le texte recherché sont d'abord convertis en minuscules grâce à toLowerCase(). La méthode includes() vérifie ensuite si le texte recherché est présent dans le nom de la formation. Si c'est le cas, la méthode renvoie true et la formation est affichée ; sinon, elle renvoie false et la formation n'est pas affichée. Cette technique est très utilisée dans les moteurs de recherche, les listes filtrées et les applications professionnelles.

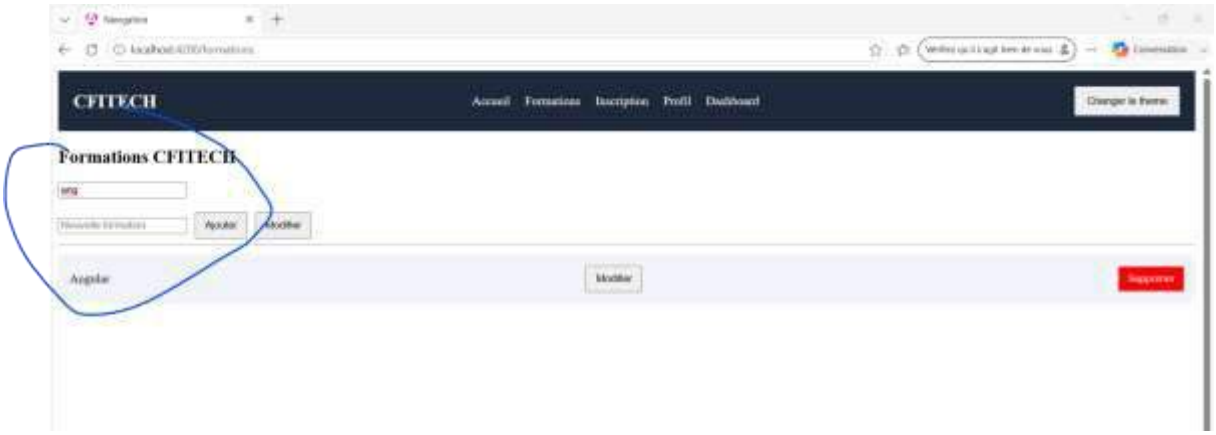
formation.html :

```
<h2>Formations CFITECH</h2>

<input

 type="text"

 placeholder="Rechercher une formation"

 [(ngModel)]="recherche">

<br><br>

<input

 type="text"

 placeholder="Nouvelle formation"

 [(ngModel)]="nouvelleFormation">

<button (click)="ajouter()">

 Ajouter

</button>

<button

 (click)="enregistrer()">

 Modifier

</button>

<hr>

@for(f of service.formations(); track f; let i = $index){

 @if(rechercher(f)){

 <div class="card">

  <span>{{ f }}</span>

  <button

    (click)="editer(i, f)">

    Modifier
```

```
    </button>
   <button
    class="supprimer"
    (click)="service.supprimer(i)">
    Supprimer
   </button>
 </div>
}
}
```



La condition @if(rechercher(f)) permet de filtrer les formations affichées. Pour chaque élément de la liste, Angular appelle la méthode rechercher(). Si cette méthode renvoie true, la formation est affichée. Si elle renvoie false, elle est masquée. Cette technique permet d'afficher uniquement les données correspondant au texte saisi par l'utilisateur.



5. Service métier (Business Service)

Définition

Un **service métier** est un service qui contient les règles de fonctionnement de l'application. Son rôle n'est pas seulement de stocker des données, mais surtout d'effectuer des traitements, des calculs, des validations ou des opérations qui répondent aux besoins du métier.

En d'autres termes, le composant gère l'interface utilisateur, tandis que le service métier gère la logique de l'application.

Le composant ne doit pas contenir toutes les règles de gestion.

Par exemple :

- vérifier que le nom est renseigné ;
- vérifier que l'email est valide ;
- vérifier que la formation existe ;
- enregistrer le stagiaire.

Toutes ces règles doivent être placées dans un service métier.

Ainsi, le composant reste simple et le code est plus facile à maintenir.

Répartition des responsabilités

Le composant

Le composant est responsable de :

- l'affichage ;

- les boutons ;
- les champs de saisie ;
- les événements utilisateur.

Exemple :

```
inscrire() {

 this.service.inscrire(
   this.nom,
   this.email,
   this.formation
 );

}
```

Le composant ne fait qu'appeler le service.

Le service métier

Le service est responsable de :

- vérifier les données ;
- effectuer les calculs ;
- appliquer les règles métier ;
- renvoyer un résultat.

Ex1 : Validation d'une inscription

Supposons que CFITECH impose la règle suivante :

Un stagiaire ne peut pas s'inscrire sans nom.

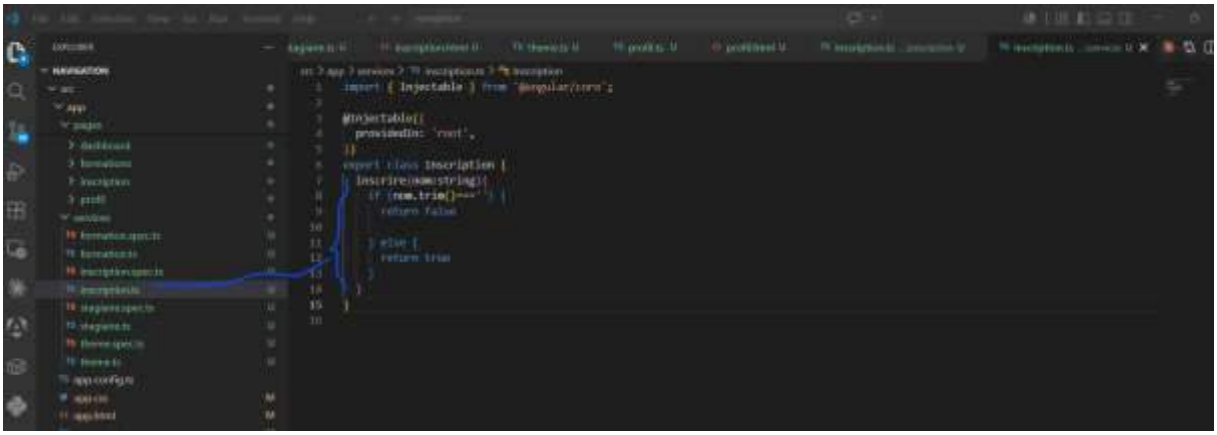
Cette règle n'a rien à voir avec l'affichage.

Elle appartient donc au service.

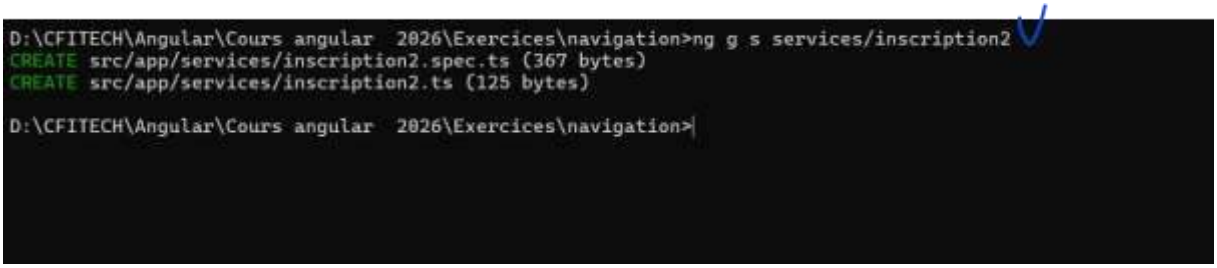
1. Le service

inscription.service.ts

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g s services/inscription
CREATE src/app/services/inscription.spec.ts (362 bytes)
CREATE src/app/services/inscription.ts (124 bytes)
```



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

@Injectable({

 providedIn: 'root'

})

export class InscriptionService {



 inscrire(nom: string): boolean {

   // Vérifie si le nom est vide

   if (nom.trim() === '') {

     return false;

   }

   return true;

 }

}
```



Le service contient la règle métier :

Un stagiaire doit obligatoirement renseigner son nom.

Le service ne connaît ni les boutons, ni les champs de saisie, ni l'interface graphique.

Son seul rôle est de vérifier si les données sont valides.

2. Le composant

inscription.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { Inscription2Service } from '../../services/inscription.service';

@Component({
  selector: 'app-inscription',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './inscription.html',
  styleUrl: './inscription.css'
})
export class Inscription {

  service = inject(Inscription2Service);
```

```
 nom = '';

 message = '';

 inscrire() {

  if (this.service.inscrire(this.nom)) {

   this.message = 'Inscription réussie.';

  } else {

   this.message = ' Le nom est obligatoire.';

  }

 }

}
```

Le composant :

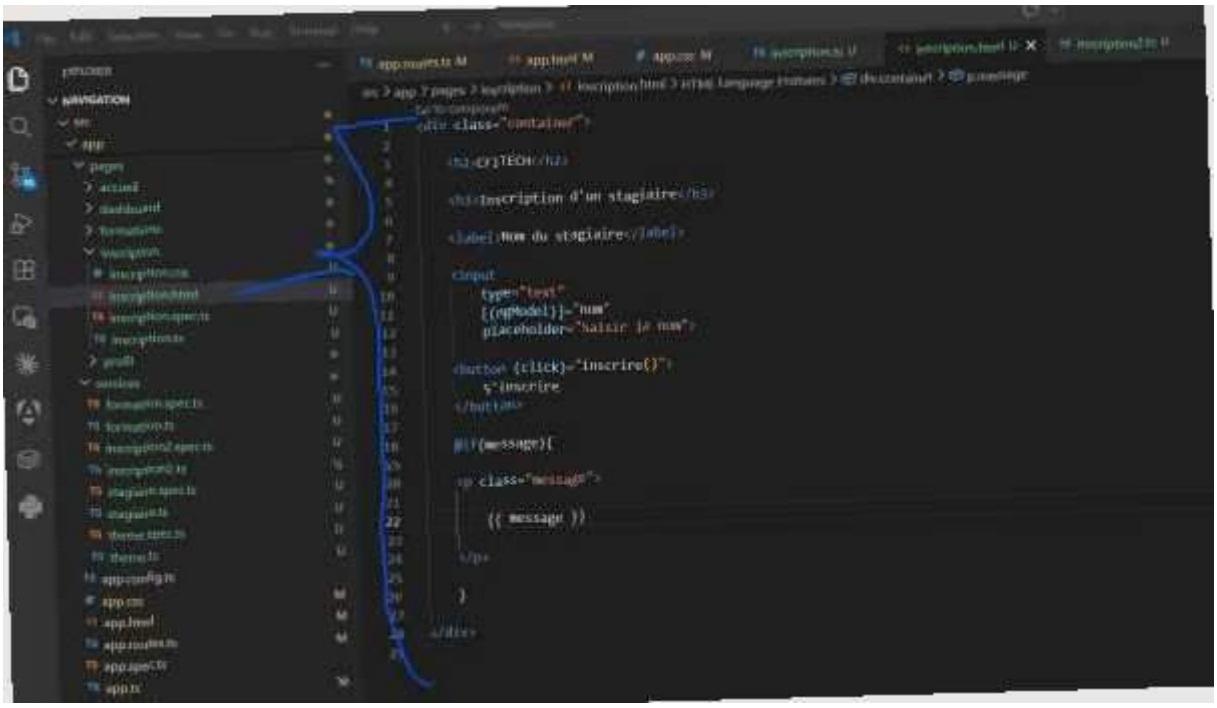
- récupère le nom saisi ;
- appelle le service ;
- affiche le résultat.

Il ne contient aucune règle métier.

3. Le HTML

inscription.html

```
<div class="container">

  <h2>CFITECH</h2>

  <h3>Inscription d'un stagiaire</h3>

  <label>Nom du stagiaire</label>

  <input
    type="text"
    [(ngModel)]="nom"
    placeholder="Saisir le nom">

  <button (click)="inscrire()">
    S'inscrire
  </button>

  @if(message){

  <p class="message">

    {{ message }}

  </p>

  }

</div>
```



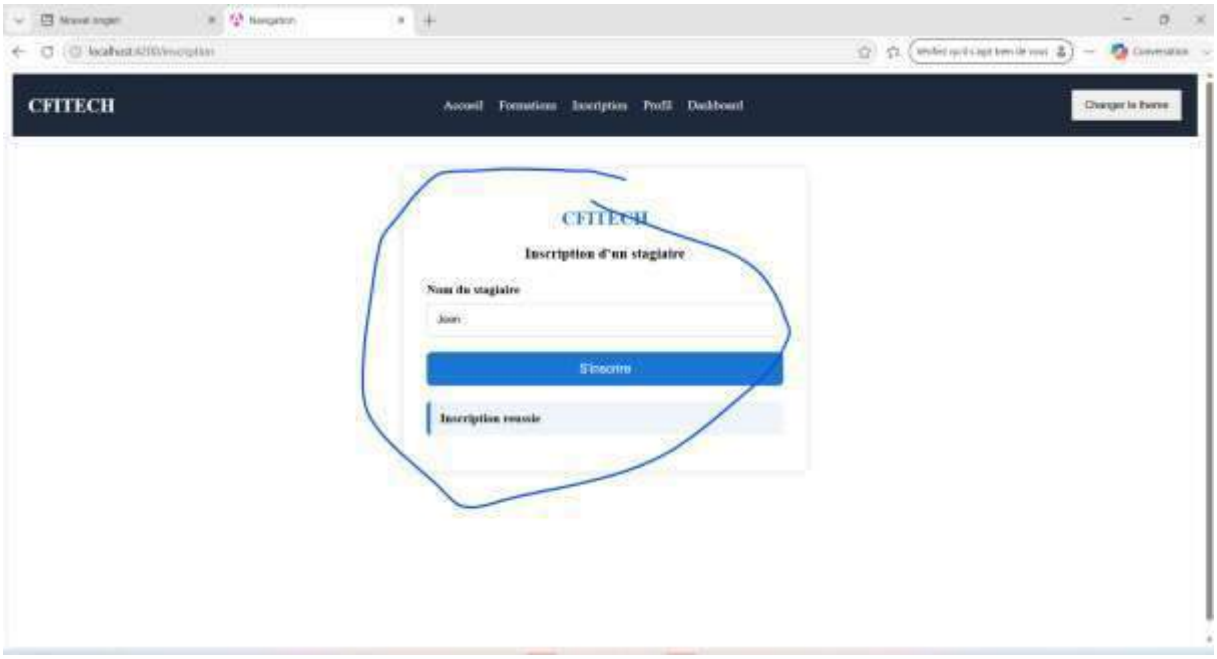
4. Le CSS

inscription.css

```
body {
   background: #f4f6f9;
}
.container {
   width: 450px;
   margin: 40px auto;
   background: white;
   padding: 30px;
   border-radius: 10px;
   box-shadow: 0 0 10px rgba(0,0,0,.15);
}
h2{
   color:#1976d2;
   text-align:center;
}
h3{
   margin-bottom:25px;
   text-align:center;
}
label{
   display:block;
   margin-bottom:8px;
   font-weight:bold;
}
```

```
input{
   width:100%;
   padding:12px;
   border:1px solid #ccc;
   border-radius:6px;
   margin-bottom:20px;
   box-sizing:border-box;
}
button{
   width:100%;
   padding:12px;
   background:#1976d2;
   color:white;
   border:none;
   border-radius:6px;
   cursor:pointer;
   font-size:16px;
}
button:hover{
   background:#1259a7;
}
.message{
   margin-top:20px;
   padding:12px;
   background:#eef5ff;
   border-left:5px solid #1976d2;
```

```
  border-radius:5px;

  font-weight:bold;

}
```



Dans cette application, le **composant** est uniquement responsable de l'interface utilisateur : il récupère les informations saisies, appelle le service et affiche le résultat à l'écran. Le **service métier** contient la règle de gestion de l'application, à savoir vérifier qu'un stagiaire ne peut pas être inscrit sans renseigner son nom. Cette séparation des responsabilités est une bonne pratique essentielle en Angular, car elle rend le code plus clair, plus réutilisable et plus facile à maintenir dans les projets professionnels.

Ex 2 : Calcul automatique du coût d'une formation

Le centre de formation **CFITECH** propose plusieurs formations. Chaque formation possède un tarif fixe.

Le directeur souhaite que le calcul du prix ne soit **jamais effectué dans le composant**. En effet, si les tarifs changent un jour, il sera plus simple de modifier uniquement le service.

Votre mission consiste donc à créer un **service métier** chargé de retourner automatiquement le prix correspondant à la formation choisie.

Tarifs des formations

Formation	Prix
Angular	350 €

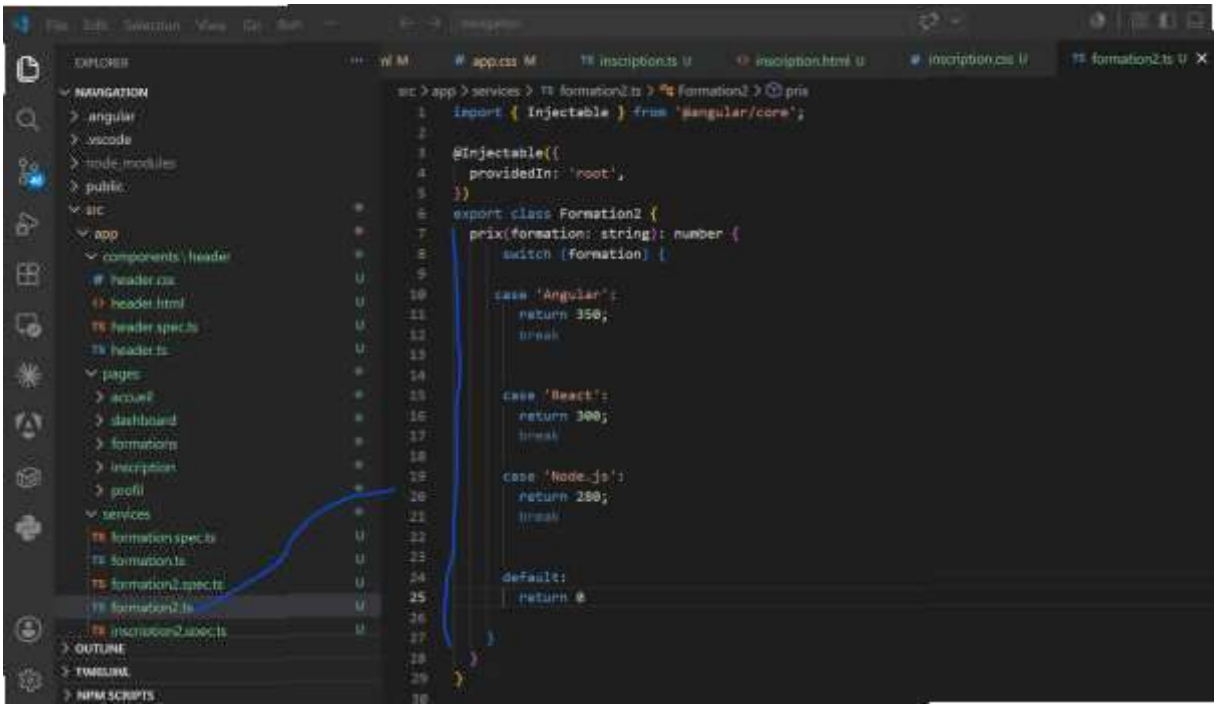
Formation	Prix
React	300 €
Node.js	280 €

1. Service

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g s services/formation2
CREATE src/app/services/formation2.spec.ts (357 bytes)
CREATE src/app/services/formation2.ts (123 bytes)
```

Formation2.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class Formation2Service {

  prix(formation: string): number {

    switch (formation) {

      case 'Angular':
        return 350;

      case 'React':
        return 300;

      case 'Node.js':
        return 280;

      default:
        return 0;

    }

  }

}
```



2. Composant

formation.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { FormationService } from '../../services/formation.service';

@Component({
 selector: 'app-formation',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './formation.html',
 styleUrl: './formation.css'
})
export class Formation {

 service = inject(FormationService);

 formation = '';

 prix = 0;

 calculer() {

  this.prix = this.service.prix(this.formation);

 }

}
```

3. HTML

formation.html

```
<div class="container">

  <h2>CFITECH</h2>

  <h3>Calcul du prix d'une formation</h3>

  <label>Choisir une formation</label>

  <select [(ngModel)]="formation">

    <option value="">-- Sélectionner --</option>
    <option>Angular</option>
    <option>React</option>
    <option>Node.js</option>

  </select>

  <button (click)="calculer()">

    Calculer le prix

  </button>

  @if(prix > 0){

    <div class="resultat">

      Prix de la formation :

      <strong>{{ prix }} €</strong>

    </div>

  }

</div>
```

4. CSS

formation.css

```
body{
```

```
  background:#f4f6f9;
}
.container{
  width:450px;
  margin:40px auto;
  background:white;
  padding:30px;
  border-radius:10px;
  box-shadow:0 0 10px rgba(0,0,0,.15);
}
h2{
  text-align:center;
  color:#1976d2;
}
h3{
  text-align:center;
  margin-bottom:25px;
}
label{
  display:block;
  font-weight:bold;
  margin-bottom:8px;
}
select{
  width:100%;
  padding:10px;
```

```
    margin-bottom:20px;
    border-radius:6px;
    border:1px solid #ccc;
}
button{
    width:100%;
    padding:12px;
    background:#1976d2;
    color:white;
    border:none;
    border-radius:6px;
    cursor:pointer;
    font-size:16px;
}
button:hover{
    background:#1259a7;
}
.resultat{
    margin-top:25px;
    padding:15px;
    background:#eef5ff;
    border-left:5px solid #1976d2;
    border-radius:6px;
    font-size:18px;
    text-align:center;
}
```

Dans cet exercice, **le composant ne connaît pas les tarifs des formations**. Son rôle est uniquement de récupérer le choix de l'utilisateur, d'appeler le service et d'afficher le résultat. Toute la logique métier liée aux prix est centralisée dans le **FormationService**. Si un tarif change (par exemple Angular passe de **250 €** à **280 €**), il suffit de modifier le service sans toucher au composant. Cette séparation des responsabilités rend l'application plus claire, plus facile à maintenir et conforme aux bonnes pratiques utilisées dans les projets Angular professionnels.

Ex3 : Vérifier si un stagiaire est majeur

Le centre de formation **CFITECH** organise certaines formations réservées aux personnes majeures.

Avant de valider une inscription, l'application doit vérifier automatiquement que le stagiaire a **au moins 18 ans**.

Le directeur demande que cette règle soit placée dans un **service métier**, afin que le composant ne contienne aucune règle de gestion.

1. Le service

inscription.service.ts

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

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

  verifierAge(age: number): boolean {

    return age >= 18;

  }

}
```

2. Le composant

inscription.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { InscriptionService } from '../../services/inscription.service';

@Component({
 selector: 'app-inscription',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './inscription.html',
 styleUrl: './inscription.css'
})
export class Inscription {

 service = inject(InscriptionService);

 age = 0;

 message = '';

 verifier() {

  if (this.service.verifierAge(this.age)) {

   this.message = ' Inscription autorisée';

  } else {

   this.message = 'Inscription refusée : le stagiaire doit être majeur';

  }

 }

}
```

3. Le HTML

inscription.html

```
<div class="container">

  <h2>CFITECH</h2>

  <h3>Vérification de l'âge du stagiaire</h3>
```

```
  <label>Âge du stagiaire</label>

  <input
    type="number"
    [(ngModel)]="age"
    placeholder="Saisir l'âge">

  <button (click)="verifier()">

    Vérifier

  </button>

  @if(message){

    <div class="resultat">

      {{ message }}

    </div>

  }

</div>
```

4. Le CSS

inscription.css

```
body{

  background:#f4f6f9;

}

.container{

  width:450px;

  margin:40px auto;

  background:white;

  padding:30px;

  border-radius:10px;

  box-shadow:0 0 10px rgba(0,0,0,.15);
```

```
}
h2{
  color:#1976d2;
  text-align:center;
}
h3{
  text-align:center;
  margin-bottom:25px;
}
label{
  display:block;
  font-weight:bold;
  margin-bottom:8px;
}
input{
  width:100%;
  padding:12px;
  border:1px solid #ccc;
  border-radius:6px;
  margin-bottom:20px;
  box-sizing:border-box;
}
button{
  width:100%;
  padding:12px;
  background:#1976d2;
```

```
  color:white;
  border:none;
  border-radius:6px;
  cursor:pointer;
  font-size:16px;
}
button:hover{
  background:#1259a7;
}
.resultat{
  margin-top:20px;
  padding:15px;
  background:#eef5ff;
  border-left:5px solid #1976d2;
  border-radius:6px;
  font-size:18px;
  text-align:center;
  font-weight:bold;
}
```

Ex 4 : Génération automatique d'un numéro de dossier

Le centre de formation **CFITECH** reçoit chaque jour de nouvelles inscriptions.

Pour retrouver facilement les dossiers des stagiaires, chaque inscription doit recevoir un **numéro de dossier unique**.

Le premier stagiaire inscrit recevra le numéro **1001**, le second **1002**, le troisième **1003**, etc.

Le directeur demande que cette logique soit placée dans un **service métier** afin que le composant ne connaisse pas la manière dont les numéros sont générés.

1. Le service

dossier.service.ts

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

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

  numero = 1000;

  genererNumero(): number {

    this.numero++;

    return this.numero;

  }

}
```

2. Le composant

inscription.ts

```
import { Component, inject } from '@angular/core';
import { DossierService } from '../../services/dossier.service';

@Component({
  selector: 'app-inscription',
  standalone: true,
  templateUrl: './inscription.html',
  styleUrl: './inscription.css'
})
export class Inscription {

  service = inject(DossierService);

  numero = 0;

  generer() {

    this.numero = this.service.genererNumero();

  }

}
```

HTML

```
<div class="container">
  <h2>CFITECH</h2>
  <h3>Génération automatique d'un dossier</h3>
  <button (click)="generer()">
    Générer un numéro
  </button>
  @if(numero > 0){
    <div class="resultat">
      Numéro du dossier :
      <strong>{{ numero }}</strong>
    </div>
  }
</div>
```

CSS

```
.container{
  width:450px;
  margin:40px auto;
  background:white;
  padding:30px;
  border-radius:10px;
  box-shadow:0 0 10px rgba(0,0,0,.2);
}
h2{
```

```
    text-align:center;
    color:#1565c0;
}
h3{
    text-align:center;
}
button{
    width:100%;
    padding:12px;
    border:none;
    border-radius:6px;
    background:#1565c0;
    color:white;
    font-size:16px;
    cursor:pointer;
}
button:hover{
    background:#0d47a1;
}
.resultat{
    margin-top:20px;
    padding:20px;
    text-align:center;
    background:#eef5ff;
    border-left:5px solid #1565c0;
    font-size:22px;
```

```
  font-weight:bold;

}
```

6. Service d'authentification

Très courant.

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

 connecte = signal(false);

 login() {
  this.connecte.set(true);
 }

 logout() {
  this.connecte.set(false);
 }

}
```

Applications :

- connexion utilisateur
- déconnexion
- rôle administrateur

Ex: Authentification d'un utilisateur avec un Service

Le centre de formation **CFITECH** développe une application permettant aux stagiaires d'accéder à leur espace personnel.

Avant d'accéder à cet espace, le stagiaire doit obligatoirement se connecter.

Lorsque l'utilisateur clique sur **Se connecter**, l'application doit mémoriser qu'il est connecté.

Lorsque l'utilisateur clique sur **Se déconnecter**, l'application doit mémoriser qu'il n'est plus connecté.

Le directeur souhaite que toute cette logique soit placée dans un **service d'authentification** afin que plusieurs composants puissent connaître l'état de connexion de l'utilisateur.

1. Le service

auth.service.ts

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

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

  connecte = signal(false);

  login() {

    this.connecte.set(true);

  }

  logout() {

    this.connecte.set(false);

  }

}
```

2. Le composant

connexion.ts

```
import { Component, inject } from '@angular/core';
import { AuthService } from '../../services/auth.service';

@Component({
  selector: 'app-connexion',
  standalone: true,
  templateUrl: './connexion.html',
  styleUrl: './connexion.css'
})
export class Connexion {

  service = inject(AuthService);

}
```

3. HTML

connexion.html

```
<div class="container">

  <h2>CFITECH</h2>

  <h3>Espace stagiaire</h3>

  @if(service.connecte()){

    <div class="succes">

      <h2>Bienvenue !</h2>

      <p>Vous êtes connecté à votre espace personnel.</p>

      <button
        class="logout"
        (click)="service.logout()">

        Se déconnecter

      </button>

    </div>

  }@else{

    <div class="login">

      <p>

        Vous devez vous connecter
        pour accéder à votre espace.

      </p>

      <button
        class="login-btn"
        (click)="service.login()">

        Se connecter

      </button>

    </div>

  }
```

```
</div>
```

4. CSS

```
body{
  background:#eef2f7;
}
.container{
  width:500px;
  margin:40px auto;
  background:white;
  padding:30px;
  border-radius:10px;
  box-shadow:0 0 15px rgba(0,0,0,.15);
  text-align:center;
}
h2{
  color:#1565c0;
}
.login,
.succes{
  margin-top:30px;
}
button{
  margin-top:20px;
  padding:12px 25px;
  border:none;
```

```
    border-radius:6px;
    cursor:pointer;
    color:white;
    font-size:16px;
}
.login-btn{
    background:#2e7d32;
}
.login-btn:hover{
    background:#1b5e20;
}
.logout{
    background:#c62828;
}
.logout:hover{
    background:#8e0000;
}
.succes{
    background:#e8f5e9;
    padding:20px;
    border-left:5px solid green;
    border-radius:8px;
}
.login{
    background:#fff3cd;
    padding:20px;
```

```
  border-left:5px solid orange;

  border-radius:8px;

}
```

Exercice : Gestion des rôles avec un Service d'authentification

Le centre de formation **CFITECH** possède deux types d'utilisateurs :

- **Le stagiaire** : il peut consulter son profil et les formations auxquelles il est inscrit.
- **L'administrateur** : il peut gérer les stagiaires, ajouter des formations, modifier des informations et supprimer des dossiers.

Lorsque l'utilisateur se connecte, l'application doit mémoriser son rôle afin d'afficher l'interface adaptée.

Le directeur souhaite que cette logique soit entièrement gérée par un **service d'authentification**.

Étape 1 : Le service

auth.service.ts

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

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

 // État de connexion
 connecte = signal(false);

 // Rôle de l'utilisateur
 role = signal('');

 // Connexion d'un stagiaire
 loginStagiaire() {

  this.connecte.set(true);

  this.role.set('stagiaire');

 }

 // Connexion d'un administrateur
```

```
 loginAdmin() {

   this.connecte.set(true);

   this.role.set('admin');

 }

 // Déconnexion
 logout() {

   this.connecte.set(false);

   this.role.set('');

 }

}
```

Étape 2 : Le composant

connexion.ts

```
import { Component, inject } from '@angular/core';
import { AuthService } from '../../services/auth.service';

@Component({
 selector: 'app-connexion',
 standalone: true,
 templateUrl: './connexion.html',
 styleUrl: './connexion.css'
})
export class Connexion {

 service = inject(AuthService);

}
```

Étape 3 : Le HTML

connexion.html

```
<div class="container">

   <h2>CFITECH</h2>

   <h3>Gestion des rôles</h3>
```

```
@if(!service.connecte()){

  <button
    class="stagiaire"
    (click)="service.loginStagiaire()">

    Connexion Stagiaire

  </button>

  <button
    class="admin"
    (click)="service.loginAdmin()">

    Connexion Administrateur

  </button>

}@else{

  @if(service.role() === 'stagiaire'){

    <div class="card stagiaire-card">

      <h2>Bienvenue Stagiaire</h2>

      <p>

        Vous pouvez consulter votre profil
        et vos formations.

      </p>

    </div>

  }

  @if(service.role() === 'admin'){

    <div class="card admin-card">

      <h2>Bienvenue Administrateur</h2>

      <p>

        Vous pouvez gérer les stagiaires,
        les formations et les inscriptions.

      </p>

    </div>
```

```
    }

    <button
      class="logout"
      (click)="service.logout()">

      Se déconnecter

    </button>

  }

</div>
```

Étape 4 : Le CSS

```
body{

  background:#eef2f7;

}

.container{

  width:550px;

  margin:40px auto;

  background:white;

  padding:30px;

  border-radius:10px;

  text-align:center;

  box-shadow:0 0 15px rgba(0,0,0,.15);

}

h2{

  color:#1565c0;

}

button{

  padding:12px 25px;
```

```
    margin:10px;
    border:none;
    border-radius:6px;
    color:white;
    cursor:pointer;
    font-size:15px;
}
.stagiaire{
    background:#2e7d32;
}
.stagiaire:hover{
    background:#1b5e20;
}
.admin{
    background:#ff9800;
}
.admin:hover{
    background:#ef6c00;
}
.logout{
    background:#c62828;
}
.logout:hover{
    background:#8e0000;
}
.card{
```

```
    margin-top:25px;
    padding:20px;
    border-radius:8px;
}
.stagiaire-card{
    background:#e8f5e9;
    border-left:5px solid green;
}
.admin-card{
    background:#fff3e0;
    border-left:5px solid orange;
}
```

7. Service de permissions

Gestion des permissions avec un Service

Ex : Le centre de formation **CFITECH** possède une application permettant de gérer les formations.

Tous les utilisateurs peuvent consulter les formations.

En revanche, **seuls les administrateurs** ont le droit :

- d'ajouter une formation ;
- de modifier une formation ;
- de supprimer une formation.

Les stagiaires n'ont pas ces droits.

Le directeur souhaite que cette règle soit centralisée dans un **PermissionService**.

1. permission.service.ts

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

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

 estAdmin = signal(false);

 formations = signal([
  'Angular',
  'React',
  'Node.js',
  'Spring Boot'
 ]);

 devenirAdmin() {

  this.estAdmin.set(true);

 }

 devenirStagiaire() {

  this.estAdmin.set(false);

 }

 supprimer(index: number) {

  this.formations.update(liste =>
   liste.filter((_, i) => i !== index)
  );

 }

}
```

2. formations.ts

```
import { Component, inject } from '@angular/core';
import { PermissionService } from '../../services/permission.service';

@Component({
 selector: 'app-formations',
 standalone: true,
 imports: [],
```

```
 templateUrl: './formations.html',
 styleUrl: './formations.css'
})
export class Formations {

 service = inject(PermissionService);

}
```

3. formations.html

```
<div class="container">

  <h1>CFITECH</h1>

  <h2>Gestion des permissions</h2>

  <div class="roles">

    <button
      class="admin"
      (click)="service.devenirAdmin()">

      Administrateur

    </button>

    <button
      class="stagiaire"
      (click)="service.devenirStagiaire()">

      Stagiaire

    </button>

  </div>

  <hr>

  @if(service.estAdmin()){

    <div class="message admin-message">

      Vous êtes connecté en tant qu'
      <strong>Administrateur</strong>

    </div>

  }@else{
```

```
      <div class="message stagiaire-message">

        Vous êtes connecté en tant que
        <strong>Stagiaire</strong>

      </div>

    }

    @for(f of service.formations(); track f; let i = $index){

      <div class="card">

        <span>

          {{ f }}

        </span>

        @if(service.estAdmin()){

          <div class="actions">

            <button class="modifier">

              Modifier

            </button>

            <button
              class="supprimer"
              (click)="service.supprimer(i)">

              Supprimer

            </button>

          </div>

        }

      </div>

    }

  </div>
```

4. formations.css

```
body{
   background:#eef2f7;
}
.container{
   width:750px;
   margin:40px auto;
   background:white;
   padding:30px;
   border-radius:10px;
   box-shadow:0 0 10px rgba(0,0,0,.2);
}
h1{
   text-align:center;
   color:#1565c0;
}
h2{
   text-align:center;
}
.roles{
   display:flex;
   justify-content:center;
   gap:20px;
   margin:20px 0;
}
button{
```

```
    padding:10px 18px;
    border:none;
    border-radius:6px;
    cursor:pointer;
    color:white;
}
.admin{
    background:#1565c0;
}
.stagiaire{
    background:#43a047;
}
.message{
    padding:15px;
    margin:20px 0;
    border-radius:8px;
    text-align:center;
    font-size:18px;
}
.admin-message{
    background:#fff3cd;
    border-left:5px solid orange;
}
.stagiaire-message{
    background:#e8f5e9;
    border-left:5px solid green;
```

```
}
.card{
   display:flex;
   justify-content:space-between;
   align-items:center;
   margin-top:15px;
   padding:15px;
   background:#f4f6f9;
   border-radius:8px;
}
.actions{
   display:flex;
   gap:10px;
}
.modifier{
   background:orange;
}
.supprimer{
   background:#d32f2f;
}
```

8. Service de configuration

Qu'est-ce qu'un Service de configuration ?

Un **Service de configuration** est un service qui centralise toutes les informations de configuration de l'application.

Au lieu d'écrire ces informations dans plusieurs composants, on les place dans un seul service.

Ainsi, si une information change, il suffit de modifier le service, et toute l'application utilise automatiquement la nouvelle valeur.

Ex : Imaginons que l'application de **CFITECH** affiche sur plusieurs pages :

- le nom du centre ;
- l'adresse ;
- le numéro de téléphone ;
- l'adresse e-mail ;
- le logo ;
- les horaires d'ouverture.

Sans service, chaque composant devrait écrire ces informations.

Par exemple :

nomCentre = 'CFITECH';

telephone = '02 445 39 08';

email = 'info@cfitech.be';

Si le téléphone change, il faudra modifier **tous les composants**.

Ce n'est pas une bonne pratique.

Avec un **Service de configuration**, toutes ces informations sont centralisées dans un seul endroit.

Arborescence

```
src
└── app

    ├── pages
    │   └── accueil
    │       ├── accueil.ts
    │       ├── accueil.html
```

```
│       └── accueil.css
│
└── services
    └── configuration.service.ts
```

1. Le service

configuration.service.ts

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

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

  nomCentre = signal('CFITECH');

  adresse = signal('rue de l'eglise st anne 63, 1081 koekelberg');

  telephone = signal('02 445 39 08');

  email = signal('info@cfitech.be');



}
```

2. Le composant

accueil.ts

```
import { Component, inject } from '@angular/core';
import { ConfigurationService } from '../../services/configuration.service';

@Component({
  selector: 'app-accueil',
  standalone: true,
  imports: [],
  templateUrl: './accueil.html',
  styleUrl: './accueil.css'
})
export class Accueil {

  service = inject(ConfigurationService);

}
```

3. Le HTML

accueil.html

```
<div class="container">

  <h1>{{ service.nomCentre() }}</h1>

  <div class="card">

    <h3>Informations du centre</h3>

    <p>

      Adresse :
      {{ service.adresse() }}

    </p>

    <p>

      Téléphone :
      {{ service.telephone() }}

    </p>

    <p>

      Email :
      {{ service.email() }}

    </p>



  </div>

</div>
```

4. Le CSS

```
body{

  background:#eef2f7;

}
```

```
.container{
   width:600px;
   margin:40px auto;
}
h1{
   text-align:center;
   color:#1565c0;
}
.card{
   margin-top:30px;
   background:white;
   padding:25px;
   border-radius:10px;
   box-shadow:0 0 10px rgba(0,0,0,.15);
}
.card p{
   font-size:18px;
   margin:15px 0;
}
```

9. Service de notifications

Qu'est-ce qu'un Service de notifications ?

Un **Service de notifications** est un service chargé d'afficher des messages à l'utilisateur.

Ces messages permettent de l'informer lorsqu'une action réussit, échoue ou nécessite son attention.

Au lieu d'utiliser plusieurs alert() dans les composants, on centralise cette logique dans un seul service.

Ex : Imaginons que plusieurs pages doivent afficher des messages :

- Inscription réussie
- Formation ajoutée
- Formation supprimée
- Profil modifié
- Paiement effectué

Sans service, chaque composant ferait :

alert('Formation ajoutée');

ou

alert('Profil enregistré');

Le code est dupliqué dans toute l'application.

Exercice :

Le centre de formation **CFITECH** permet d'ajouter des formations.

Après chaque ajout, une notification doit apparaître pendant quelques secondes.

Le directeur souhaite que cette fonctionnalité soit gérée par un **NotificationService**.

1. Le service

notification.service.ts

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

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

```
 message = signal('');
 afficher(message: string) {
   this.message.set(message);
   setTimeout(() => {
     this.message.set('');
   }, 3000);
 }
}
```

Au départ :

message = signal('');

Le message est vide.

Aucune notification n'est affichée.

Lorsque l'on appelle :

this.afficher('Formation ajoutée');

Angular exécute :

this.message.set('Formation ajoutée');

Le Signal contient maintenant :

Formation ajoutée

La notification apparaît immédiatement.

Puis :

```
setTimeout(() => {
   this.message.set('');
},3000);
```

attend **3 secondes** avant de vider le message.

La notification disparaît automatiquement.

2. Le composant

formations.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { NotificationService } from '../../services/notification.service';

@Component({
 selector: 'app-formations',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './formations.html',
 styleUrl: './formations.css'
})
export class Formations {

 service = inject(NotificationService);

 formation = '';

 ajouter() {

  if(this.formation.trim()){

   this.service.afficher(

    'La formation "' +

    this.formation +

    '" a été ajoutée avec succès.'

   );

   this.formation='';

  }

 }

}
```

3. Le HTML

formations.html

```
<div class="container">
  <h2>CFITECH</h2>
  <h3>Ajout d'une formation</h3>
  <input
    type="text"
    [(ngModel)]="formation"
    placeholder="Nom de la formation">
  <button (click)="ajouter()">
    Ajouter
  </button>
  @if(service.message()){
    <div class="notification">
      {{ service.message() }}
    </div>
  }
</div>
```

4. Le CSS

```
body{
  background:#eef2f7;
}
.container{
  width:500px;
  margin:40px auto;
```

```
    background:white;
    padding:30px;
    border-radius:10px;
    box-shadow:0 0 10px rgba(0,0,0,.2);
}
h2{
    text-align:center;
    color:#1565c0;
}
input{
    width:100%;
    padding:12px;
    margin:15px 0;
    border:1px solid #ccc;
    border-radius:6px;
}
button{
    width:100%;
    padding:12px;
    background:#1565c0;
    color:white;
    border:none;
    border-radius:6px;
    cursor:pointer;
}
.notification{
```

```
  margin-top:20px;
  padding:15px;
  background:#d4edda;
  border-left:5px solid green;
  border-radius:6px;
  font-size:17px;
  animation:apparition .5s;
}
@keyframes apparition{
  from{
    opacity:0;
    transform:translateY(-15px);
  }
  to{
    opacity:1;
    transform:translateY(0);
  }
}
```

10. Service de panier

Le **Service de panier (Cart Service)** est l'un des services les plus utilisés dans les applications professionnelles. On le retrouve dans :

- Amazon
- Jumia
- Decathlon
- Carrefour
- Shopify
- Uber Eats

L'objectif est que **plusieurs composants puissent accéder au même panier**. Par exemple, un produit est ajouté depuis la page "Produits", mais le nombre d'articles est affiché dans la barre de navigation. Sans service, ce partage serait très difficile.

Exercice : Gestion d'un panier d'achat avec un Service

CFITECH ouvre une boutique en ligne permettant aux stagiaires d'acheter des livres de formation.

Les livres disponibles sont :

- Angular
- React
- Node.js
- Spring Boot

Lorsqu'un utilisateur clique sur **Ajouter au panier**, le livre doit être ajouté au panier.

Le panier doit afficher :

- la liste des livres ajoutés ;
- le nombre total de livres.

Le directeur souhaite que toute cette logique soit gérée dans un **CartService**.

Arborescence

```
src
└── app
    ├── pages
    │   └── boutique
    │       ├── boutique.ts
    │       ├── boutique.html
    │       └── boutique.css
    │
    └── services
        └── panier.service.ts
```

1. Le service

panier.service.ts

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

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

  panier = signal<string[]>([]);

  ajouter(produit: string) {

    this.panier.update(
      liste => [...liste, produit]
    );

  }

  supprimer(index: number) {

    this.panier.update(
      liste => liste.filter((_, i) => i !== index)
    );

  }

  nombreArticles() {

    return this.panier().length;

  }

}
```

Lorsque l'on clique sur **Ajouter au panier** :

```
this.ajouter('Angular');
```

Angular exécute :

```
this.panier.update(
   liste => [...liste, produit]
);
```

2. Le composant

boutique.ts

```
import { Component, inject } from '@angular/core';
import { PanierService } from '../../services/panier.service';

@Component({
 selector: 'app-boutique',
 standalone: true,
 imports: [],
 templateUrl: './boutique.html',
 styleUrl: './boutique.css'
})
export class Boutique {

 service = inject(PanierService);

 produits = [

  'Angular',

  'React',

  'Node.js',

  'Spring Boot'

 ];

}
```

3. Le HTML

boutique.html

```
<div class="container">

  <h1>CFITECH Store</h1>

  <h2>Boutique de formations</h2>

  <div class="panier">

     Articles dans le panier :

    <strong>

      {{ service.nombreArticles() }}
```

```
  </strong>
</div>
<h3>Produits disponibles</h3>
@for(produit of produits; track produit){
  <div class="card">
    <span>
      {{ produit }}
    </span>
    <button
      (click)="service.ajouter(produit)">
      Ajouter au panier
    </button>
  </div>
}
<hr>
<h3>Mon panier</h3>
@if(service.panier().length === 0){
  <p>
    Votre panier est vide.
  </p>
}
@for(article of service.panier(); track article; let i = $index){
  <div class="card">
    <span>
      {{ article }}
    </span>
```

```
        <button
          class="supprimer"
          (click)="service.supprimer(i)">
          Supprimer
        </button>
      </div>
  }
</div>
```

4. Le CSS

boutique.css

```
body{
  background:#eef2f7;
}
.container{
  width:750px;
  margin:40px auto;
  background:white;
  padding:30px;
  border-radius:10px;
  box-shadow:0 0 12px rgba(0,0,0,.15);
}
h1{
  text-align:center;
  color:#1565c0;
```

```
}
.panier{
  background:#1565c0;
  color:white;
  padding:15px;
  text-align:center;
  border-radius:8px;
  margin-bottom:25px;
  font-size:20px;
}
.card{
  display:flex;
  justify-content:space-between;
  align-items:center;
  background:#f4f6f9;
  padding:15px;
  margin-top:12px;
  border-radius:8px;
}
button{
  background:#2e7d32;
  color:white;
  border:none;
  padding:10px 15px;
  border-radius:5px;
  cursor:pointer;
```

```
}

button:hover{

  background:#1b5e20;

}

.supprimer{

  background:#d32f2f;

}

.supprimer:hover{

  background:#b71c1c;

}
```

11. Service de recherche

Qu'est-ce qu'un Service de recherche ?

Un **Service de recherche** est un service chargé de filtrer ou rechercher des données selon un ou plusieurs critères.

Le composant ne réalise jamais la recherche lui-même.

Il transmet simplement les informations au service, qui renvoie le résultat.

Ex : Imaginons que plusieurs pages doivent rechercher des formations.

Sans service, chaque composant ferait :

```
formations.filter(f =>
  f.toLowerCase().includes(recherche.toLowerCase())
);
```

Cette logique serait répétée partout.

Avec un **SearchService**, toute la logique est centralisée dans un seul endroit.

Exercice :

Le centre de formation **CFITECH** possède plus de **500 formations**.

Les utilisateurs doivent pouvoir rechercher rapidement une formation.

Le directeur souhaite que cette recherche soit entièrement gérée dans un **SearchService**.

Étape 1 : Le service

recherche.service.ts

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

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

  rechercher(
    liste: string[],
    motCle: string
  ): string[] {

    return liste.filter(formation =>

      formation
        .toLowerCase()
        .includes(
          motCle.toLowerCase()
        )

    );

  }

}
```

La méthode

rechercher(liste, motCle)

reçoit :

- la liste complète des formations ;

- le texte saisi par l'utilisateur.

Elle renvoie uniquement les formations contenant ce texte.

Étape 2 : Le composant

formations.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { RechercheService } from '../../services/recherche.service';

@Component({
 selector: 'app-formations',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './formations.html',
 styleUrl: './formations.css'
})
export class Formations {

 service = inject(RechercheService);

 recherche = '';

 formations = [

  'Angular',

  'React',

  'Node.js',

  'Spring Boot',

  'Laravel',

  'PHP',

  'Java'

 ];

}
```

Étape 3 : Le HTML

formations.html

```
<div class="container">
  <h1>CFITECH</h1>
  <h2>Recherche d'une formation</h2>
  <input
    type="text"
    [(ngModel)]="recherche"
    placeholder="Rechercher une formation">
  <hr>
  @for(
   formation of service.rechercher(formations, recherche);
   track formation
  ){
    <div class="card">
      {{ formation }}
    </div>
  }
</div>
```

Étape 4 : Le CSS

```
body{
  background:#eef2f7;
}
.container{
  width:600px;
  margin:40px auto;
```

```
    background:white;
    padding:30px;
    border-radius:10px;
    box-shadow:0 0 10px rgba(0,0,0,.2);
}
h1{
    text-align:center;
    color:#1565c0;
}
input{
    width:100%;
    padding:12px;
    border:1px solid #ccc;
    border-radius:6px;
    margin-bottom:20px;
    font-size:16px;
}
.card{
    padding:15px;
    margin-top:10px;
    background:#f4f6f9;
    border-radius:8px;
    border-left:5px solid #1565c0;
}
```

12. Service API

Qu'est-ce qu'un Service API ?

Un **Service API** est un service chargé de communiquer avec un serveur (Backend).

Au lieu que le composant contacte directement le serveur, il demande au service d'effectuer cette communication.

Le service est donc l'intermédiaire entre Angular et le serveur.

Ex : Imaginons que CFITECH possède une base de données contenant des milliers de stagiaires.

Le composant ne doit jamais accéder directement au serveur.

Il demande simplement au service :

"Donne-moi la liste des stagiaires."

Le service contacte alors le serveur, récupère les données et les renvoie au composant.

Exercice :

Le centre de formation **CFITECH** possède une base de données contenant toutes les formations.

Le directeur souhaite afficher ces formations dans Angular.

Les données ne seront plus écrites dans le composant.

Elles devront être récupérées depuis une API.

Étape 1 : Importer HttpClient

Dans **app.config.ts**

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

export const appConfig: ApplicationConfig = {

 providers: [

  provideHttpClient()

 ]
```

};

Angular ne sait pas communiquer avec Internet par défaut.

Le provider :

provideHttpClient()

active toutes les fonctionnalités HTTP.

Sans lui :

NullInjectorError: No provider for HttpClient

Arborescence

```
src
└── app
    ├── pages
    │   └── formations
    │       ├── formations.ts
    │       ├── formations.html
    │       └── formations.css
    │
    └── services
        └── formation.service.ts
```

Étape 2 : Le Service API

Nous allons utiliser une API gratuite :

https://jsonplaceholder.typicode.com/users

Elle renvoie une liste d'utilisateurs.

formation.service.ts

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

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

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

  private http = inject(HttpClient);

  api = 'https://jsonplaceholder.typicode.com/users';

  getFormations() {

    return this.http.get<any[]>(this.api);

  }

}
```

On injecte HttpClient :

```
private http = inject(HttpClient);
```

C'est lui qui sait envoyer des requêtes HTTP.

Puis :

```
api =
'https://jsonplaceholder.typicode.com/users';
```

C'est l'adresse du serveur.

Lorsque le composant appelle :

```
service.getFormations()
```

Le service exécute :

```
this.http.get(...)
```

Angular envoie une requête HTTP :

GET

https://jsonplaceholder.typicode.com/users

Le serveur répond :

```
[
 {
  id:1,
  name:"Leanne Graham"
 },

 {
  id:2,
  name:"Ervin Howell"
 }

...
]
```

Le service renvoie ces données au composant.

Étape 3 : Le composant

formations.ts

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

import { FormationService } from '../../services/formation.service';

@Component({

 selector:'app-formations',

 standalone:true,

 imports:[],

 templateUrl:'./formations.html',

 styleUrl:'./formations.css'

})

export class Formations{

 service = inject(FormationService);
```

```
 utilisateurs:any[]=[];
 ngOnInit(){
  this.service.getFormations()
    .subscribe(
     data=>{
       this.utilisateurs=data;
     }
    );
 }
}
```

Étape 4 : HTML

```
<div class="container">
<h1>CFITECH</h1>
<h2>Liste des utilisateurs</h2>
@for(u of utilisateurs; track u.id){
<div class="card">
<h3>{{u.name}}</h3>
<p>{{u.email}}</p>
<p>{{u.phone}}</p>
</div>
}
</div>
```

CSS

```
.container{

width:700px;

margin:40px auto;

}

.card{

background:white;

padding:20px;

margin-top:15px;

border-radius:8px;

box-shadow:0 0 8px rgba(0,0,0,.2);

}

h1{

text-align:center;

color:#1565c0;

}
```

Que peut faire un Service API ?

En entreprise, un Service API ne sert pas uniquement à récupérer des données. Il permet d'effectuer toutes les opérations CRUD :

Méthode HTTP	Action	Exemple
GET	Lire des données	Récupérer la liste des formations
POST	Ajouter une donnée	Ajouter un nouveau stagiaire
PUT	Modifier une donnée	Modifier les informations d'une formation
DELETE	Supprimer une donnée	Supprimer une inscription

Exemples :

```
// Lire les données
this.http.get(url);

// Ajouter une donnée
this.http.post(url, formation);

// Modifier une donnée
this.http.put(url + '/' + id, formation);

// Supprimer une donnée
this.http.delete(url + '/' + id);
```

Le **Service API** est responsable de toute la communication entre Angular et le serveur. Les composants ne connaissent ni l'adresse de l'API ni la manière d'envoyer les requêtes HTTP : ils demandent simplement au service de récupérer, ajouter, modifier ou supprimer des données. Le service utilise HttpClient pour effectuer les appels HTTP (GET, POST, PUT, DELETE) et renvoie les résultats au composant sous forme d'Observable. Cette séparation des responsabilités rend le code plus propre, plus réutilisable et plus facile à maintenir. C'est une architecture utilisée dans toutes les applications Angular professionnelles.

13. Service de stockage local (Local Storage)

Qu'est-ce que le Local Storage ?

Le **Local Storage** est un espace de stockage fourni par le navigateur.

Il permet de conserver des données **même après avoir fermé ou rechargé la page**.

Contrairement à une variable Angular, les données ne disparaissent pas lorsque l'utilisateur actualise le navigateur.

Pourquoi utiliser un Service de stockage local ?

Imaginons que le stagiaire ajoute plusieurs formations dans son panier.

Sans Local Storage :

Angular
React
Node.js

Tout fonctionne...

Mais si l'utilisateur appuie sur **F5** ou ferme le navigateur :

Le panier est vide.

Toutes les données sont perdues.

Avec le Local Storage :

Les données sont sauvegardées automatiquement.

Même après avoir fermé le navigateur, elles sont toujours présentes.

Exercice :

Le directeur de **CFITECH** souhaite que les stagiaires puissent enregistrer leurs formations préférées.

Même après avoir quitté l'application, leurs choix doivent être conservés.

Le directeur demande donc de créer un **StorageService**.

Étape 1 : Le service

storage.service.ts

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

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

  enregistrer(cle: string, valeur: any) {

    localStorage.setItem(

      cle,

      JSON.stringify(valeur)

    );

  }

  recuperer(cle: string) {

    const valeur = localStorage.getItem(cle);

    return valeur ? JSON.parse(valeur) : [];
```

```
 }

 supprimer(cle: string) {

   localStorage.removeItem(cle);

 }

}
```

Comprendre le service

Enregistrer une donnée

```
localStorage.setItem(
   cle,
   JSON.stringify(valeur)
);
```

Cette méthode enregistre une donnée dans le navigateur.

Par exemple :

```
enregistrer(
'favoris',
['Angular','React']
);
```

Le navigateur sauvegarde :

favoris

```
[
 Angular,
 React
]
```

Même après fermeture du navigateur.

Pourquoi JSON.stringify() ?

Le Local Storage ne sait enregistrer **que du texte**.

Un tableau comme :

[
'Angular',
'React'
]

doit être transformé en chaîne de caractères.

C'est exactement le rôle de :

JSON.stringify()

Le tableau devient :

["Angular","React"]

Cette chaîne peut alors être stockée.

Lire une donnée

localStorage.getItem(cle);

Cette méthode récupère la valeur enregistrée.

Mais elle renvoie une chaîne de caractères.

Exemple :

["Angular","React"]

Pourquoi JSON.parse() ?

Pour retrouver un vrai tableau Angular.

JSON.parse(valeur)

transforme

["Angular","React"]

en

[
'Angular',
'React'
]

Étape 2 : Le composant

favoris.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { StorageService } from '../../services/storage.service';

@Component({
 selector: 'app-favoris',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './favoris.html',
 styleUrl: './favoris.css'
})
export class Favoris {

 service = inject(StorageService);

 formation = '';

 favoris: string[] = [];

 ngOnInit() {

  this.favoris = this.service.recuperer('favoris');

 }

 ajouter() {

  if(this.formation.trim()){

   this.favoris.push(this.formation);

   this.service.enregistrer(

    'favoris',

    this.favoris

   );

   this.formation='';

  }

 }

 supprimer(i:number){

  this.favoris.splice(i,1);
```

```
  this.service.enregistrer(
    'favoris',
    this.favoris
  );
 }
}
```

Étape 3 : Le HTML

```
<div class="container">
<h1>CFITECH</h1>
<h2>Mes formations favorites</h2>
<input
[(ngModel)]="formation"
placeholder="Nouvelle formation">
<button
(click)="ajouter()">
Ajouter
</button>
<hr>
@for(f of favoris; track f; let i=$index){
<div class="card">
{{f}}
<button
class="supprimer"
(click)="supprimer(i)">
Supprimer
```

</button>

}

Étape 4 : Le CSS

```
body{

background:#eef2f7;

}

.container{

width:650px;

margin:40px auto;

background:white;

padding:30px;

border-radius:10px;

box-shadow:0 0 12px rgba(0,0,0,.15);

}

h1{

text-align:center;

color:#1565c0;

}

input{

width:100%;

padding:12px;

margin:15px 0;

border:1px solid #ccc;
```

```
border-radius:6px;
}
button{
padding:10px 18px;
background:#1565c0;
color:white;
border:none;
border-radius:6px;
cursor:pointer;
}
.card{
display:flex;
justify-content:space-between;
align-items:center;
margin-top:12px;
padding:15px;
background:#f4f6f9;
border-radius:8px;
}
.supprimer{
background:#d32f2f;
}
```

14. Service de calcul

Qu'est-ce qu'un Service de calcul ?

Un **Service de calcul** est un service qui effectue tous les calculs de l'application.

Au lieu d'écrire les formules dans les composants, on les place dans un service.

Les composants demandent simplement :

"Calcule ce montant."

Le service effectue le calcul puis renvoie le résultat.

Ex : Imaginons une application de gestion des formations.

Chaque inscription coûte :

- Angular : **350 €**
- React : **300 €**
- Node.js : **280 €**

Une remise de **10 %** est accordée aux étudiants.

Sans service, chaque composant devrait refaire le calcul.

Si demain la remise passe à **15 %**, il faudrait modifier plusieurs composants.

Avec un **Service de calcul**, une seule modification suffit.

Exercice :

Le centre **CFITECH** souhaite créer une page d'inscription.

Règle métier :

- Le prix d'une formation est saisi.
- Une remise de **10 %** est appliquée automatiquement.
- Le montant final est affiché.

Le directeur souhaite que **le calcul soit entièrement réalisé par un service**.

Étape 1 : Le service

calcul.service.ts

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

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

  calculerPrixFinal(prix: number): number {

    const remise = prix * 0.10;

    return prix - remise;

  }

}
```

Étape 2 : Le composant

inscription.ts

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CalculService } from '../../services/calcul.service';

@Component({
  selector: 'app-inscription',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './inscription.html',
  styleUrl: './inscription.css'
})
export class Inscription {

  service = inject(CalculService);

  prix = 0;

  resultat = 0;

  calculer() {

    this.resultat = this.service.calculerPrixFinal(this.prix);

  }
```

```
}
```

Étape 3 : Le HTML

```
<div class="container">

  <h1>CFITECH</h1>

  <h2>Calcul du prix final</h2>

  <input
    type="number"
    [(ngModel)]="prix"
    placeholder="Prix de la formation">

  <button (click)="calculer()">

    Calculer

  </button>

  <div class="resultat">

    Prix final après remise :

    <strong>

      {{ resultat }} €

    </strong>

  </div>

</div>
```

Étape 4 : Le CSS

```
body{

  background:#eef2f7;

}

.container{

  width:500px;

  margin:40px auto;
```

```
    background:white;
    padding:30px;
    border-radius:10px;
    box-shadow:0 0 10px rgba(0,0,0,.2);
}
h1{
    text-align:center;
    color:#1565c0;
}
input{
    width:100%;
    padding:12px;
    margin:20px 0;
    border:1px solid #ccc;
    border-radius:6px;
}
button{
    width:100%;
    padding:12px;
    background:#1565c0;
    color:white;
    border:none;
    border-radius:6px;
    cursor:pointer;
}
.resultat{
```

```
  margin-top:25px;

  padding:20px;

  background:#d4edda;

  border-left:5px solid green;

  border-radius:6px;

  font-size:20px;

  text-align:center;

}
```

15. Service avec Signals

Qu'est-ce qu'un Service avec Signals ?

Un **Service avec Signals** est un service qui stocke des données dans des **Signals** afin que tous les composants soient automatiquement mis à jour lorsqu'une valeur change.

Avec les Signals, Angular détecte immédiatement les modifications et met à jour l'interface sans qu'il soit nécessaire d'utiliser subscribe() ou des événements complexes.

Ex : Imaginons une application de gestion des inscriptions.

Plusieurs composants doivent connaître le nombre de stagiaires inscrits :

- le Dashboard ;
- la barre de navigation ;
- la page Statistiques ;
- le Footer.

Sans Signal, il faudrait envoyer les données entre les composants.

Avec un Signal, dès que la valeur change, **tous les composants sont automatiquement mis à jour**.

Exercice :

Le directeur de **CFITECH** souhaite afficher en permanence le nombre de stagiaires inscrits.

Chaque fois qu'un stagiaire est ajouté, le compteur doit être mis à jour automatiquement sur toutes les pages.

Le directeur demande donc de créer un **Service utilisant les Signals**.

Étape 1 : Le service

compteur.service.ts

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

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

  compteur = signal(0);

  incrementer(){

    this.compteur.update(

      valeur => valeur + 1

    );

  }

  decrementer(){

    this.compteur.update(

      valeur => valeur - 1

    );

  }

  reinitialiser(){

    this.compteur.set(0);

  }

}
```

Étape 2 : Le composant

compteur.ts

```
import { Component, inject } from '@angular/core';
import { CompteurService } from '../../services/compteur.service';

@Component({
```

```
  selector: 'app-compteur',
  standalone: true,
  imports: [],
  templateUrl: './compteur.html',
  styleUrl: './compteur.css'
})
export class Compteur {

  service = inject(CompteurService);

}
```

Étape 3 : Le HTML

```
<div class="container">

   <h1>CFITECH</h1>

   <h2>Compteur des inscriptions</h2>

   <div class="compteur">

      {{ service.compteur() }}

   </div>

   <button

      (click)="service.incrementer()">

      Ajouter un stagiaire

   </button>

   <button

      class="moins"

      (click)="service.decrementer()">

      Retirer un stagiaire

   </button>

   <button

      class="reset"

      (click)="service.reinitialiser()">
```

```
    Réinitialiser
  </button>
</div>
```

Étape 4 : Le CSS

```
body{
  background:#eef2f7;
}
.container{
  width:500px;
  margin:40px auto;
  background:white;
  padding:30px;
  border-radius:10px;
  box-shadow:0 0 12px rgba(0,0,0,.2);
  text-align:center;
}
h1{
  color:#1565c0;
}
.compteur{
  width:150px;
  height:150px;
  margin:25px auto;
  border-radius:50%;
  background:#1565c0;
```

```
    color:white;
    display:flex;
    justify-content:center;
    align-items:center;
    font-size:50px;
    font-weight:bold;
}
button{
    width:100%;
    padding:12px;
    margin-top:12px;
    border:none;
    border-radius:6px;
    background:#1565c0;
    color:white;
    cursor:pointer;
}
.moins{
    background:#ef6c00;
}
.reset{
    background:#c62828;
}
```