Module 1 : Les formulaires

Introduction

Dans la plupart des applications web, les utilisateurs doivent saisir des informations : créer un compte, se connecter, envoyer un message, passer une commande ou encore s'inscrire à une formation. Toutes ces actions se réalisent à l'aide de **formulaires**.

Angular propose des outils puissants permettant de créer des formulaires interactifs, de récupérer les informations saisies par l'utilisateur et de vérifier automatiquement qu'elles sont correctes avant leur envoi.

Dans ce point, nous allons apprendre à créer des formulaires professionnels en utilisant les deux approches proposées par Angular : **Template Driven Forms** et **Reactive Forms**.

Les **Template Driven Forms** utilisent principalement :

- FormsModule
- ngModel
- les validations directement dans le HTML

1. Importer FormsModule

Dans Angular 21 (Standalone Component), on importe directement FormsModule.

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

@Component({
 selector: 'app-user',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './user.component.html'
})
export class UserComponent {

}
```

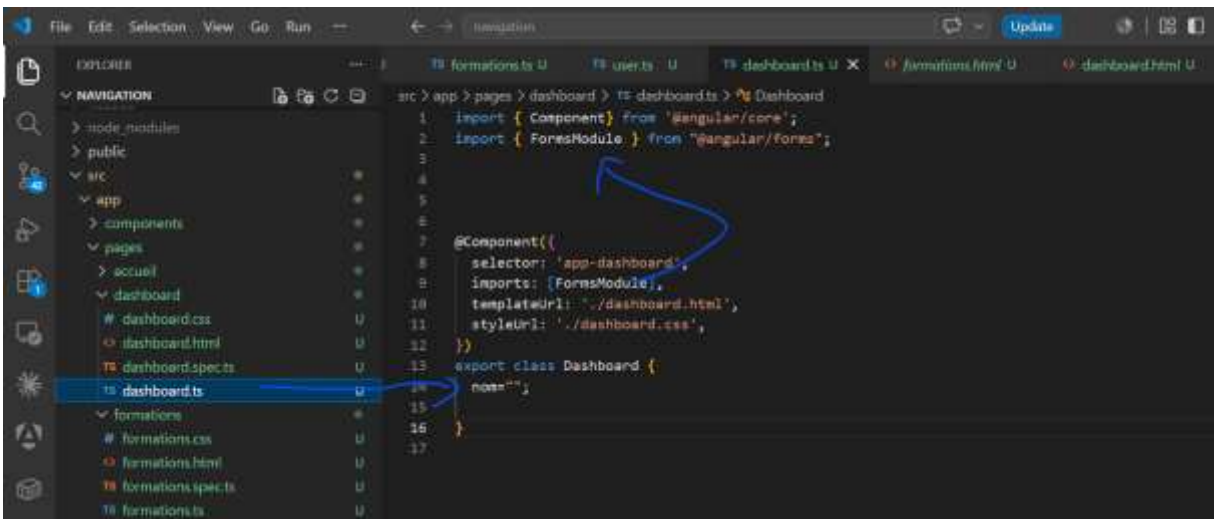
2. Premier exemple avec ngModel

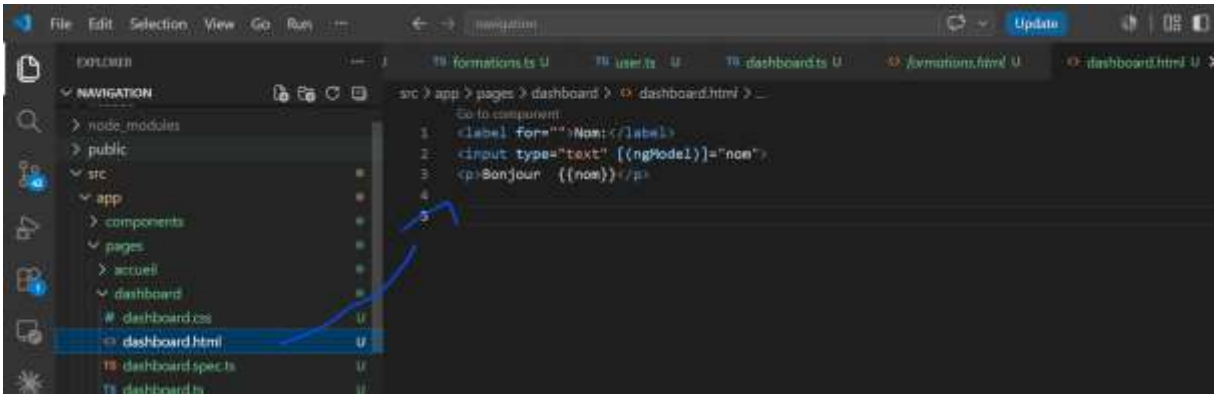
user.component.ts

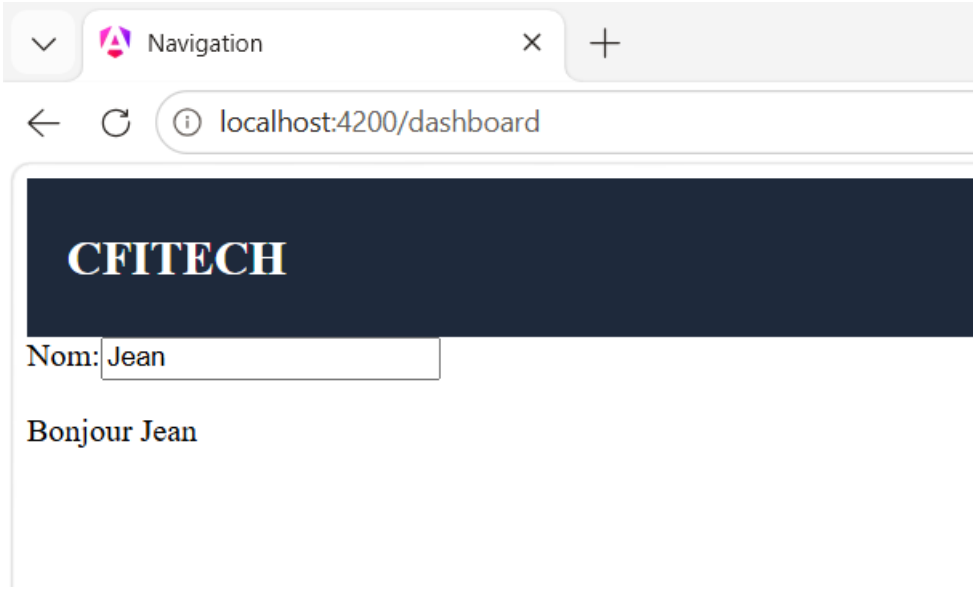
```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
 selector: 'app-user',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './user.component.html'
})
export class UserComponent {

 nom = "";

}
```

user.component.html

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

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







Le [(ngModel)] fait la liaison dans les deux sens :

HTML <------> TypeScript

3. Exemple de formulaire

user.component.ts

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

@Component({
 selector: 'app-user',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './user.component.html'
})
```

```
export class UserComponent {

 utilisateur = {
  nom: '',
  email: ''
 };

 enregistrer() {
  console.log(this.utilisateur);
 }

}
```

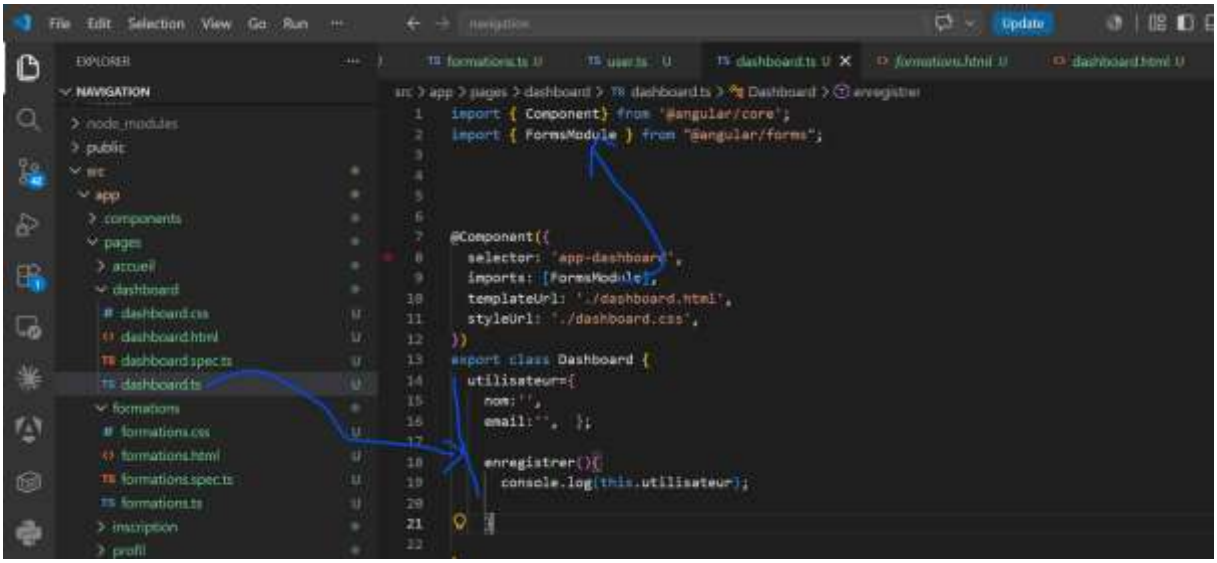
HTML

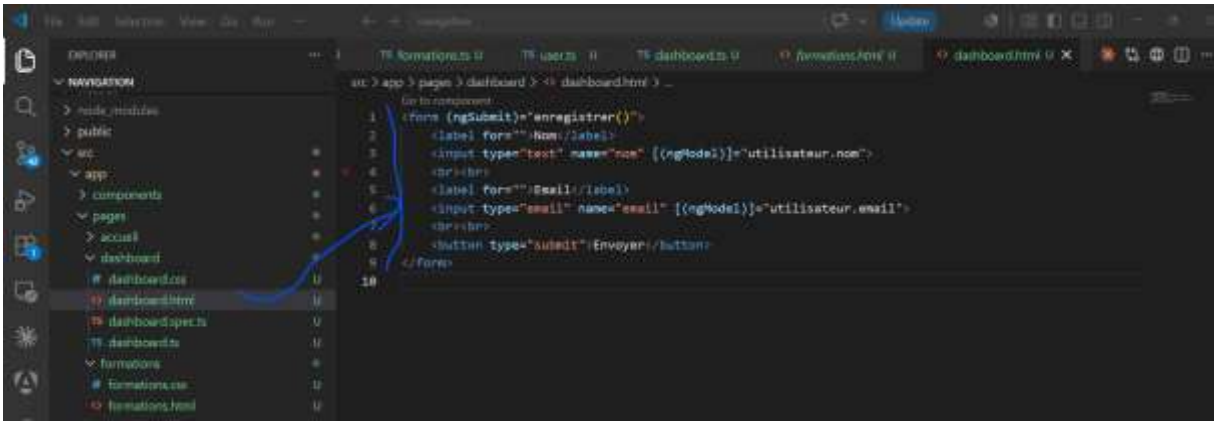
```
<form (ngSubmit)="enregistrer()">

  <label>Nom</label>

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

  <br><br>

  <label>Email</label>

  <input
    type="email"
    name="email"
    [(ngModel)]="utilisateur.email">
```

```
  <br><br>

  <button type="submit">
    Enregistrer
  </button>

</form>
```

Lorsque tu cliques sur **Enregistrer**, la console affiche

```
{
 nom: "Ali",
 email: "ali@gmail.com"
}
```





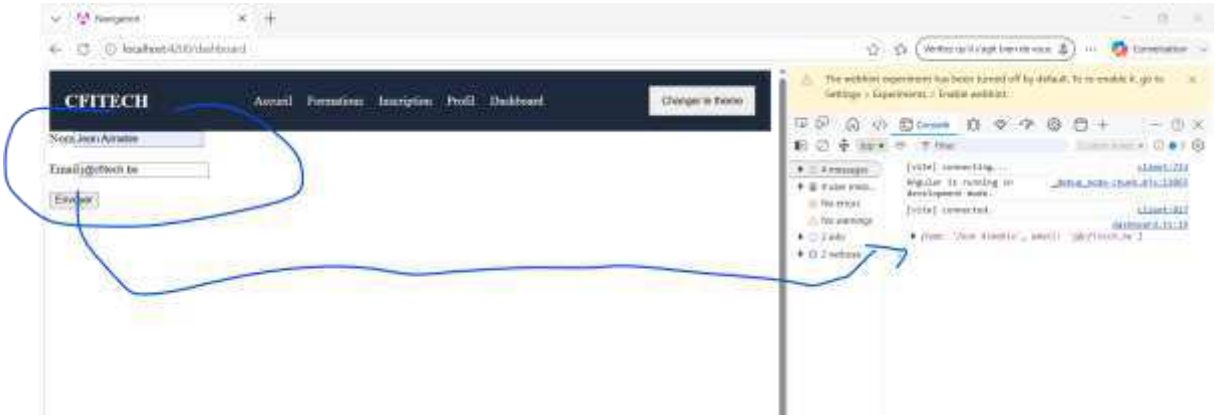
ngSubmit

ngSubmit est un événement spécial des formulaires Angular.

<form (ngSubmit)="enregistrer()">

Ça signifie que ：

Lorsque le formulaire est soumis, Angular appelle enregistrer().



4. Validation Required

HTML

<form #f="ngForm">

<input

type="text"

name="nom"

required

[(ngModel)]="utilisateur.nom"

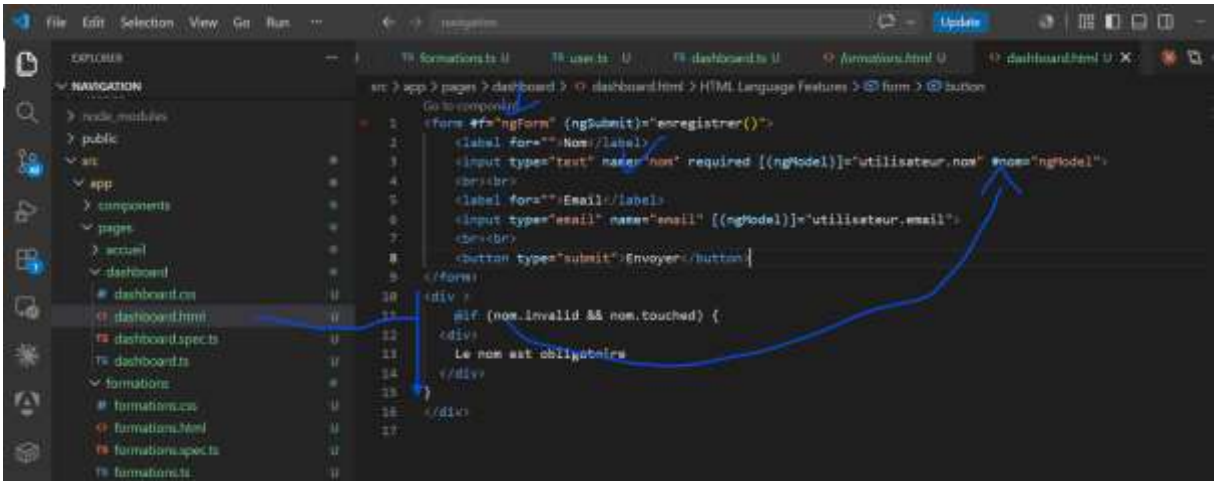
#nom="ngModel">

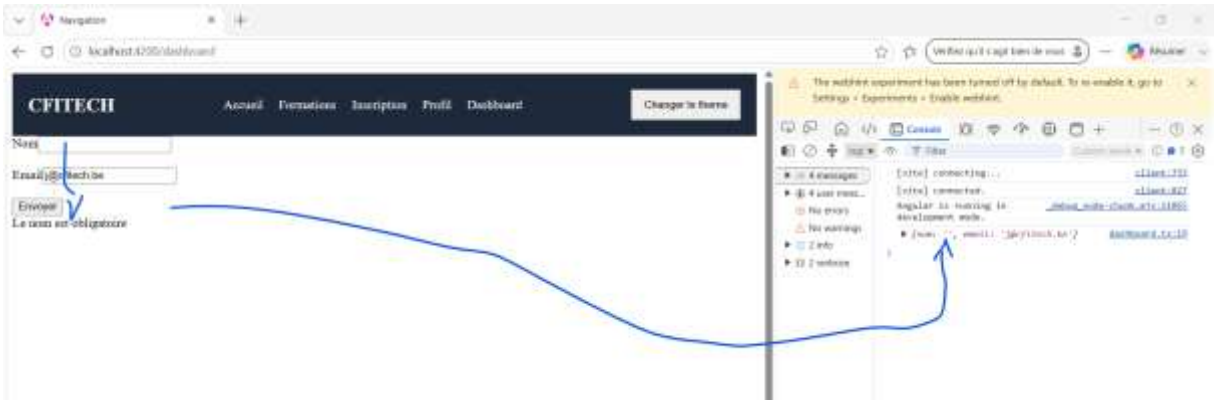
</form>

Afficher l'erreur

<div @if(nom.invalid && nom.touched)>

Le nom est obligatoire





#f="ngForm"

<form #f="ngForm">

ngForm est une directive Angular qui transforme une simple balise <form> HTML en un **formulaire Angular**. La syntaxe #f="ngForm" crée une **référence locale** appelée f vers ce formulaire. Grâce à cette référence, tu peux connaître l'état global du formulaire : est-il valide (f.valid) ? invalide (f.invalid) ? a-t-il été soumis (f.submitted) ? Par exemple, on utilise souvent f.invalid pour désactiver le bouton **Envoyer** tant que tous les champs ne sont pas correctement remplis.

Exemple :

```
<form #f="ngForm">
  ...
  <button [disabled]="f.invalid">Envoyer</button>
</form>
```

required

<input required>

required est une validation HTML reconnue par Angular. Elle indique que ce champ est **obligatoire**. Si l'utilisateur laisse le champ vide, Angular considère que le contrôle est **invalide**. Cette validation est ensuite accessible via ngModel pour afficher un message d'erreur ou empêcher l'envoi du formulaire.

Exemple :

```
<input
  name="nom"
  required
  [(ngModel)]="utilisateur.nom">
```

Ici, le champ **Nom** doit obligatoirement être rempli.

name="nom"

<input name="nom">

L'attribut name sert à **identifier le champ dans le formulaire Angular**. Dans un **Template Driven Form**, chaque champ utilisant [(ngModel)] doit obligatoirement avoir un name. Sans cet attribut, Angular ne peut pas enregistrer le champ dans le formulaire et affiche l'erreur **NG01352**. Ce nom doit être unique à l'intérieur du formulaire.

Exemple :

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

Ici, Angular connaît ce contrôle sous le nom **nom**.

#nom="ngModel"

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

Cette syntaxe crée une **référence locale** appelée nom vers la directive NgModel du champ. Grâce à cette référence, tu peux consulter toutes les informations concernant ce contrôle : est-il valide (nom.valid) ? invalide (nom.invalid) ? l'utilisateur l'a-t-il déjà touché (nom.touched) ? Quelles sont les erreurs (nom.errors) ? C'est cette référence qui permet d'afficher des messages d'erreur adaptés.

Exemple 1:

```
<input
  type="text"
  name="nom"
  required
  minlength="3"
  [(ngModel)]="utilisateur.nom"
  #nom="ngModel">

@if (nom.errors?.['required'] && nom.touched) {
  <p>Le nom est obligatoire.</p>
}

@if (nom.errors?.['minlength'] && nom.touched) {
  <p>Minimum 3 caractères.</p>
}
```

Exemple 2 :

```
<input
  name="nom"
  required
  [(ngModel)]="utilisateur.nom"
  #nom="ngModel">
```

@if (nom.invalid && nom.touched)

```
@if (nom.invalid && nom.touched) {
  <p>Le nom est obligatoire.</p>
}
```

Cette condition permet d'afficher un message **uniquement lorsque cela est nécessaire**. nom.invalid signifie que le champ ne respecte pas les règles de validation (par exemple required). nom.touched signifie que l'utilisateur a déjà cliqué dans le champ puis l'a quitté. En combinant les deux avec &&, le message d'erreur n'apparaît que lorsque le champ est invalide **et** que l'utilisateur a déjà essayé de le remplir. Cela évite d'afficher des erreurs dès le chargement de la page.

```
<form #f="ngForm">

  <input
    type="text"
    name="nom"
    required
    [(ngModel)]="utilisateur.nom"
    #nom="ngModel">

  @if (nom.invalid && nom.touched) {
    <p>Le nom est obligatoire.</p>
  }

  <button [disabled]="f.invalid">
    Envoyer
  </button>

</form>
```

C'est la combinaison de ces éléments qui permet à Angular de gérer automatiquement la validation d'un formulaire Template Driven.

5. Validation Email

```
<input
type="email"
name="email"
email
required
[(ngModel)]="utilisateur.email"
#emailCtrl="ngModel">
```

Erreur

```
<div @if="emailCtrl.errors?.['required'] && emailCtrl.touched">
  Email obligatoire
</div>

<div @if="emailCtrl.errors?.['email']">
  Email invalide
</div>
```

6. Validation MinLength

```
<input
type="password"
name="password"
required
minlength="6"
[(ngModel)]="password"
#pwd="ngModel">
Message
<div *nglf="pwd.errors?.['minlength']">

Le mot de passe doit contenir au moins 6 caractères.

</div>
```

7. Désactiver le bouton tant que le formulaire est invalide

```
<form #f="ngForm">

  <input
    name="nom"
    required
    [(ngModel)]="utilisateur.nom">
```

```
  <input
    name="email"
    email
    required
    [(ngModel)]="utilisateur.email">

  <button
    [disabled]="f.invalid">
    Enregistrer
  </button>

</form>
```

Le bouton devient actif uniquement lorsque :

- le nom est rempli ;
- l'email est valide.

8. Exemple complet

user.component.ts

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

@Component({
 selector: 'app-user',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './user.component.html'
})
export class UserComponent {

 utilisateur = {
```

```
  nom: '',
  email: '',
  age: null
 };

 enregistrer() {
  console.log(this.utilisateur);
 }

}
```

user.component.html

```
<form #f="ngForm" (ngSubmit)="enregistrer()">

  <label>Nom</label>

  <input
    name="nom"
    required
    [(ngModel)]="utilisateur.nom"
    #nom="ngModel">

  <div @if="nom.invalid && nom.touched">
    Nom obligatoire
  </div>

  <br>

  <label>Email</label>

  <input
    type="email"
```

```
  name="email"
  required
  email
  [(ngModel)]="utilisateur.email"
  #email="ngModel">

<div @if="email.errors?.['required'] && email.touched">
  Email obligatoire
</div>

<div @if="email.errors?.['email']">
  Email incorrect
</div>

<br>

<label>Age</label>

<input
  type="number"
  name="age"
  [(ngModel)]="utilisateur.age">

<br><br>

<button
  type="submit"
  [disabled]="f.invalid">
  Enregistrer
</button>
```

</form>

Les **Template Driven Forms** sont simples et rapides pour les petits formulaires. Pour des formulaires plus complexes (validation avancée, formulaires dynamiques, formulaires imbriqués), Angular recommande généralement les **Reactive Forms**.

Template Driven Forms → la logique est surtout dans le **HTML**.

Reactive Forms → la logique est surtout dans le **TypeScript**.

Reactive Forms :

les **Reactive Forms** sont la méthode recommandée pour les formulaires complexes.

Dans les Reactive Forms, Angular met à notre disposition plusieurs classes importantes : **FormControl**, **FormGroup** et **FormBuilder**. Elles servent à créer et à gérer les champs et les formulaires de notre application.

Nous allons voir chaque notion avec un exemple pratique.

1. Importer ReactiveFormsModule

Dans un composant standalone :

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

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


@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [ReactiveFormsModule],
 templateUrl: './dashboard.html'
})
export class DashboardComponent {


}
```

2. FormControl

Un **FormControl** représente **un seul champ**.

Exemple :

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

nom = new FormControl('');

Ici on crée un champ appelé **nom**.

HTML

<input [formControl]="nom">

{{ nom.value }}

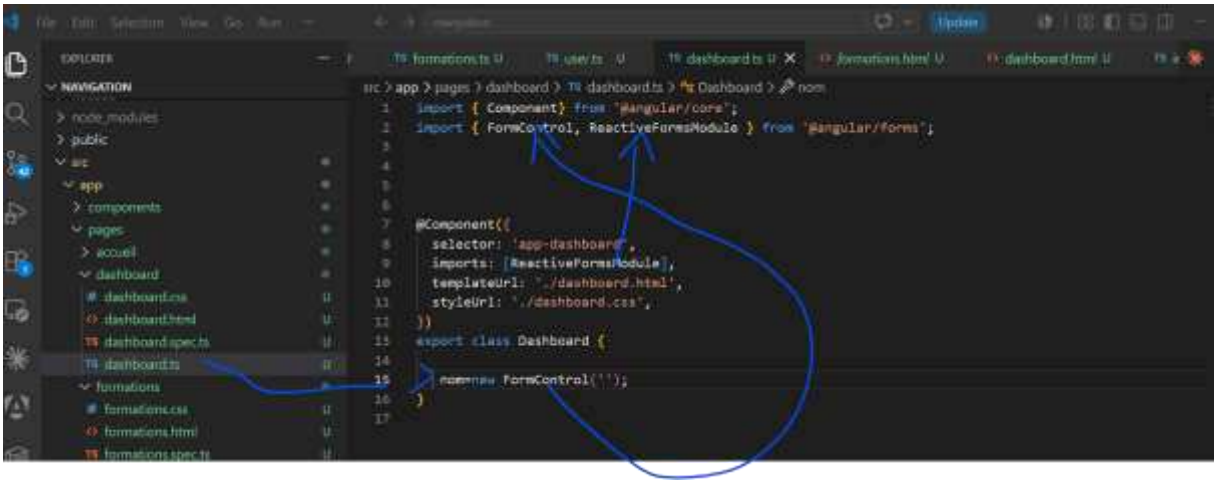
Si l'utilisateur tape :

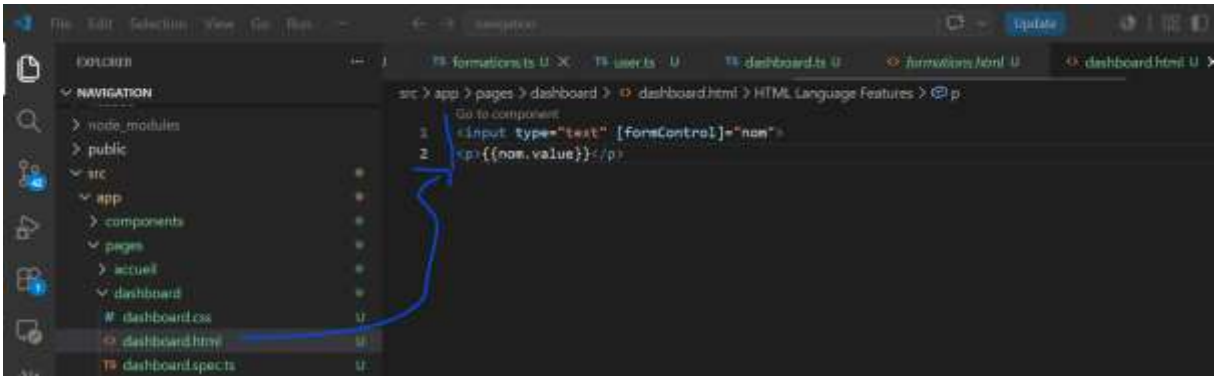
Ahmed

Angular affiche :

Ahmed

Comme avec ngModel, mais cette fois le contrôle est créé en TypeScript.





Le **FormControl** est la **brique de base** des Reactive Forms.

Il représente un seul champ du formulaire.

Par exemple, si tu as un formulaire avec :

- Nom
- Email
- Âge

Alors tu auras **3 FormControl** :

Formulaire

│

├── FormControl (Nom)

├── FormControl (Email)

└── FormControl (Âge)

Exemple 1 : un seul champ

TypeScript

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


nom = new FormControl('');
```

Ici, tu crées un contrôle appelé nom.

Sa valeur initiale est une chaîne vide ('').

HTML

```
<input type="text" [formControl]="nom">
```

{{ nom.value }}

Si l'utilisateur tape :

Ahmed

Alors :

nom.value

vaut :

Ahmed

Le FormControl **stocke la valeur** du champ.

Que contient un FormControl ?

Imaginons :

nom = new FormControl('');

Le FormControl ne contient pas seulement la valeur.

Il connaît aussi :

- la valeur (value)
- si le champ est valide (valid)
- si le champ est invalide (invalid)
- les erreurs (errors)
- si l'utilisateur a touché le champ (touched)

Par exemple :

console.log(nom.value);

→

Ahmed

console.log(nom.valid);

→

true

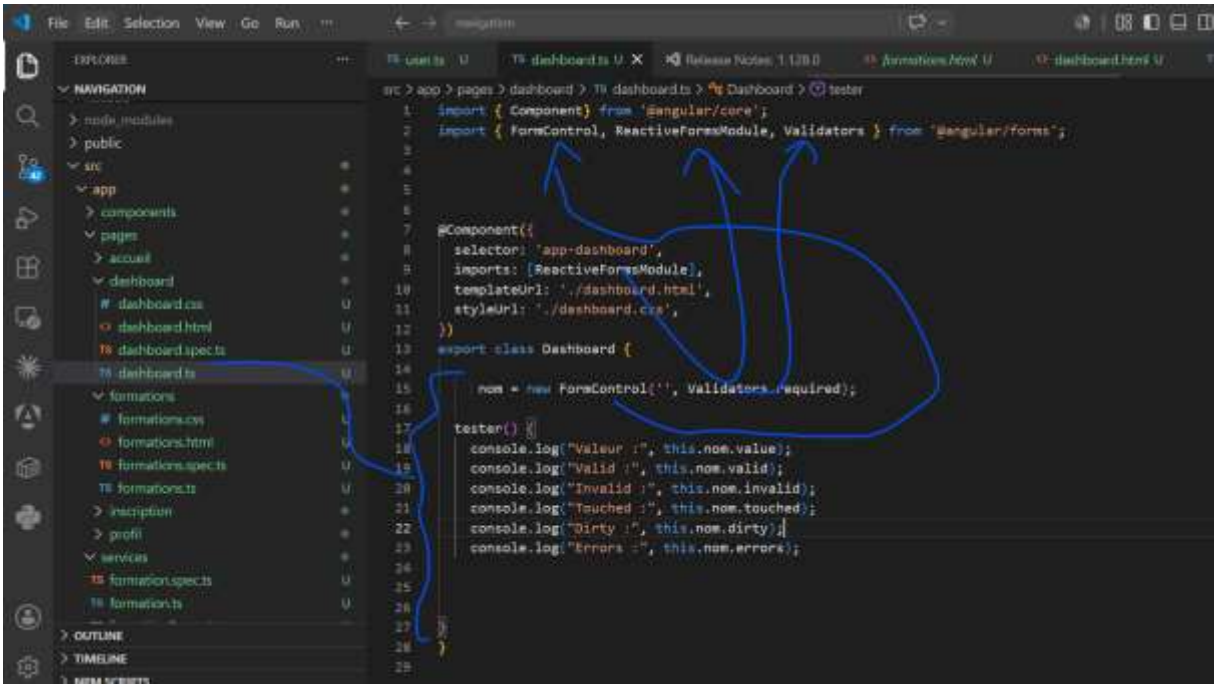
console.log(nom.touched);

→

True

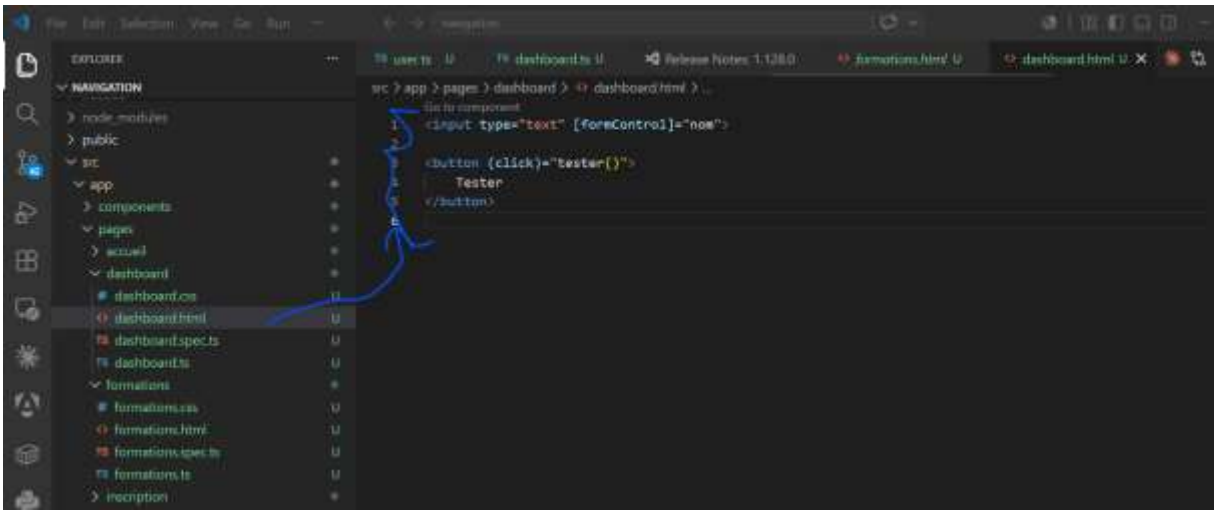
Ex :

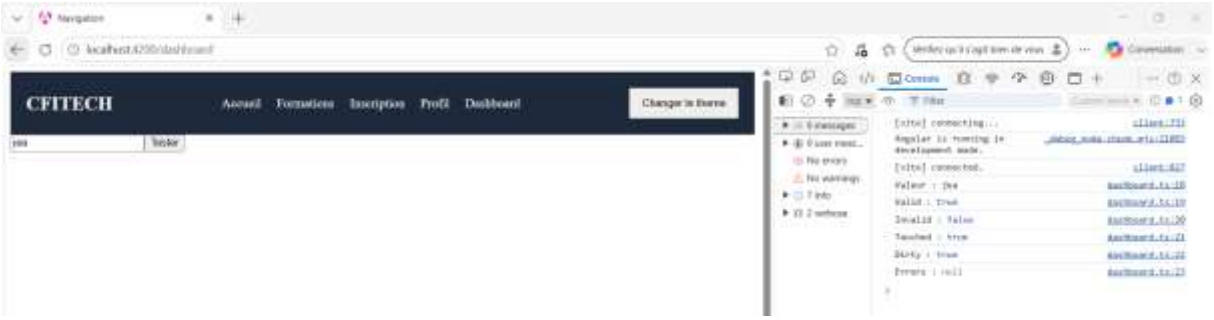
```
import { Component } from '@angular/core';
import {
 ReactiveFormsModule,
 FormControl,
 Validators
} from '@angular/forms';

@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [ReactiveFormsModule],
 templateUrl: './dashboard.html'
})
export class DashboardComponent {

 nom = new FormControl('', Validators.required);

 tester() {
  console.log("Valeur :", this.nom.value);
  console.log("Valid :", this.nom.valid);
  console.log("Invalid :", this.nom.invalid);
  console.log("Touched :", this.nom.touched);
  console.log("Dirty :", this.nom.dirty);
  console.log("Errors :", this.nom.errors);
 }

}
```



Html :

```
<input type="text" [formControl]="nom">


<button (click)="tester()">
   Tester
</button>
```





dirty signifie :

"Est-ce que l'utilisateur a modifié la valeur du champ ?"

C'est un booléen :

- true → la valeur a été modifiée.
- false → la valeur n'a jamais été modifiée.

Les **Validators** servent à **vérifier que les données saisies par l'utilisateur respectent certaines règles**.

Au lieu d'écrire les validations dans le HTML (comme avec les Template Driven Forms), on les écrit dans le **TypeScript**.

Exemple avec un Validator

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

nom = new FormControl('', Validators.required);

Ici le champ est obligatoire.

Si le champ est vide :

nom.value

↓

""

nom.valid

↓

false

nom.errors

↓

{

  required: true

}

Si l'utilisateur tape :

Ali

Alors :

nom.value

↓

Ali

nom.valid

↓

true

nom.errors

↓

null

3. FormGroup

Un formulaire contient généralement plusieurs champs.

On les regroupe dans un **FormGroup**.

[formControl] est utilisé lorsque tu possèdes directement un objet FormControl.

formControlName est utilisé lorsque le FormControl est **à l'intérieur d'un FormGroup**.

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

utilisateur = new FormGroup({

 nom: new FormControl(''),

 email: new FormControl('')

});
```

Le formulaire possède maintenant :

```
utilisateur
│
├── nom
└── email
```

HTML

```
<form [formGroup]="utilisateur">

  <input
    type="text"
    formControlName="nom">

  <input
    type="email"
```

formControlName="email">

</form>

Ici :

- [formGroup] relie le formulaire HTML au FormGroup.
- formControlName relie un champ HTML à un FormControl.

Afficher les valeurs :

console.log(this.utilisateur.value);

Résultat :

{

nom: "Ali",

email: "ali@gmail.com"

}

Ex concret : formulaire d'inscription

Imagine que tu développes une page d'inscription.

L'utilisateur doit remplir :

- Nom
- Email
- Mot de passe

Visuellement :

Nom : _____

Email : _____

Mot de passe: _____

[S'inscrire]

Ici :

- **Nom** → un FormControl
- **Email** → un FormControl

- **Mot de passe** → un FormControl

L'ensemble constitue **un seul FormGroup**.

On peut le représenter ainsi :

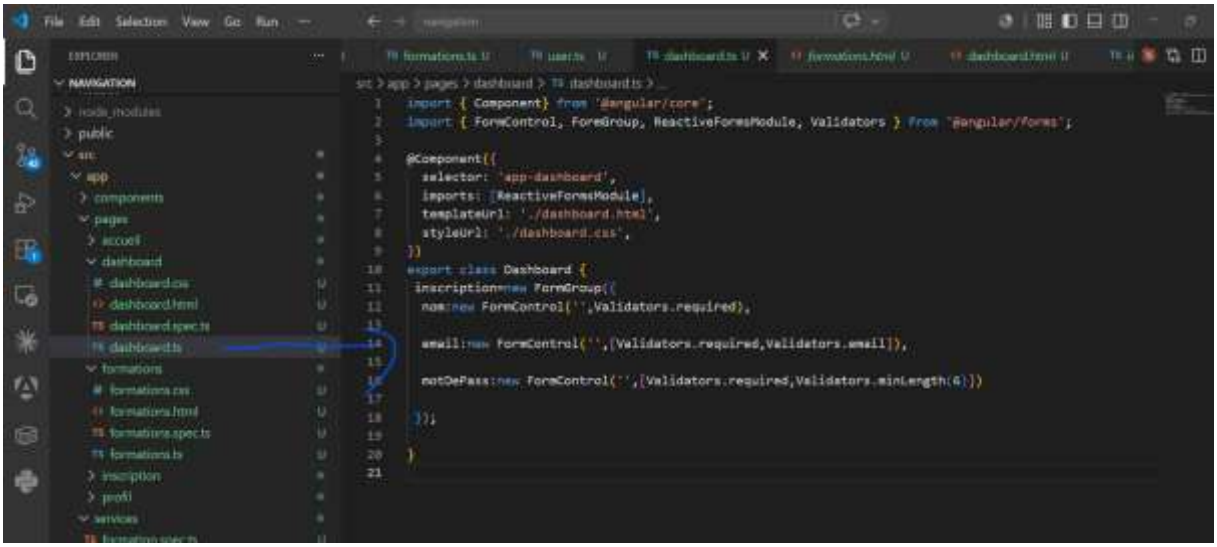
FormGroup (inscription)

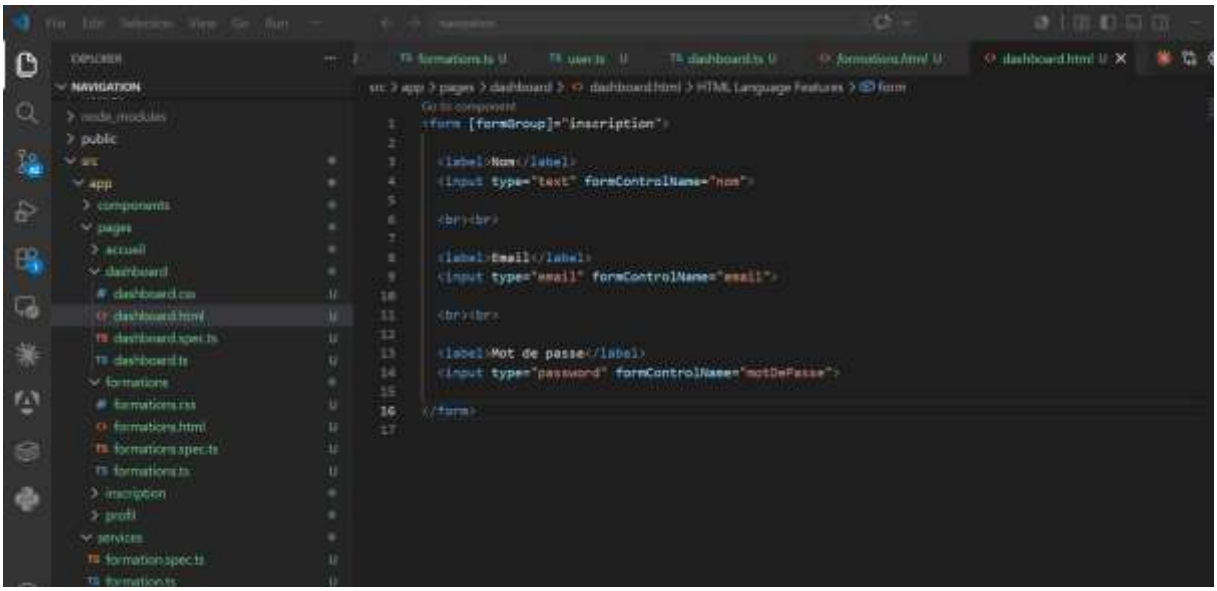
│

├── FormControl (nom)

├── FormControl (email)

└── FormControl (motDePasse)





4. FormBuilder

Créer les contrôles avec new FormControl() fonctionne, mais devient vite répétitif.

Angular fournit **FormBuilder** pour écrire moins de code.

FormBuilder construit le FormGroup et les FormControl à ta place

Sans FormBuilder :

```
utilisateur = new FormGroup({

  nom: new FormControl(''),

  email: new FormControl('')

});
```

Avec FormBuilder :

```
import { FormBuilder } from '@angular/forms';
utilisateur:FormGroup;

  constructor(private fb: FormBuilder){

  this.utilisateur = this.fb.group({

    nom: ['', Validators.required],

    email: ['', [
      Validators.required,
      Validators.email
    ]]

  });
```

C'est exactement le même formulaire.

La plupart des développeurs utilisent **FormBuilder**.

5. Validators

Les validateurs remplacent :

required

email

minlength

qu'on mettait dans le HTML.

Ils sont maintenant dans le TypeScript.

```
import { Validators } from '@angular/forms';

utilisateur = this.fb.group({

  nom: ['', Validators.required],

  email: ['', [Validators.required, Validators.email]]

});
```

Le HTML devient plus simple.

```
<form [formGroup]="utilisateur">

  <input
    type="text"
    formControlName="nom">

  <input
    type="email"
    formControlName="email">

</form>
```

6. Afficher les erreurs

Pour accéder au champ :

this.utilisateur.get('nom')

Dans le HTML :

```
@if (
utilisateur.get('nom')?.invalid &&
utilisateur.get('nom')?.touched
) {
  <p>Nom obligatoire</p>
}
```

Pour l'email :

```
@if (utilisateur.get('email')?.errors?.['email']) {
  <p>Email invalide</p>
}
```

7. Bouton désactivé

Comme avec Template Forms :

```
<button
type="submit"
[disabled]="utilisateur.invalid">
Envoyer
</button>
```

8. Soumettre le formulaire

HTML

```
<form
[formGroup]="utilisateur"
(ngSubmit)="enregistrer()">

...
</form>
```

TS

```
enregistrer(){

  console.log(this.utilisateur.value);

}
```

9. Exemple complet

dashboard.ts

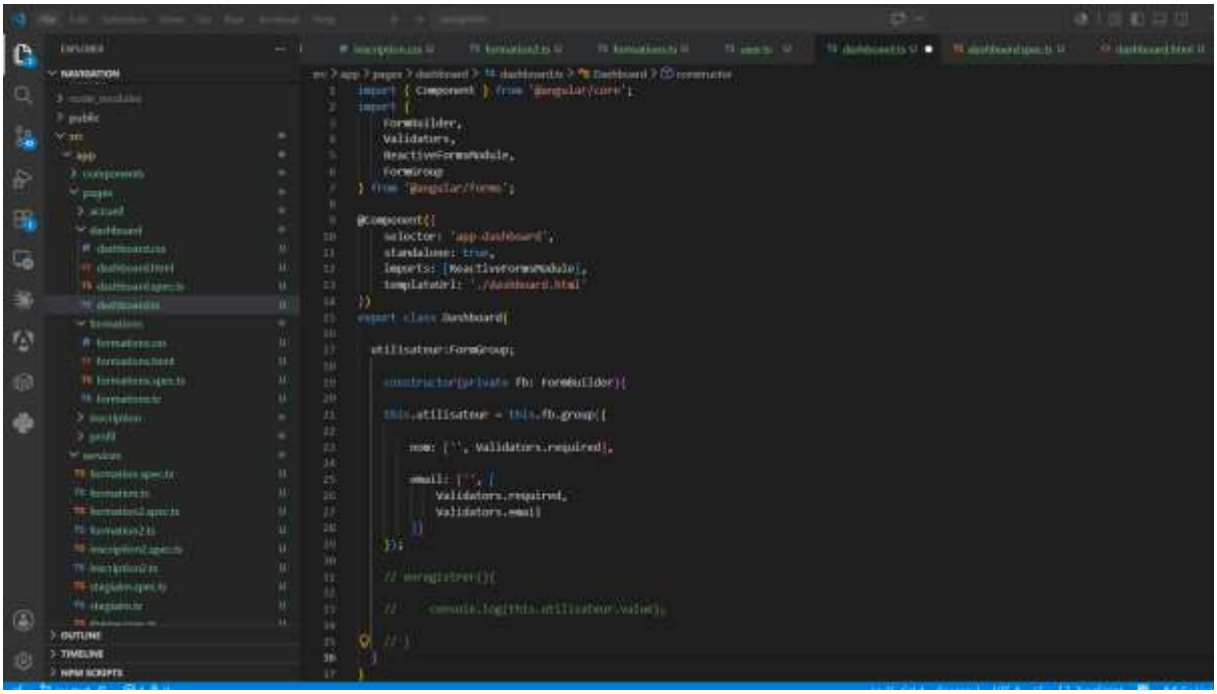
```
import { Component } from '@angular/core';
import {
  FormBuilder,
  Validators,
  ReactiveFormsModule
} from '@angular/forms';

@Component({
  selector: 'app-dashboard',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './dashboard.html'
})
export class DashboardComponent {

  constructor(private fb: FormBuilder){}

  utilisateur = this.fb.group({

    nom: ['', Validators.required],

    email: ['', [
      Validators.required,
```

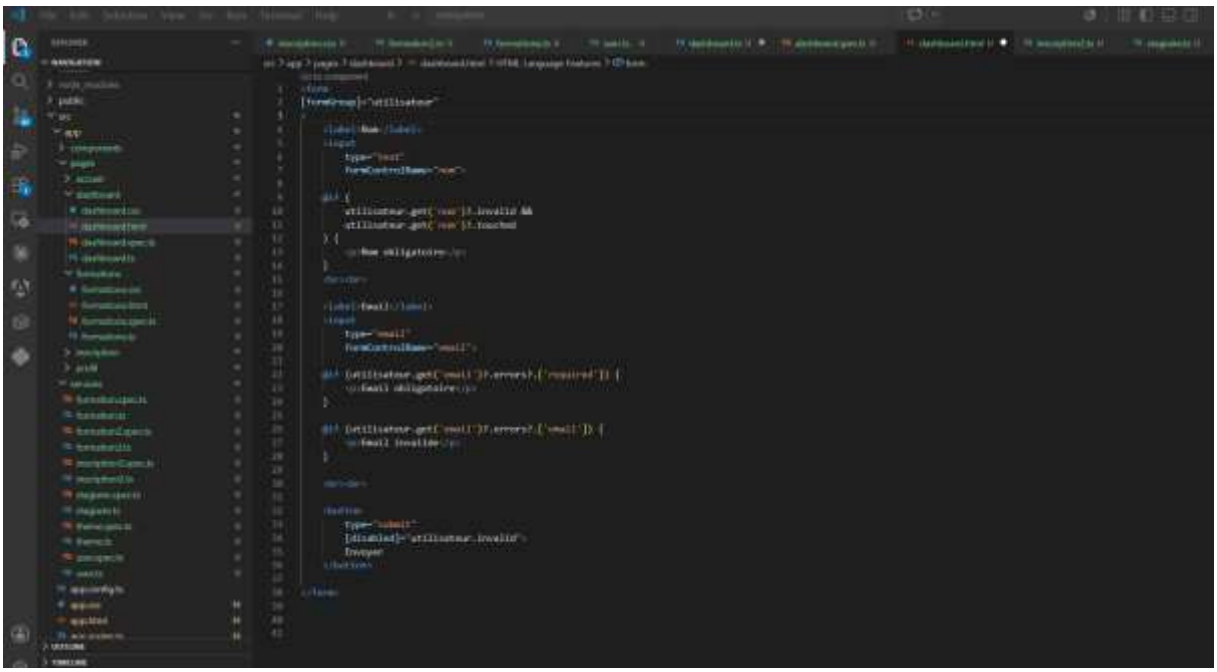
```
      Validators.email
    ]]

  });

  enregistrer(){

    console.log(this.utilisateur.value);

  }

}
```

dashboard.html

```
<form
[formGroup]="utilisateur"
(ngSubmit)="enregistrer()">

  <label>Nom</label>

  <input
    type="text"
    formControlName="nom">

  @if (
    utilisateur.get('nom')?.invalid &&
    utilisateur.get('nom')?.touched
  ) {
    <p>Nom obligatoire</p>
  }

  <br><br>
```

```
  <label>Email</label>

  <input
    type="email"
    formControlName="email">

  @if (utilisateur.get('email')?.errors?.['required']) {
    <p>Email obligatoire</p>
  }

  @if (utilisateur.get('email')?.errors?.['email']) {
    <p>Email invalide</p>
  }

  <br><br>

  <button
    type="submit"
    [disabled]="utilisateur.invalid">
    Envoyer
  </button>

</form>
```





ReactiveFormsModule	Active les Reactive Forms dans le composant.
FormControl	Représente un seul champ (nom, email, âge...).
FormGroup	Regroupe plusieurs FormControl pour former un formulaire.
FormBuilder	Permet de créer un FormGroup plus facilement et avec moins de code.
Validators	Ajoute des règles de validation (required, email, minLength, etc.).

formGroup	Lie le formulaire HTML au FormGroup TypeScript.
formControlName	Relie un champ HTML à un FormControl du FormGroup.
utilisateur.get('nom')	Permet d'accéder au contrôle nom pour lire son état (value, valid, invalid, errors, touched, etc.).

En pratique, pour un projet Angular moderne, on utilise généralement **FormBuilder + Validators**, car c'est la solution la plus concise et la plus facile à maintenir.

Module 2 : Les Pipes Angular

Chapitre 1 : Qu'est-ce qu'un Pipe ?

Un **Pipe** est une fonctionnalité d'Angular qui permet de **transformer l'affichage d'une donnée** dans une page HTML **sans modifier la valeur originale**.

Autrement dit :

Le Pipe modifie uniquement la façon dont la donnée est affichée à l'écran.

La donnée stockée dans le composant reste inchangée.

1. Le Pipe uppercase

Définition

Le **Pipe uppercase** permet de transformer un texte en **lettres majuscules**.

Il modifie uniquement l'affichage du texte dans le HTML. La valeur de la variable reste inchangée dans le composant TypeScript.

Importation

Dans un composant **Standalone**, il faut importer le CommonModule (ou le UpperCasePipe).

Solution recommandée

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

@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [CommonModule],
 templateUrl: './dashboard.html'
})
```

```
export class Dashboard {


  nom = "ahmed";


}
```

Pourquoi importer CommonModule ?

Parce qu'il contient plusieurs Pipes intégrés d'Angular comme :

- uppercase
- lowercase
- currency
- date
- percent
- json
- slice

Si vous oubliez de l'importer, Angular affichera une erreur :

No pipe found with name 'uppercase'

Syntaxe

{{ variable | uppercase }}

Ex 1 : Premier exemple

dashboard.ts

```
nom = "ahmed";
```

dashboard.html

```
<p>Valeur originale : {{ nom }}</p>


<p>Avec uppercase : {{ nom | uppercase }}</p>
```

Rés :

Valeur originale : ahmed

Avec uppercase : AHMED

La variable reste inchangée.

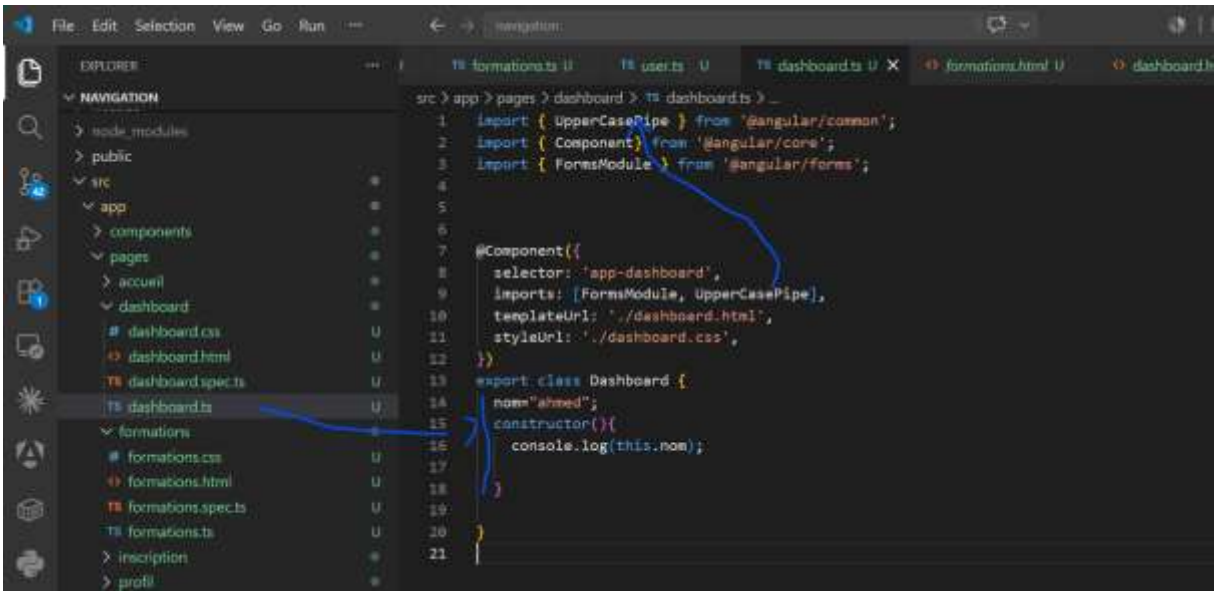
console.log(this.nom);

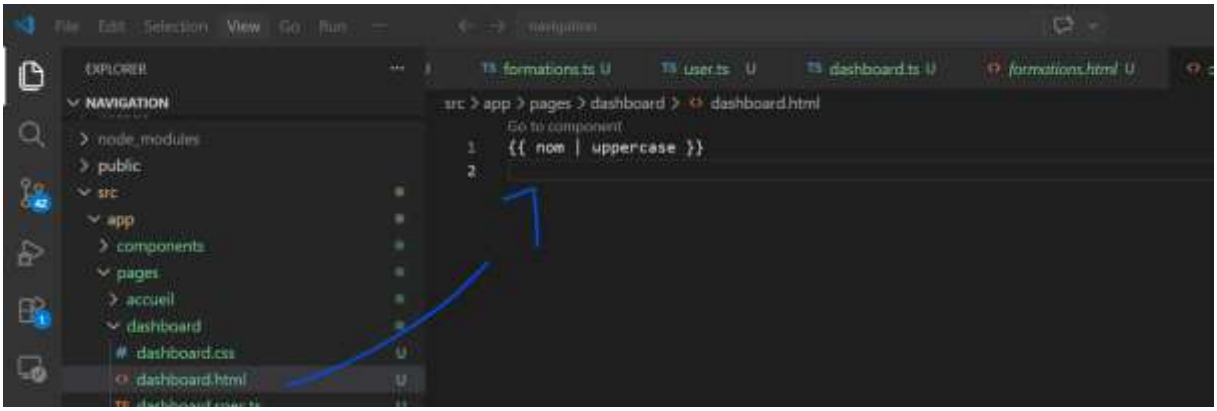
Console

Ahmed

Solution 1 : utiliser le constructeur

```
export class Dashboard {

 nom = "ahmed";

 constructor() {
   console.log(this.nom);
 }

}
```

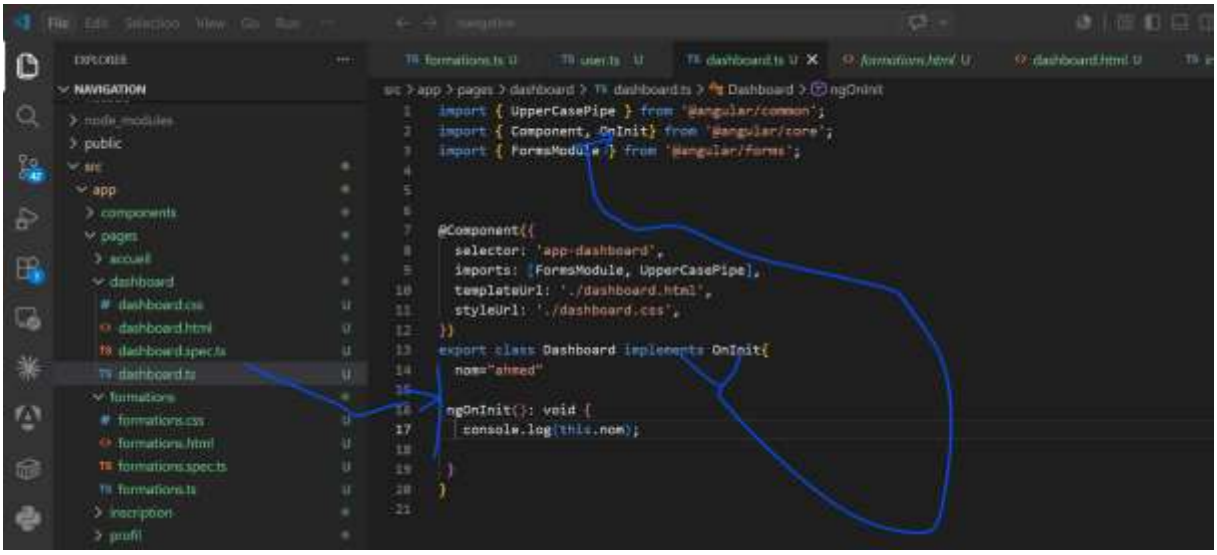


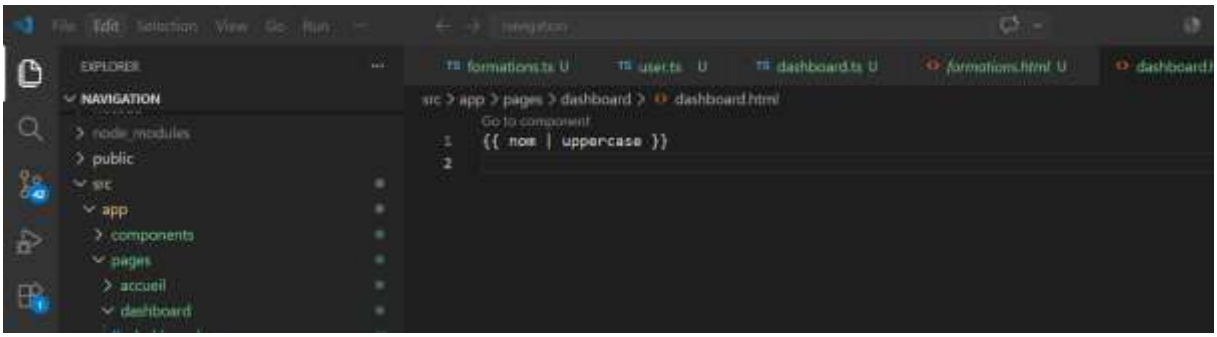


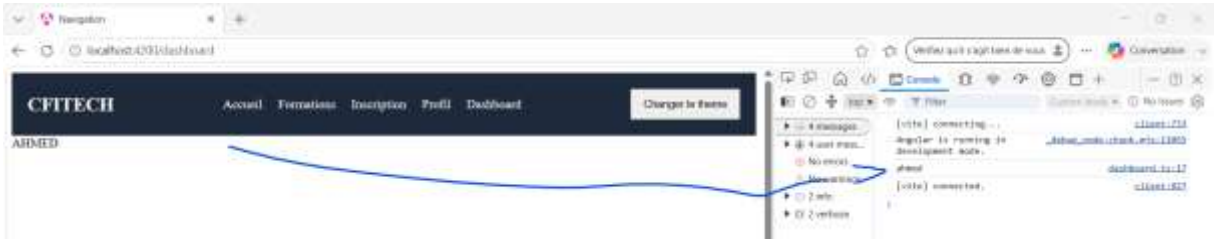
Solution 2 : utiliser ngOnInit()

ngOnInit() est une méthode qu'Angular exécute automatiquement au démarrage du composant. Elle ne fait rien par elle-même. C'est le développeur qui décide quoi mettre à l'intérieur : afficher un message, initialiser une variable, charger des données, appeler une API, ou simplement afficher une valeur dans la console.Lorsque le composant est prêt à être affiché, Angular exécute automatiquement la méthode ngOnInit().

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


export class Dashboard implements OnInit {


 nom = "ahmed";


 ngOnInit(): void {
  console.log(this.nom);
 }


}
```







- Un composant Angular est une **classe TypeScript**.
- Une **propriété** est une variable qui appartient à la classe.
- Une **méthode** est une fonction qui appartient à la classe.
- Le mot-clé **this** permet d'accéder aux propriétés ou aux méthodes de l'objet courant.

- On ne peut pas écrire une instruction comme console.log() directement dans le corps d'une classe : elle doit être placée dans une méthode, un constructeur ou ngOnInit().

- **constructor()** est appelé automatiquement lors de la création de l'objet.

- **ngOnInit()** est appelé automatiquement par Angular lorsque le composant est initialisé et prêt à exécuter sa logique de démarrage.

Solution 3 : appeler la méthode avec un bouton

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

  Afficher dans la console

</button>

Puis :

afficher() {

  console.log(this.nom);

}

Quand tu cliques sur le bouton, la console affiche :

ahmed

Ex 2: Carte d'identité

nom = "Mohamed";

prenom = "Ali";

HTML

Nom : {{ nom | uppercase }}

Prénom : {{ prenom }}

Résultat

Nom : MOHAMED

Prénom : Ali

Exercice 1

Créer les variables suivantes :

framework = "angular";

entreprise = "openai";

ville = "bruxelles";

Afficher toutes les valeurs en majuscules.

Rés :

ANGULAR

OPENAI

BRUXELLES

2. Le Pipe lowercase

Définition

Le **Pipe lowercase** permet de transformer un texte en **lettres minuscules**.

Comme uppercase, il ne modifie que l'affichage.

Syntaxe

{{ variable | lowercase }}

Ex1 :

dashboard.ts

nom = "ANGULAR";

dashboard.html

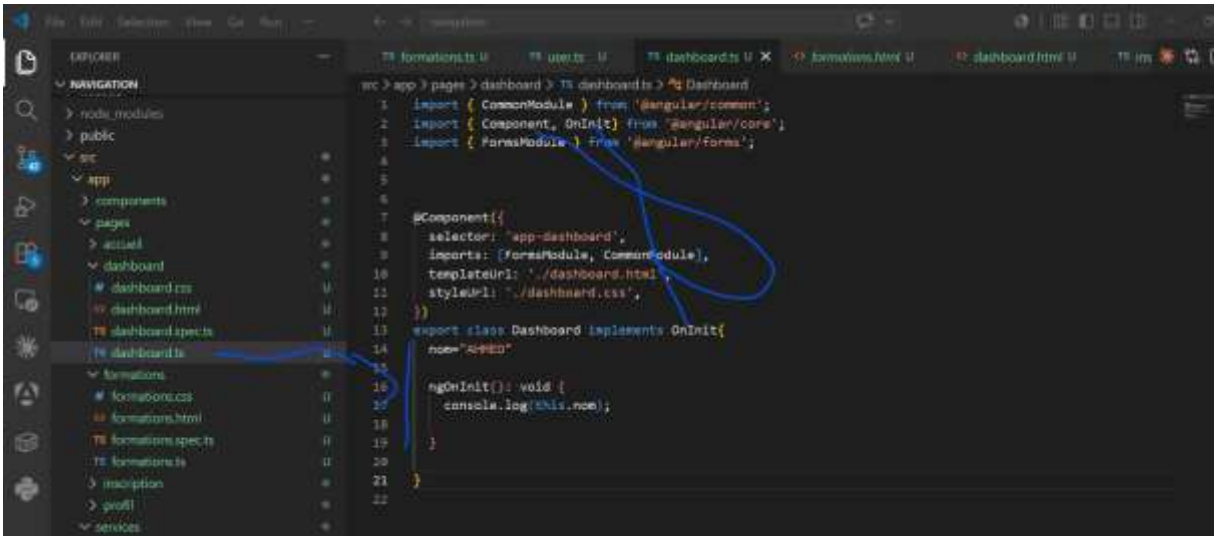
Valeur originale : {{ nom }}

Avec lowercase : {{ nom | lowercase }}

Résultat

Valeur originale : ANGULAR

Avec lowercase : angular



Ex : Adresse Email

Les emails sont souvent enregistrés avec des majuscules.

email = "Ahmed.KHALID@GMAIL.COM";

HTML

{{ email | lowercase }}

Résultat

ahmed.khalid@gmail.com

Ex : Nom d'une ville

ville = "PARIS";

HTML

{{ ville | lowercase }}

Résultat

paris

Ex: Nom d'une entreprise

entreprise = "MICROSOFT";

HTML

{{ entreprise | lowercase }}

Résultat

microsoft

Exercice 1

Créer les variables suivantes :

framework = "ANGULAR";

entreprise = "OPENAI";

ville = "BRUXELLES";

Afficher :

angular

openai

bruxelles

3. Le Pipe currency

Définition

Le **Pipe currency** permet d'afficher un **nombre sous la forme d'une monnaie**.

Il ajoute automatiquement :

- le symbole de la devise (€,$,£,...)
- les décimales
- le séparateur des milliers (selon la locale)

Le Pipe currency modifie uniquement l'affichage du nombre. La valeur de la variable reste inchangée dans le composant TypeScript.

Importation

Dans un composant **Standalone**, il suffit généralement d'importer **CommonModule**.

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

@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [CommonModule],
 templateUrl: './dashboard.html'
})
export class Dashboard {

}
```

Pourquoi utiliser le Pipe currency ?

Imaginons que nous avons un prix.

```
prix = 1250;
```

Sans Pipe :

```
{{ prix }}
```

Rés :

1250

Le client souhaite afficher :

1 250,00 €

ou

€1,250.00

Le Pipe currency permet de le faire automatiquement.

Syntaxe

{{ variable | currency }}

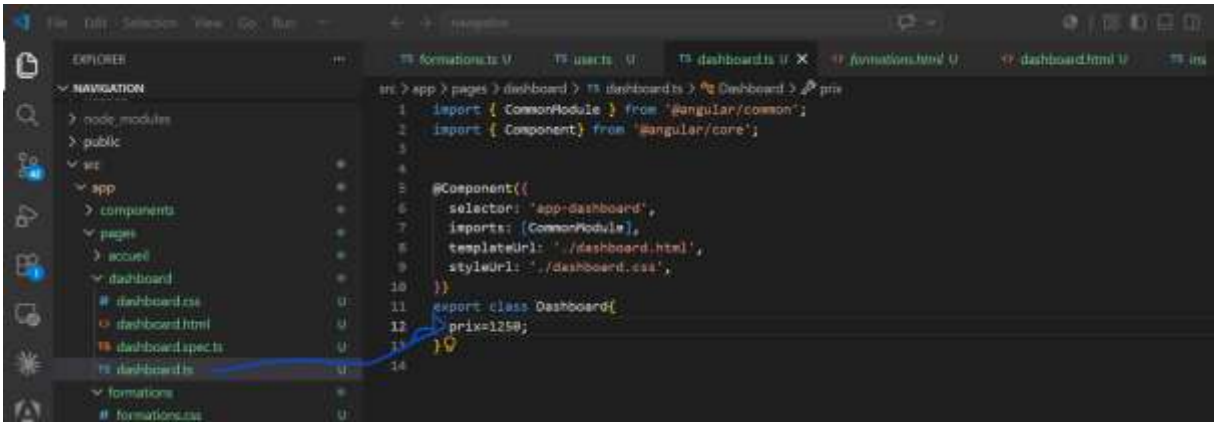
ou

{{ variable | currency:'EUR' }}

Ex1

dashboard.ts

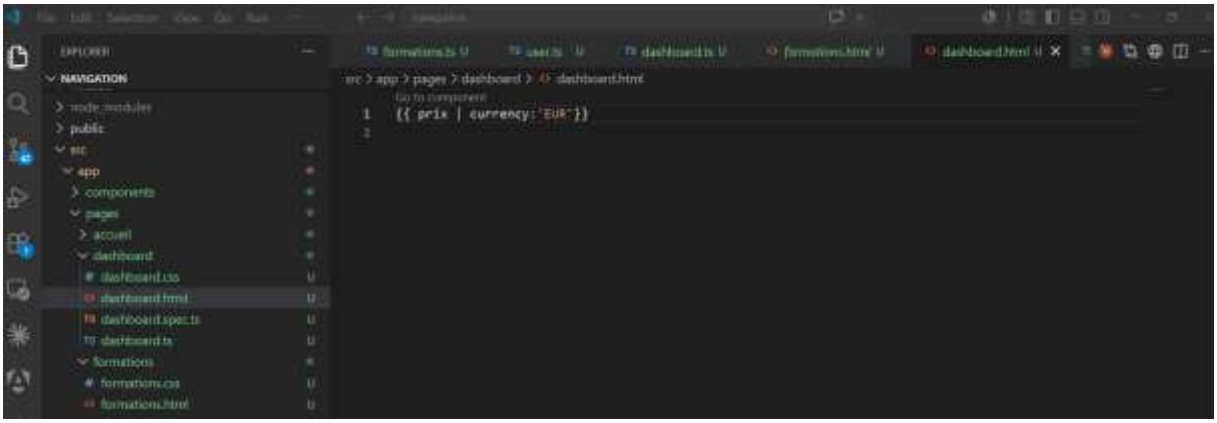
prix = 1250;



dashboard.html

Prix : {{ prix }}

Avec currency : {{ prix | currency }}



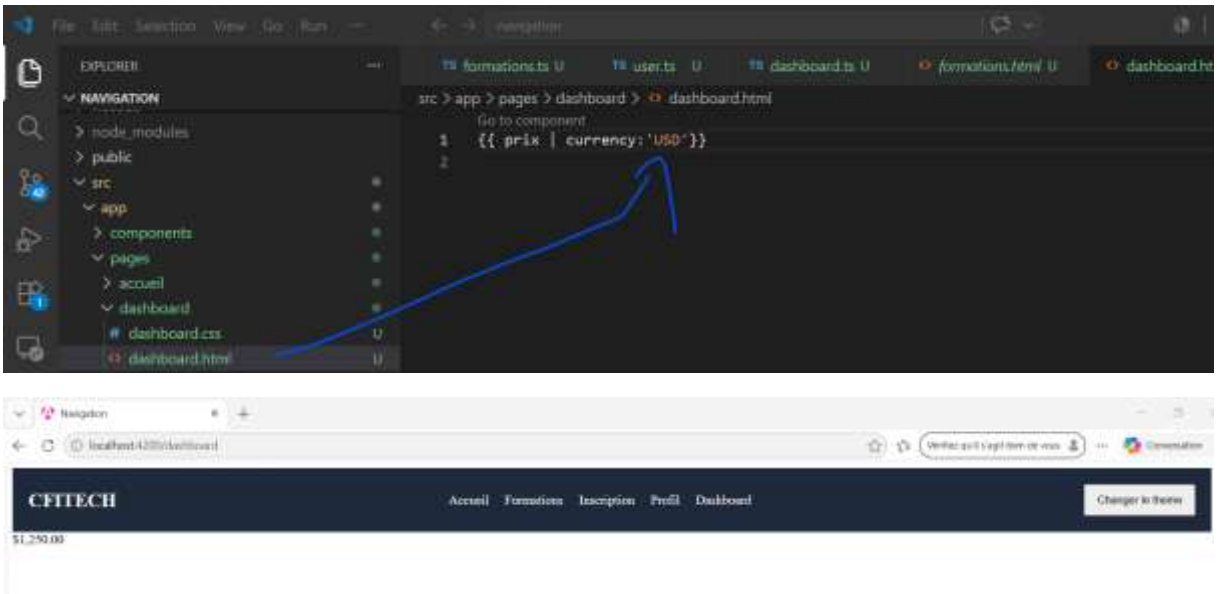
Ex : Afficher en Euro

Exemple de résultat (selon la configuration régionale de l'application)

Prix : 1250

Avec currency : $1,250.00

Le résultat dépend de la locale configurée dans l'application.



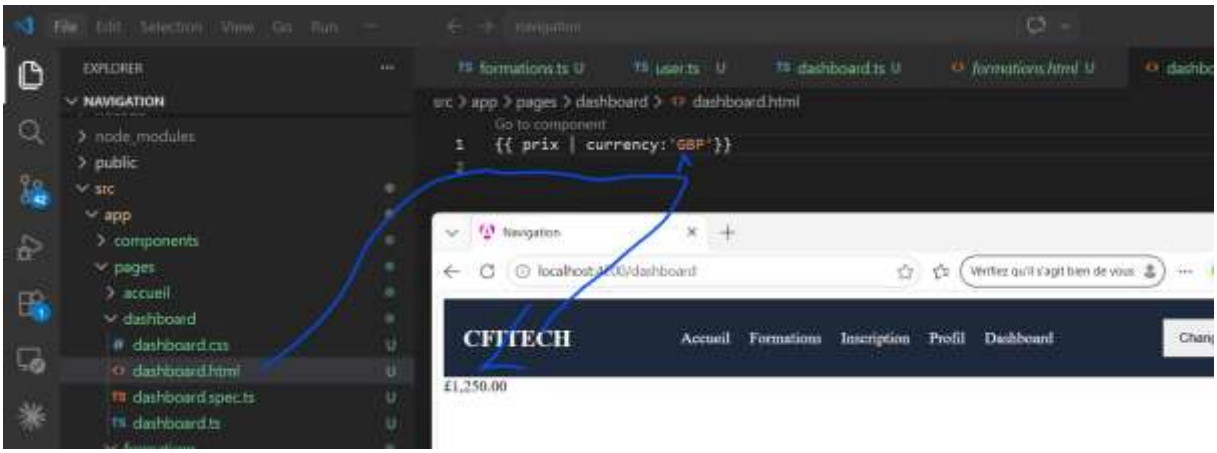
Ex: Afficher en Livre Sterling

prix = 1250;

{{ prix | currency:'GBP' }}

Résultat

£1,250.00



Ex : Site E-commerce

dashboard.ts

ordinateur = 950;

telephone = 699;

tablette = 450;

HTML

Ordinateur : {{ ordinateur | currency:'EUR' }}

Téléphone : {{ telephone | currency:'EUR' }}

Tablette : {{ tablette | currency:'EUR' }}

Rés :

Ordinateur : 950,00 €

Téléphone : 699,00 €

Tablette : 450,00 €

- Le Pipe **currency** sert à afficher un **nombre comme une monnaie**.
- Il ajoute automatiquement le **symbole**, les **décimales** et le **format de la devise**.
- Il **ne modifie jamais la valeur de la variable** dans le composant TypeScript.
- Dans un composant **Standalone**, il est généralement disponible en important **CommonModule**.
- On peut choisir la devise (EUR, USD, GBP...), le type d'affichage (symbol ou code) et le format des décimales.

4. Le Pipe percent

Définition

Le **Pipe percent** permet d'afficher un **nombre sous la forme d'un pourcentage (%)**.

Il multiplie automatiquement la valeur par **100**, puis ajoute le symbole **%**.

Le Pipe percent modifie uniquement l'affichage. La variable dans le composant TypeScript reste inchangée.

Importation

Dans un composant **Standalone**, il suffit d'importer CommonModule.

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

@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [CommonModule],
 templateUrl: './dashboard.html'
```

})

export class Dashboard {

}

Dans une application, certaines valeurs représentent un **taux** :

- une remise
- une taxe
- une progression
- un taux de réussite
- un pourcentage de batterie

Sans Pipe, on obtient une valeur difficile à comprendre.

Exemple :

taux = 0.25;

HTML

{{ taux }}

Résultat

0.25

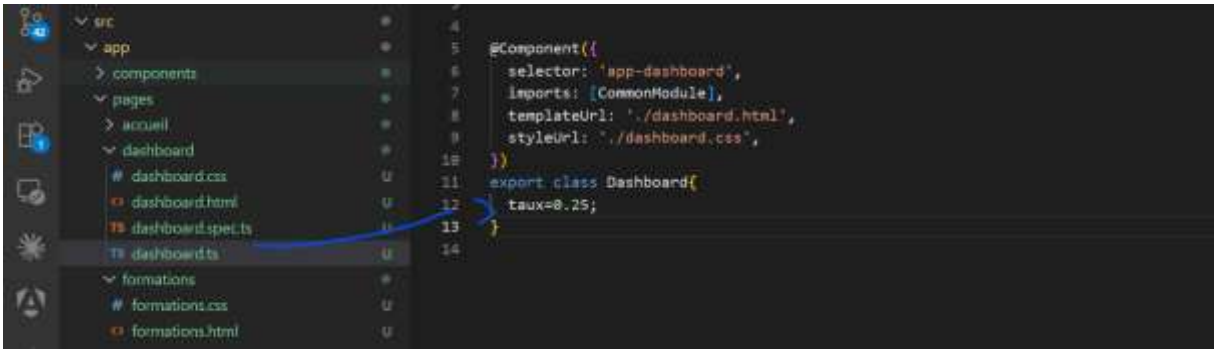
Ce n'est pas très parlant.

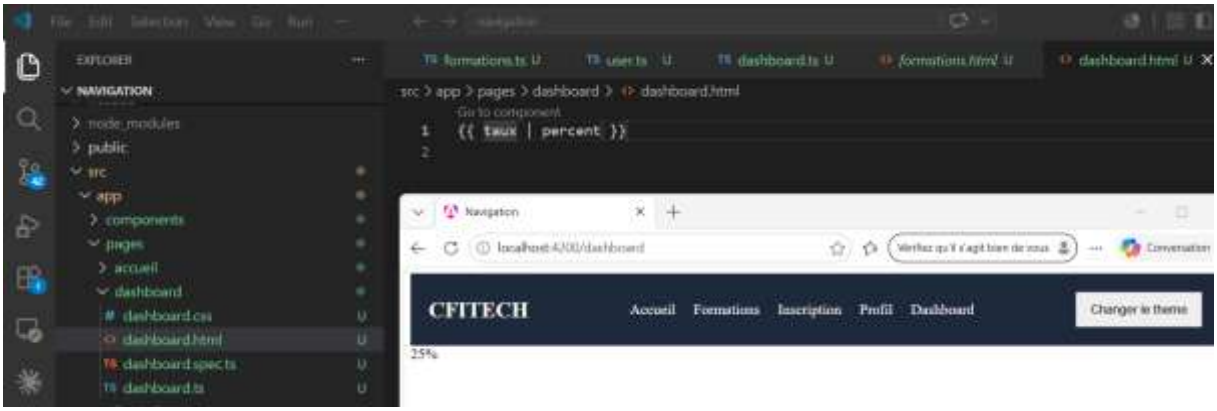
Avec le Pipe :

{{ taux | percent }}

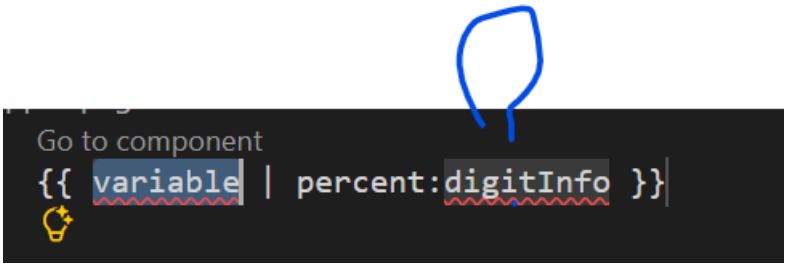
Résultat

25%





Ou



digitInfo n'est pas un mot-clé Angular. C'est juste un **placeholder** (un nom générique) utilisé dans la documentation Angular pour dire : **"ici, tu dois mettre le format des chiffres"**.

En pratique, tu remplaces digitInfo par une chaîne de caractères.

Par ex:

{{ taux | percent:'1.0-0' }}

ou

{{ taux | percent:'1.1-1' }}

ou

{{ taux | percent:'1.2-2' }}

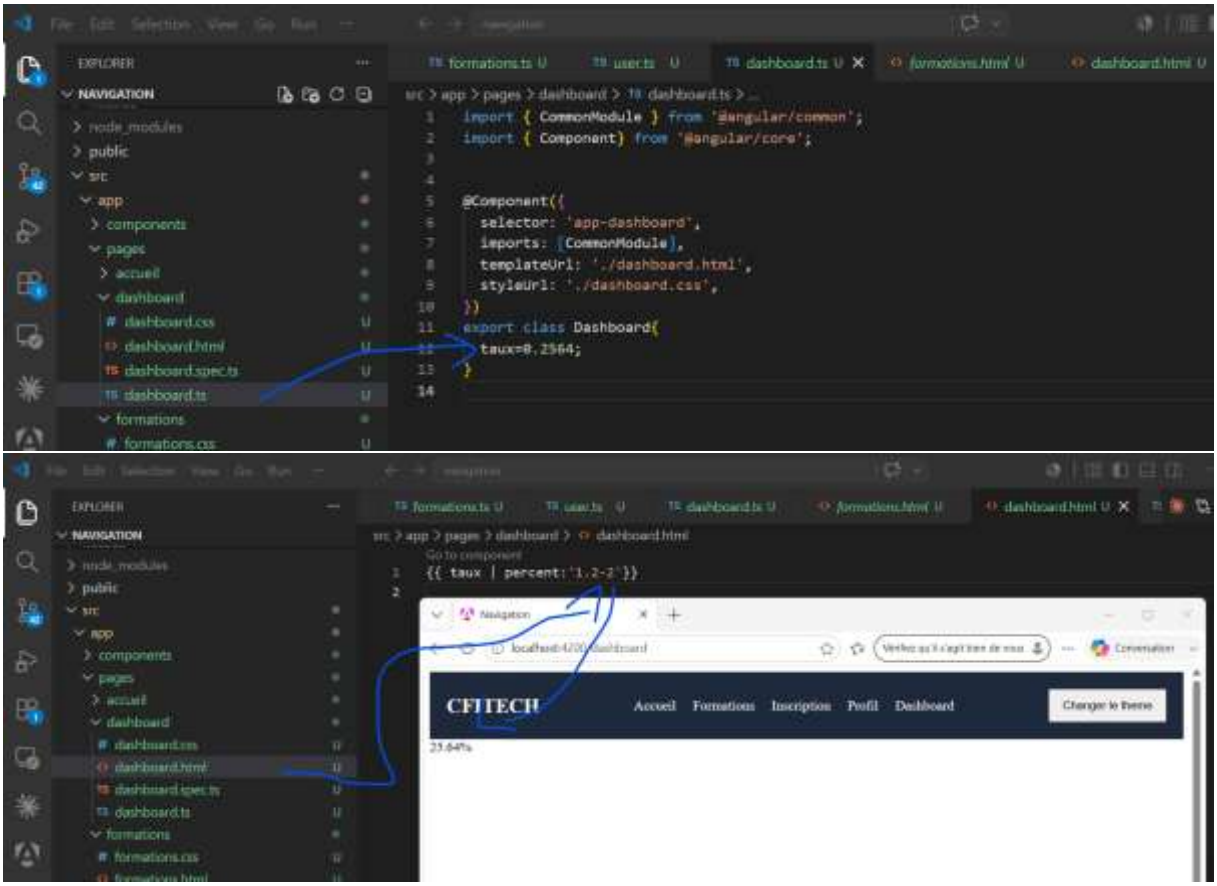
1.2-2

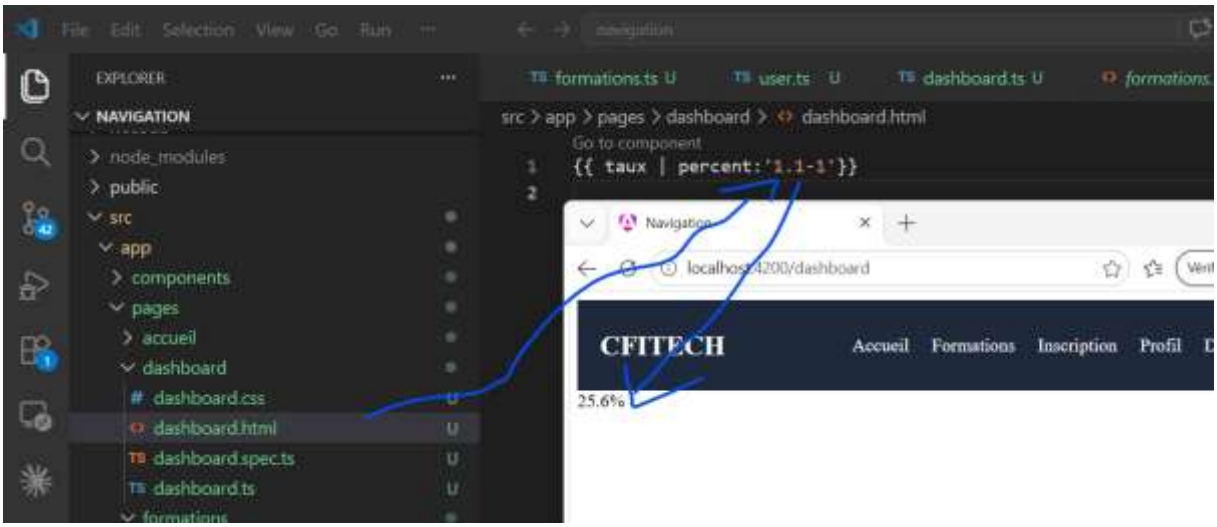
| | |

| | └─ Maximum 2 décimales

| └─── Minimum 2 décimales

└──── Minimum 1 chiffre avant la virgule





- Le Pipe **percent** transforme un nombre en **pourcentage**.
- Il **multiplie automatiquement la valeur par 100**, puis ajoute le symbole **%**.
- Pour afficher **50 %**, la variable doit contenir **0.5** (et non 50).
- Il **ne modifie jamais la valeur de la variable** dans le composant TypeScript.

- On peut personnaliser le nombre de décimales grâce au format, par exemple percent:'1.2-2'.

5. Le Pipe date

Définition

Le **Pipe date** permet de formater une **date** afin de l'afficher dans un format plus lisible.

Il peut afficher :

- le jour
- le mois
- l'année
- l'heure
- les minutes
- les secondes
- le jour de la semaine

Le Pipe date modifie uniquement l'affichage de la date. La valeur de la variable reste inchangée dans le composant TypeScript.

Importation

Dans un composant **Standalone**, il suffit d'importer CommonModule.

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

@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [CommonModule],
 templateUrl: './dashboard.html'
})
export class Dashboard {

}
```

Imaginons que nous avons une date.

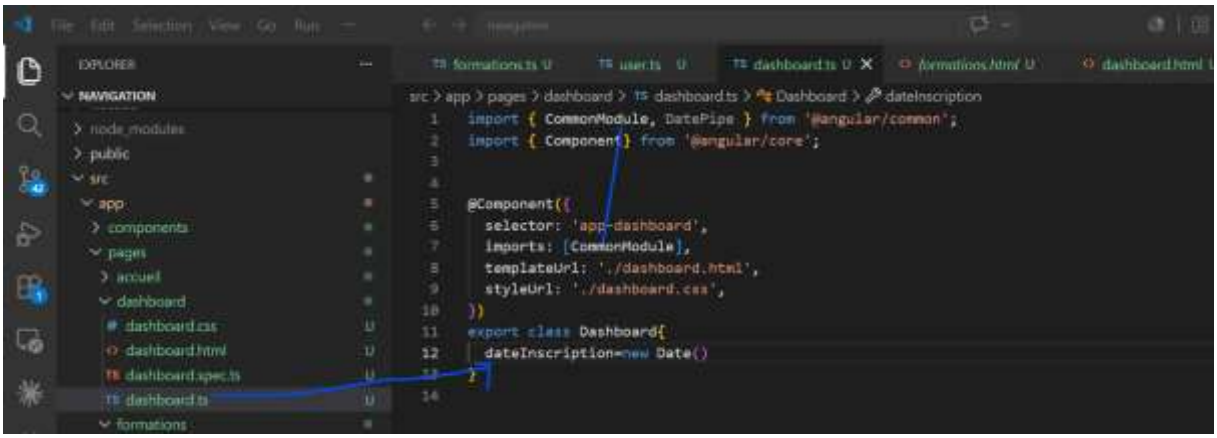
dateInscription = new Date();

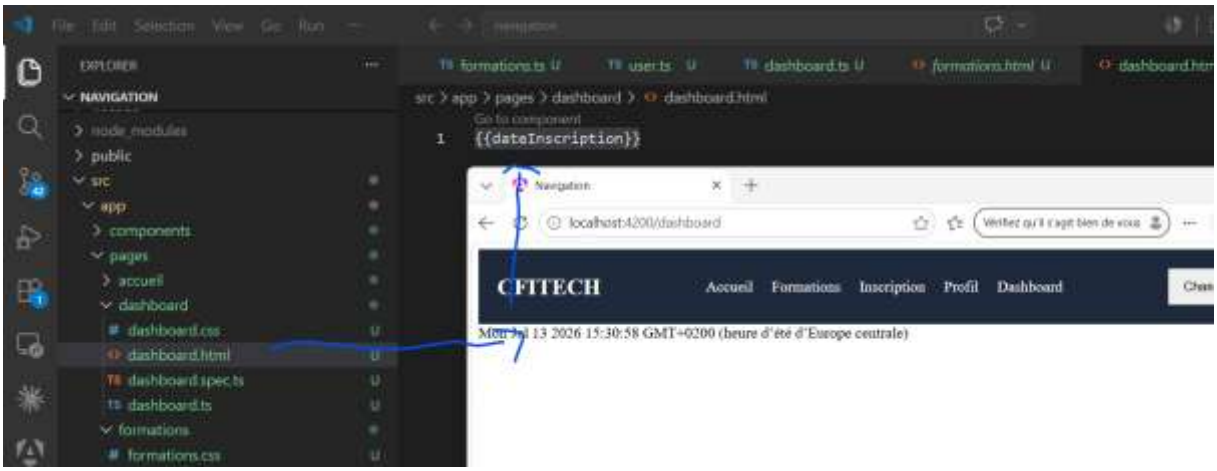
Sans Pipe :

{{ dateInscription }}

Rés :

Thu Jul 10 2025 14:36:18 GMT+0200





Ce n'est pas très lisible.

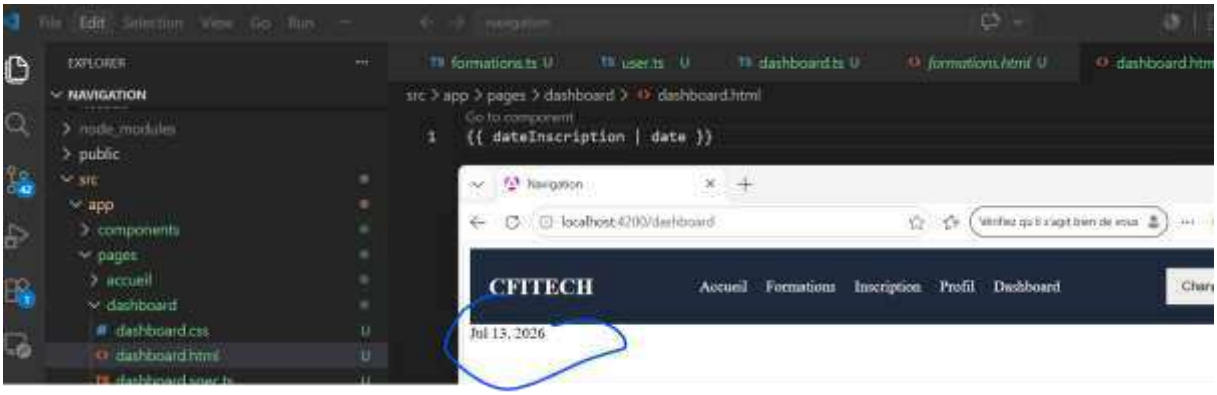
Avec le Pipe date :

{{ dateInscription | date }}

Rés :

Jul 10, 2025

L'affichage est beaucoup plus clair.



Syntaxe

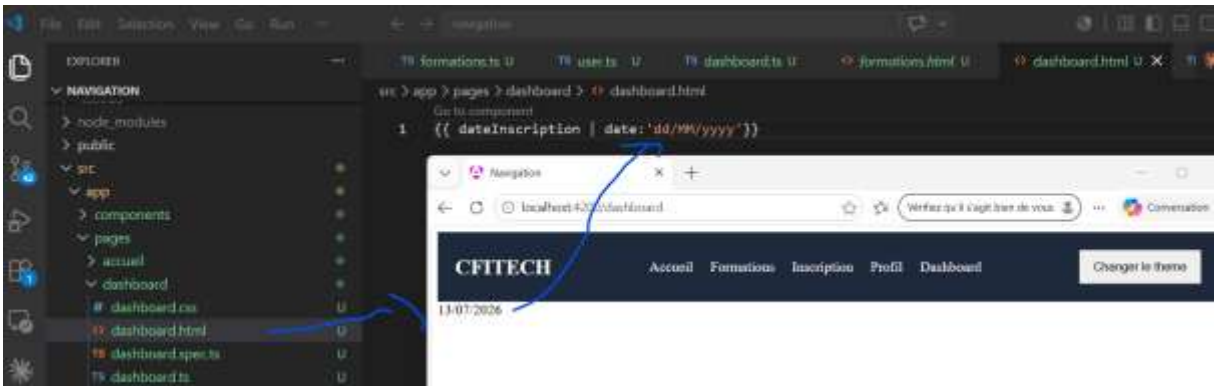
{{ variable | date }}

ou

{{ variable | date:'short' }}

ou

{{ variable | date:'dd/MM/yyyy' }}



Ex 1

dashboard.ts

dateAujourdhui = new Date();

dashboard.html

Date originale :

{{ dateAujourdhui }}

Avec le Pipe :

{{ dateAujourdhui | date }}

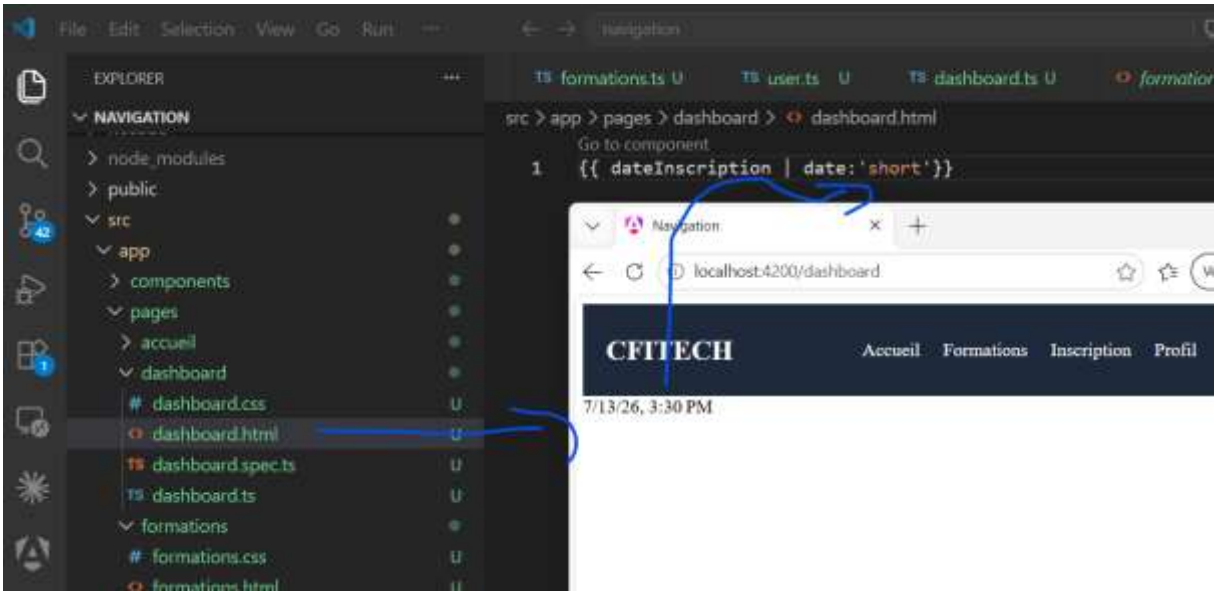
Résultat

Date originale :

Thu Jul 10 2025 14:36:18 GMT+0200

Avec le Pipe :

Jul 10, 2025



Les formats prédéfinis

Angular fournit plusieurs formats.

Pipe	Exemple de résultat
short	10/07/25 14:30
medium	Jul 10, 2025, 2:30:45 PM
long	July 10, 2025 at 2:30:45 PM GMT+2
full	Thursday, July 10, 2025 at 2:30:45 PM GMT+2
shortDate	10/07/25

Pipe	Exemple de résultat
mediumDate	Jul 10, 2025
longDate	July 10, 2025
fullDate	Thursday, July 10, 2025
shortTime	14:30
mediumTime	14:30:45

Exemple 2 : shortDate

date = new Date();

HTML

{{ date | date:'shortDate' }}

Rés :

10/07/25

Exemple 3 : longDate

{{ date | date:'longDate' }}

Rés :

July 10, 2025

Exemple 4 : fullDate

{{ date | date:'fullDate' }}

Rés :

Thursday, July 10, 2025

Créer son propre format

C'est le plus utilisé en entreprise.

Syntaxe

{{ date | date:'dd/MM/yyyy' }}

Les principaux symboles

Symbole	Signification
dd	Jour (01-31)
MM	Mois (01-12)
yyyy	Année sur 4 chiffres
yy	Année sur 2 chiffres
HH	Heure (24 h)
mm	Minutes
ss	Secondes

Exemple 5

dashboard.ts

date = new Date();

HTML

{{ date | date:'dd/MM/yyyy' }}

Rés :

10/07/2025

Exemple 6

{{ date | date:'dd-MM-yyyy' }}

Rés :

10-07-2025

Exemple 7

{{ date | date:'yyyy/MM/dd' }}

Rés :

2025/07/10

Exemple 8 : Date et heure

{{ date | date:'dd/MM/yyyy HH:mm:ss' }}

Rés :

10/07/2025 14:35:42

Ex: Date de naissance

dateNaissance = new Date(1998,4,15);

HTML

Date de naissance :

{{ dateNaissance | date:'dd/MM/yyyy' }}

Rés :

15/05/1998

Ex : Heure actuelle

heure = new Date();

HTML

Heure actuelle :

{{ heure | date:'HH:mm:ss' }}

Rés :

14:35:20

Comprendre les symboles du Pipe date

Lorsque nous utilisons le Pipe date, nous pouvons personnaliser l'affichage grâce à des **symboles**.

Par exemple :

{{ date | date:'EEEE dd MMMM yyyy' }}

Chaque symbole a une signification précise.

Symbole	Signification	Exemple (16 juillet 2026)
EEEE	Jour de la semaine	jeudi
dd	Jour du mois sur 2 chiffres	16
MMMM	Nom complet du mois	juillet
yyyy	Année sur 4 chiffres	2026

Si la variable contient la date d'aujourd'hui :

date = new Date();

Alors :

{{ date | date:'EEEE dd MMMM yyyy' }}

affichera :

jeudi 16 juillet 2026

Pourquoi Angular utilise la lettre E pour le jour de la semaine et non la lettre D ?

Angular n'a pas inventé cette notation.

Il utilise une **norme internationale de formatage des dates (Unicode CLDR / ICU)**, également utilisée par Java, Android, JavaScript et d'autres technologies.

Dans cette norme, chaque lettre représente une partie de la date.

Lettre	Représente
y	Year (année)
M	Month (mois)
d	Day of month (jour du mois)
H	Hour (heure)
m	Minute
s	Seconde
E	Day of week (jour de la semaine)

Pourquoi ne pas utiliser D ?

La lettre **d** est déjà utilisée pour représenter le **jour du mois**.

Par exemple :

{{ date | date:'dd/MM/yyyy' }}

Avec la date d'aujourd'hui, le résultat sera :

16/07/2026

Ici :

- dd → **16** (jour du mois)
- MM → **07** (mois)
- yyyy → **2026** (année)

Comme d est déjà utilisé, la norme internationale a choisi la lettre **E** pour représenter le **jour de la semaine**.

Que signifie EEEE ?

Le nombre de lettres indique simplement **le niveau de détail de l'affichage**.

Si aujourd'hui est **jeudi**, voici ce que l'on obtient :

Format	Résultat
E	J
EE	Je
EEE	Jeu
EEEE	Jeudi

En anglais :

Format	Résultat
E	T
EE	Th
EEE	Thu
EEEE	Thursday

Même principe avec les mois

Le même fonctionnement existe pour les mois.

Si nous sommes en **juillet** :

Format	Résultat
M	7
MM	07

Format Résultat

MMM juil.

MMMM juillet

Exemple :

{{ date | date:'MMMM yyyy' }}

Résultat :

juillet 2026

Symboles fréquemment utilisés

Supposons qu'il soit **16 juillet 2026 à 09 h 05 min 08 s**.

Symbole	Signification	Résultat
d	Jour du mois	16
dd	Jour du mois sur 2 chiffres	16
M	Mois	7
MM	Mois sur 2 chiffres	07
MMM	Mois abrégé	juil.
MMMM	Nom complet du mois	juillet
yy	Année sur 2 chiffres	26
yyyy	Année sur 4 chiffres	2026
H	Heure	9
HH	Heure sur 2 chiffres	09
m	Minutes	5
mm	Minutes sur 2 chiffres	05
s	Secondes	8
ss	Secondes sur 2 chiffres	08
E	Jour de la semaine abrégé	J
EEE	Jour abrégé	Jeu
EEEE	Jour complet	Jeudi

Exemples complets

Exemple 1

{{ date | date:'dd/MM/yyyy' }}

Résultat

16/07/2026

Exemple 2

{{ date | date:'dd-MM-yyyy' }}

Résultat

16-07-2026

Exemple 3

{{ date | date:'yyyy/MM/dd' }}

Résultat

2026/07/16

Exemple 4

{{ date | date:'EEEE dd MMMM yyyy' }}

Résultat

jeudi 16 juillet 2026

Exemple 5

{{ date | date:'HH:mm:ss' }}

Résultat (si l'heure est 09 h 05 min 08 s)

09:05:08

Exemple 6

{{ date | date:'EEEE dd MMMM yyyy HH:mm:ss' }}

Résultat

jeudi 16 juillet 2026 09:05:08

- Le Pipe date utilise une **norme internationale** pour formater les dates.
- Chaque lettre représente une partie de la date (y, M, d, H, m, s, E...).
- La lettre **d** correspond au **jour du mois**.
- La lettre **E** correspond au **jour de la semaine**.
- Plus une lettre est répétée (MM, MMMM, EEE, EEEE), plus l'affichage est détaillé.
- Les formats les plus utilisés en entreprise sont :
    - dd/MM/yyyy → **16/07/2026**
    - HH:mm:ss → **09:05:08**
    - dd/MM/yyyy HH:mm:ss → **16/07/2026 09:05:08**
    - EEEE dd MMMM yyyy → **jeudi 16 juillet 2026**.

6. Le Pipe json

Définition

Le **Pipe json** permet d'afficher un **objet JavaScript ou TypeScript** au format **JSON**.

Il est principalement utilisé pour le **débogage (debug)** afin de voir le contenu d'un objet ou d'un tableau directement dans le navigateur.

Le Pipe json ne sert pas à modifier les données. Il permet simplement au développeur **d'afficher le contenu complet d'un objet ou d'un tableau afin de vérifier que les données sont correctes pendant le développement**. C'est pour cette raison qu'on dit qu'il est utilisé pour le **débogage (debugging)**.

Le Pipe json est comme une "loupe" : il permet au développeur de voir exactement ce qu'il y a à l'intérieur d'un objet ou d'un tableau pendant qu'il développe son application.

En entreprise, on l'utilise surtout pendant le développement, rarement dans une application en production.

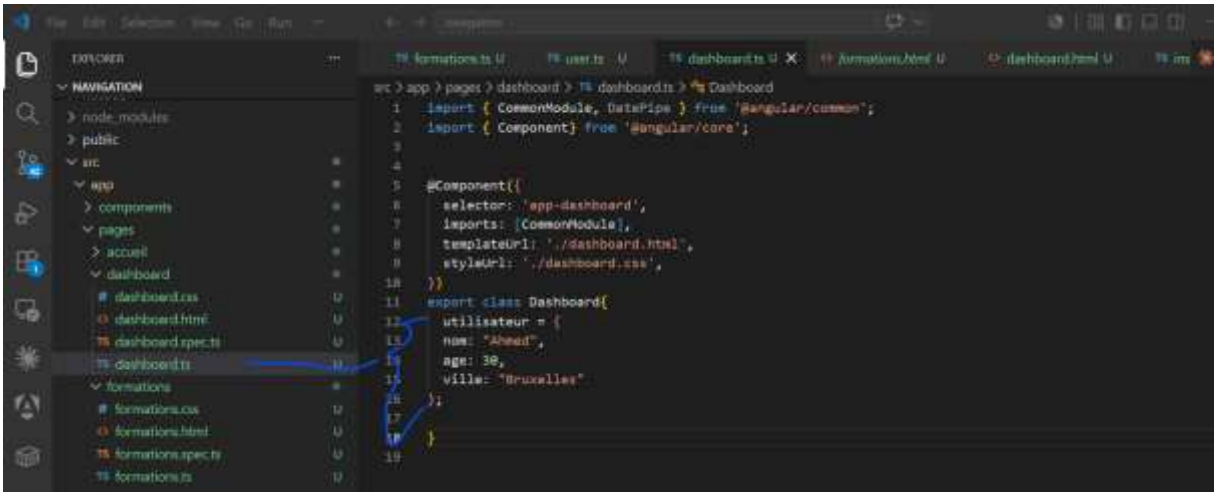
Importation

Dans un composant **Standalone**, il suffit d'importer CommonModule.

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


@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [CommonModule],
 templateUrl: './dashboard.html'
})
export class Dashboard {


}
```

Imaginons que nous avons un objet.

```
utilisateur = {
 nom: "Ahmed",
 age: 30,
 ville: "Bruxelles"
};
```



Si on écrit :

{{ utilisateur }}

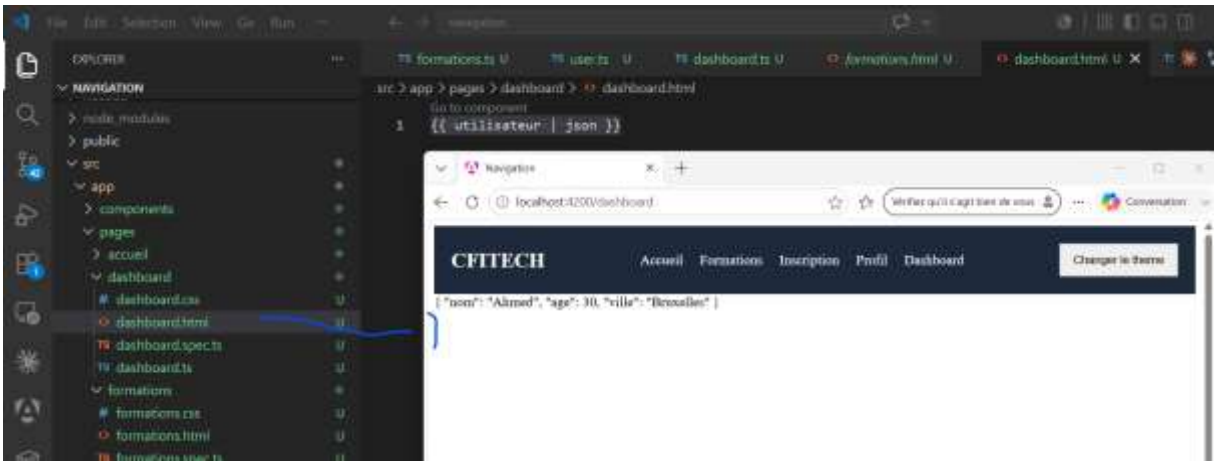
Le navigateur affiche

[object Object]

Impossible de voir les propriétés.

Avec le Pipe json :

{{ utilisateur | json }}



Rés :

{

"nom": "Ahmed",

"age": 30,

"ville": "Bruxelles"

}

Syntaxe

{{ variable | json }}

Ex : Objet

dashboard.ts

utilisateur = {

nom: "Ahmed",

email: "ahmed@gmail.com",

age: 30

};

dashboard.html

{{ utilisateur | json }}

Résultat


```
{
  "nom": "Ahmed",
  "email": "ahmed@gmail.com",
  "age": 30
}
```


Ex : Tableau

```
formations = [
  "Angular",
  "Java",
  "Spring Boot"
];
```

HTML

{{ formations | json }}

Résultat


```
[
  "Angular",
  "Java",
  "Spring Boot"
]
```


Ex : Tableau d'objets

```
etudiants = [

 {
   nom: "Ahmed",
   age: 22
 },
```

```
 {
   nom: "Sara",
   age: 24
 }

];
```

HTML

```
{{ etudiants | json }}
```

Rés :

```
[
 {
   "nom":"Ahmed",
   "age":22
 },
 {
   "nom":"Sara",
   "age":24
 }
]
```

Ex : concret

Imaginons un formulaire.

```
utilisateur = {

  nom: "Ahmed",

  email: "ahmed@gmail.com",

  ville: "Bruxelles"
```

};

Pendant le développement, on veut voir le contenu de l'objet.

<pre>{{ utilisateur | json }}</pre>

Rés :

{

"nom":"Ahmed",

"email":"ahmed@gmail.com",

"ville":"Bruxelles"

}

Le <pre> permet de conserver les retours à la ligne.

- Le Pipe **json** transforme un objet ou un tableau en texte JSON.
- Très utile pour **déboguer** une application.
- Il est principalement utilisé pendant le développement.

Le Pipe slice

Définition

Le **Pipe slice** permet d'extraire une partie :

- d'une chaîne de caractères
- d'un tableau

Il fonctionne comme la méthode JavaScript **slice()**.

Syntaxe

{{ variable | slice:debut:fin }}

où :

- **debut** : indice de départ
- **fin** : indice de fin (non inclus)

Exemple 1 : Sur une chaîne

dashboard.ts

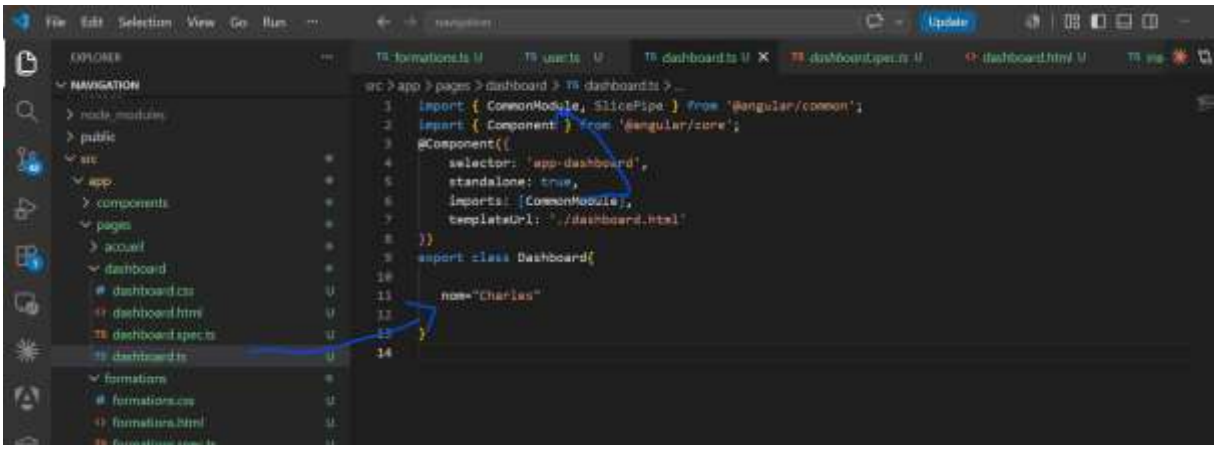
nom = "Mohamed";

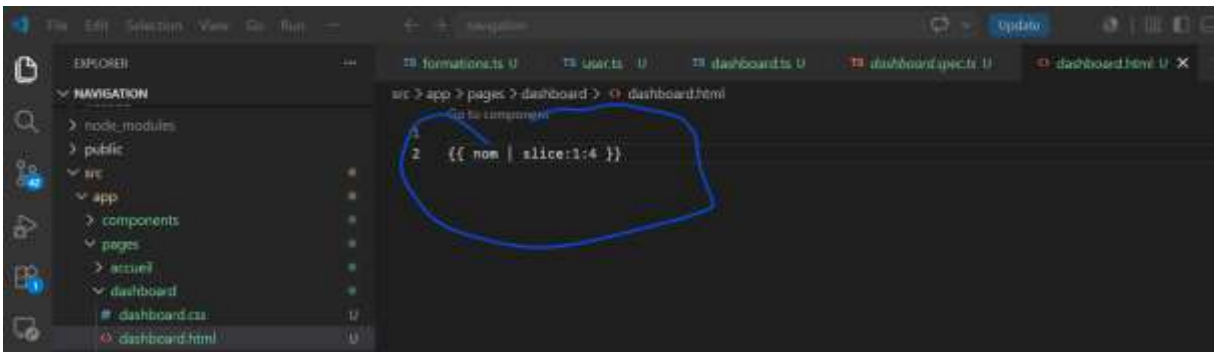
HTML

{{ nom | slice:0:3 }}

Résultat

Moh



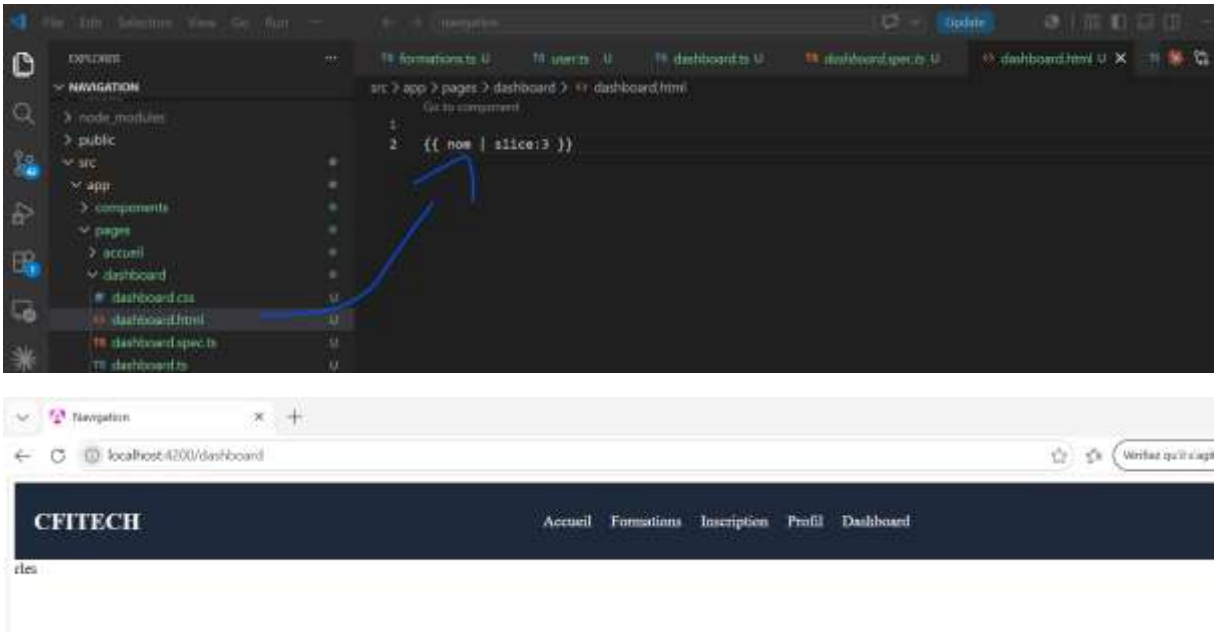


Exemple 2

{{ nom | slice:3 }}

Résultat

amed



Exemple 3

{{ nom | slice:2:5 }}

Résultat

ham

Exemple concret : Masquer une partie d'un numéro

telephone = "0475123456";

HTML

{{ telephone | slice:0:4 }}

Résultat

0475

Exemple concret : Afficher les 4 derniers chiffres

{{ telephone | slice:6 }}

Résultat

3456

Utiliser slice avec un tableau

dashboard.ts

formations = [

"Angular",

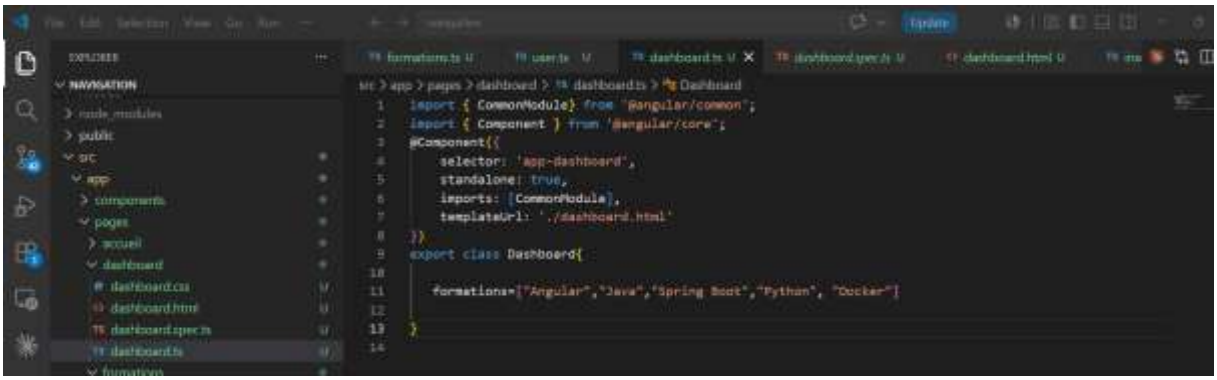
"Java",

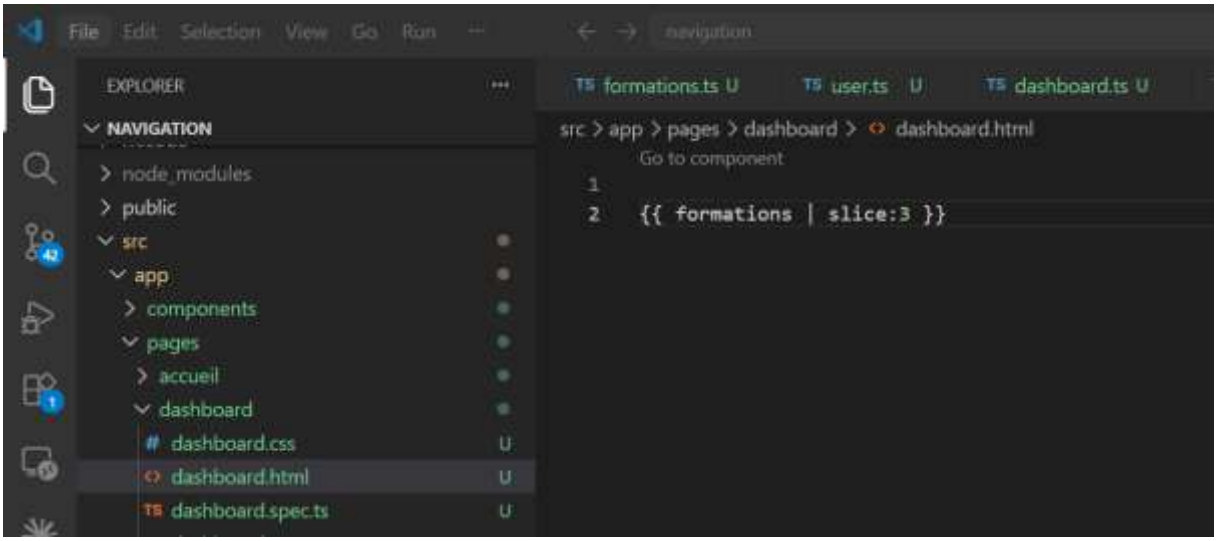
"Spring Boot",

"Python",

"Docker"

];





Exemple 1

HTML

{{ formations | slice:0:3 | json }}

Résultat

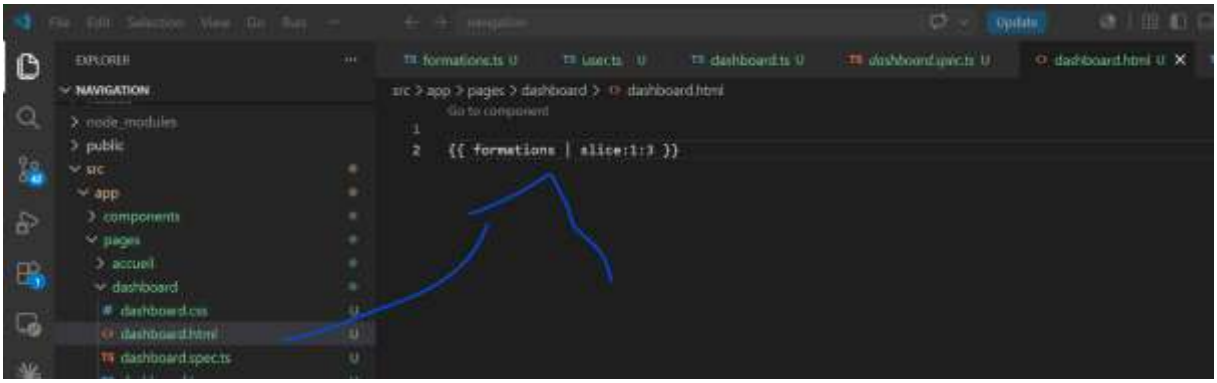
[

"Angular",

"Java",

"Spring Boot"

]



Exemple 2

{{ formations | slice:2 | json }}

Résultat

[

"Spring Boot",

"Python",

"Docker"

]

Exemple 3

{{ formations | slice:1:4 | json }}

Résultat

[

"Java",

"Spring Boot",

"Python"

]

Exemple concret : Liste des étudiants

etudiants = [

"Ahmed",

"Sara",

"Youssef",

"Nadia",

"Ali"

];

HTML

Premiers étudiants :

{{ etudiants | slice:0:2 | json }}

Résultat

[

"Ahmed",

"Sara"

]

Utiliser un indice négatif

Comme en JavaScript, on peut utiliser des indices négatifs.

Quand tu mets un nombre négatif, Angular commence à compter **depuis la fin**.

Le même mot peut être vu comme ceci :

Lettre	**A**	**n**	**g**	**u**	**l**	**a**	**r**
Indice positif	0	1	2	3	4	5	6
Indice négatif	-7	-6	-5	-4	**-3**	-2	-1

dashboard.ts

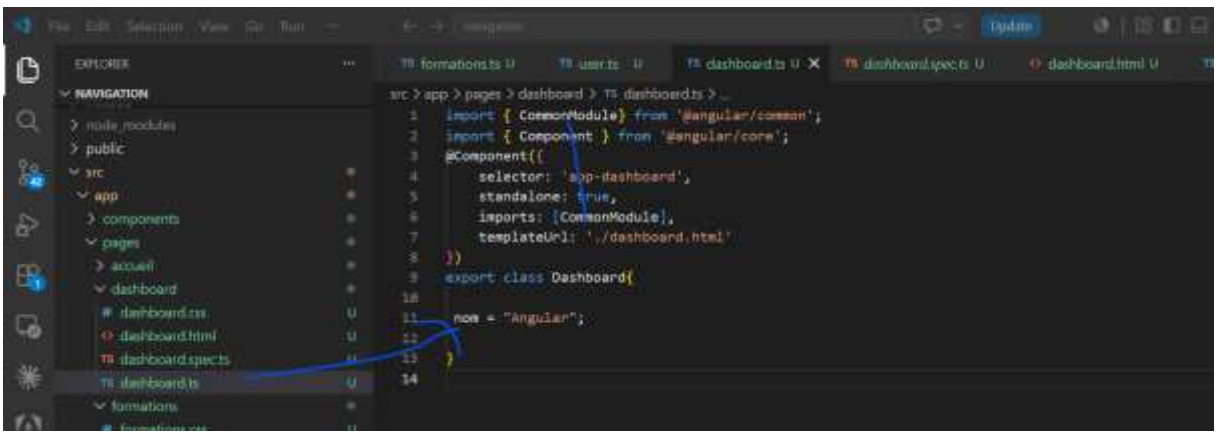
nom = "Angular";

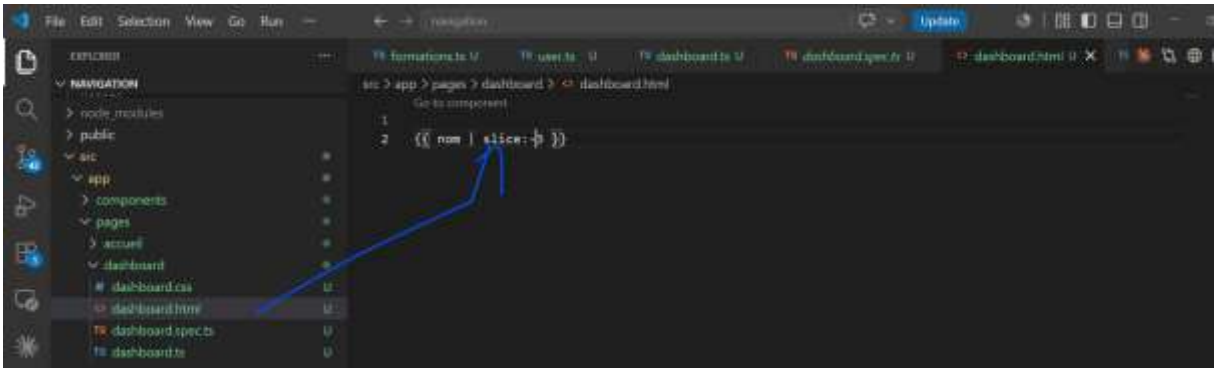
HTML

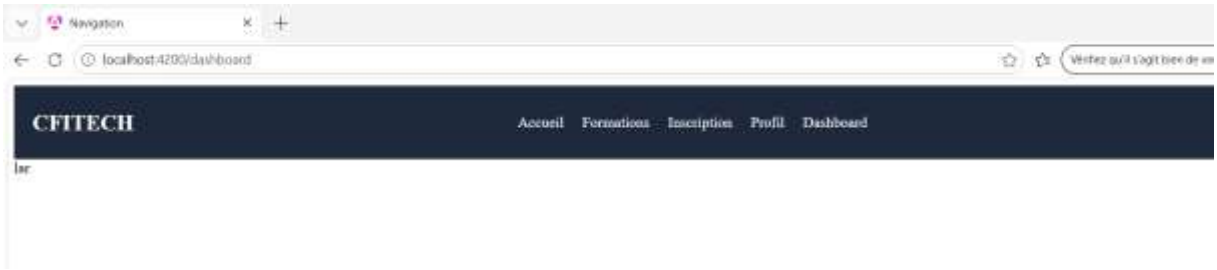
{{ nom | slice:-3 }}

Résultat

lar







Autre exemple

formations = [

"Angular",

"Java",

"Spring Boot",

"Python"

];

HTML

{{ formations | slice:-2 | json }}

Résultat

[

"Spring Boot",

"Python"

]

Exercices

Exercice 1

Créer

nom = "OpenAI";

Afficher :

Open

Exercice 2

Créer

telephone = "0499123456";

Afficher uniquement :

3456

Exercice 3

Créer

technologies = [

"Angular",

"Spring",

"Java",

"Docker",

"Kubernetes"

];

Afficher uniquement les **3 premières technologies**.

Résultat attendu

[

"Angular",

"Spring",

"Java"

]

- Le Pipe **slice** permet d'extraire une partie d'une chaîne de caractères ou d'un tableau.
- slice accepte un **indice de début**, un **indice de fin** (optionnel) et même des **indices négatifs** pour compter à partir de la fin.
- Il est fréquent de combiner slice avec json pour visualiser facilement une partie d'un tableau dans le template.

Les Pipes personnalisés (Custom Pipes)

Définition

Jusqu'à présent, nous avons utilisé les Pipes fournis par Angular :

- uppercase
- lowercase
- currency
- date
- percent
- json
- slice

Ces Pipes répondent à de nombreux besoins.

Mais il arrive qu'une application ait des besoins spécifiques.

Par exemple :

- Transformer un nom en majuscules avec des étoiles.
- Masquer une partie d'un numéro de téléphone.
- Ajouter automatiquement "Bonjour".
- Transformer une note en mention.
- Convertir un nombre en texte.

Les Pipes d'Angular ne savent pas faire cela.

Dans ce cas, **nous créons notre propre Pipe**.

C'est ce qu'on appelle un **Pipe personnalisé (Custom Pipe)**.

Ex :

Imaginons que dans toute l'application nous voulons afficher :

Ahmed

sous la forme

Bonjour Ahmed

Angular ne possède pas de Pipe qui fait cela.

Nous allons donc créer le nôtre.

Création d'un Pipe

Commande :

ng generate pipe pipes/bonjour

ou la version courte :

ng g p bonjour

Angular crée automatiquement :

bonjour.pipe.ts

Structure générée

Angular génère un fichier ressemblant à ceci :

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

@Pipe({
  name: 'bonjour'
})
export class BonjourPipe implements PipeTransform {

  transform(value: unknown): unknown {
    return value;
  }

}
```

```
@Pipe({
  name: 'bonjour'
})
```

Cette partie déclare un nouveau Pipe.

Son nom est **bonjour**.

C'est ce nom qui sera utilisé dans le HTML.

```
implements PipeTransform
```

La classe doit implémenter l'interface **PipeTransform**.

Cette interface oblige Angular à créer la méthode :

```
transform()
```

Premier Pipe personnalisé

Modifions le code.

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

@Pipe({
  name: 'bonjour'
})
export class BonjourPipe implements PipeTransform {

  transform(value: string): string {

    return "Bonjour " + value;

  }

}
```

Utilisation

Dans le composant Standalone :

```
import { Component } from '@angular/core';
import { BonjourPipe } from './bonjour.pipe';

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

```
  nom = "Ahmed";


}
```

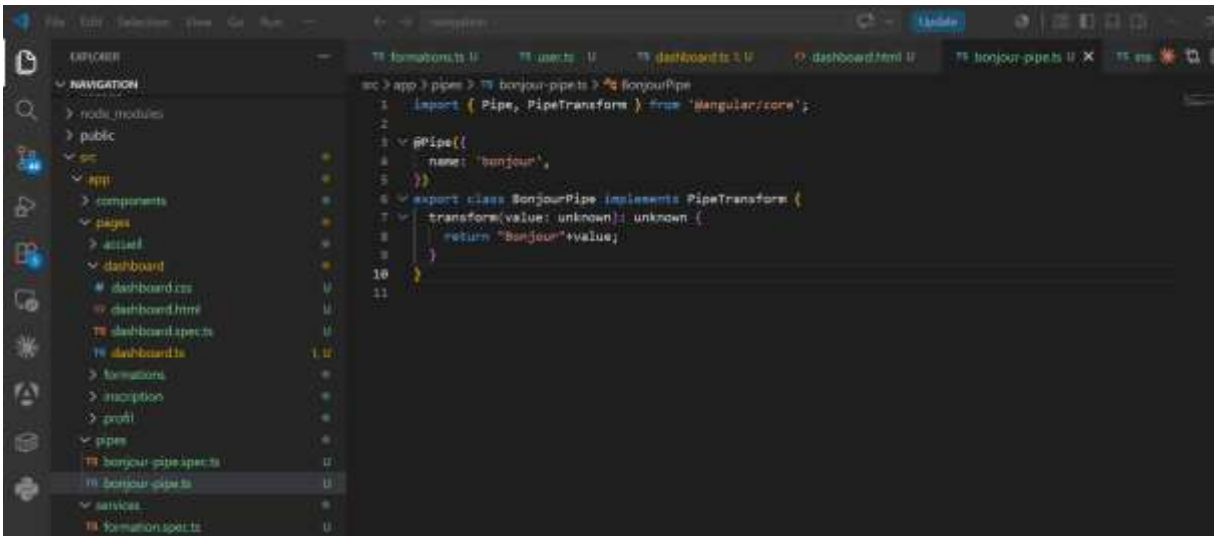
HTML

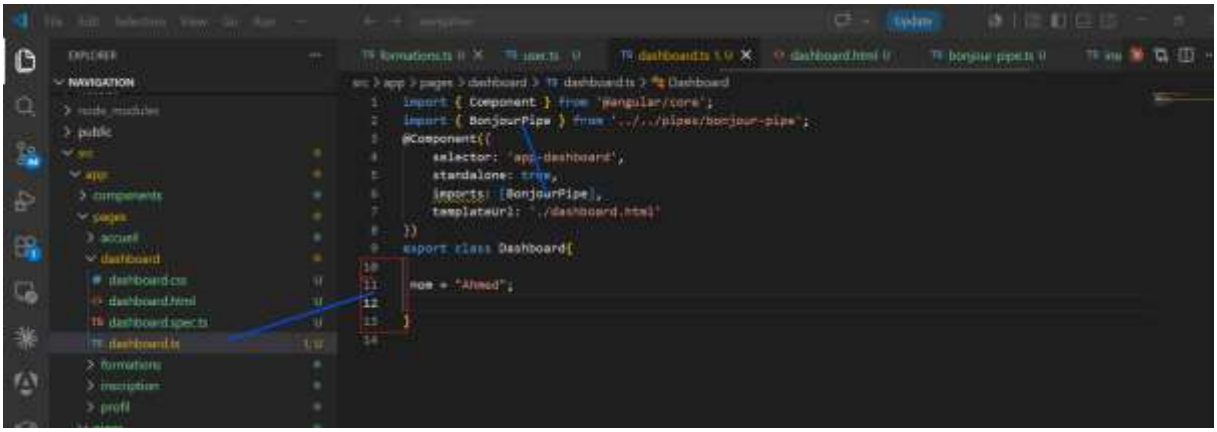
{{ nom | bonjour }}

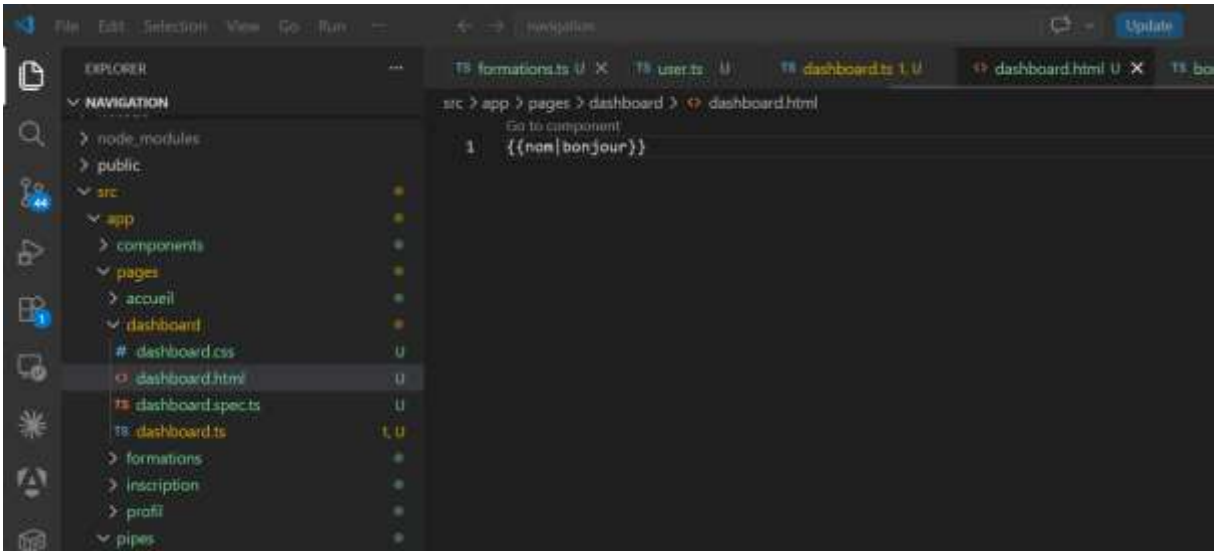
Résultat

Bonjour Ahmed

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g p pipes/bonjour
CREATE src/app/pipes/bonjour-pipe.spec.ts (199 bytes)
CREATE src/app/pipes/bonjour-pipe.ts (232 bytes)
```







Le Pipe agit comme une fonction qui reçoit une valeur et retourne une nouvelle valeur.

Exemple concret 2 : Ajouter des étoiles

transform(value: string): string {

return "*** " + value + " ***";

}

HTML

{{ nom | bonjour }}

Résultat

*** Ahmed ***

Exemple concret 3 : Transformer une note en mention

note.pipe.ts

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

@Pipe({
  name: 'mention'
})
export class MentionPipe implements PipeTransform {

  transform(note: number): string {

    if (note >= 16)
      return "Très Bien";

    if (note >= 14)
      return "Bien";

    if (note >= 12)
      return "Assez Bien";

    return "Insuffisant";

  }

}
```

Composant

```
note = 15;
```

HTML

```
{{ note | mention }}
```

Résultat

Bien

Exemple concret 4 : Masquer un numéro

transform(value: string): string {

return "******" + value.slice(-4);

}

TS

telephone = "0499123456";

HTML

{{ telephone | bonjour }}

Résultat

******3456

Les paramètres d'un Pipe

Un Pipe peut recevoir plusieurs paramètres.

Syntaxe

{{ valeur | pipe:param1:param2 }}

Exemple

HTML

{{ nom | bonjour:'Monsieur' }}

Pipe

transform(value: string, titre: string): string {

return "Bonjour " + titre + " " + value;

}

Résultat

Bonjour Monsieur Ahmed

Autre exemple

HTML

{{ nom | bonjour:'Dr' }}

Résultat

Bonjour Dr Ahmed

Plusieurs paramètres

HTML

{{ nom | bonjour:'Dr':'Bienvenue' }}

Pipe

```
transform(
  value: string,
  titre: string,
  message: string
): string {

  return message + " " + titre + " " + value;

}
```

Résultat

Bienvenue Dr Ahmed

- Un **Pipe personnalisé** permet de créer ses propres transformations lorsque les Pipes d'Angular ne répondent pas au besoin.
- Il se crée avec la commande **ng g p nomDuPipe**.
- La méthode **transform()** est le cœur du Pipe : elle reçoit la valeur à transformer et retourne le résultat.
- Un Pipe personnalisé s'utilise exactement comme un Pipe Angular ({{ valeur | nomDuPipe }}).
- Dans une application **Standalone (Angular 21)**, le Pipe doit être importé dans le tableau imports du composant qui l'utilise.
- Un Pipe peut également recevoir un ou plusieurs paramètres pour rendre son comportement plus flexibl

Exercices :

TP 1 – Application de gestion des étudiants

Vous développez une application de gestion d'une école.

Pour rendre l'interface plus agréable, le directeur souhaite que chaque nom d'étudiant soit précédé du texte **Étudiant :**.

Au lieu d'écrire ce texte partout dans le HTML, vous décidez de créer un **Pipe personnalisé**.

Travail demandé

Créer un Pipe nommé :

etudiant

Ce Pipe devra transformer :

Ahmed

en

Étudiant : Ahmed

Données

nom1 = "Ahmed";

nom2 = "Sara";

nom3 = "Ali";

Affichage attendu :

Étudiant : Ahmed

Étudiant : Sara

Étudiant : Ali

TP 2 – Application de gestion des employés

Une entreprise souhaite afficher le poste de chaque employé avant son nom.

Le responsable informatique vous demande de créer un Pipe réutilisable.

Travail demandé

Créer un Pipe nommé :

fonction

Le Pipe recevra un paramètre représentant la fonction.

Exemple

{{ nom | fonction:'Développeur' }}

Affichage attendu:

Développeur : Ahmed

Tester également avec :

Comptable

Chef de projet

Designer

Technicien

TP 3 – Plateforme E-commerce

Vous développez un site de vente en ligne.

Le client souhaite afficher tous les prix suivis du symbole €.

Pour éviter de répéter ce traitement dans tous les composants, vous décidez de créer un Pipe.

Travail demandé

Créer un Pipe nommé :

prix

Données

ordinateur = 1200;

telephone = 850;

tablette = 499;

Affichage attendu

1200 €

850 €

499 €

TP 4 – Gestion des notes

Une école souhaite afficher automatiquement la mention correspondant à la note obtenue par chaque étudiant.

Travail demandé

Créer un Pipe nommé :

mention

Règles

Note	Mention
≥16	Très Bien
≥14	Bien
≥12	Assez Bien

Note	Mention
≥10	Passable
<10	Échec

Données

18

15

13

11

8

Affichage attendu :

Très Bien

Bien

Assez Bien

Passable

Échec

TP 5 – Banque

Une banque souhaite masquer une partie du numéro de compte affiché à l'écran afin de protéger les données des clients.

Seuls les quatre derniers chiffres doivent rester visibles.

Travail demandé

Créer un Pipe nommé :

compte

Données

BE68539007547034

Affichage attendu :

**********7034

TP 6 – Application RH :

Le service des ressources humaines souhaite afficher le statut des employés en fonction de leur âge.

Travail demandé

Créer un Pipe nommé :

age

Règles

Âge	**Résultat**
moins de 18	Mineur
de 18 à 59	Adulte
60 et plus	Senior

Affichage attendu

16 → Mineur

28 → Adulte

65 → Senior

TP 7 – Gestion de stock

Une entreprise vend du matériel informatique.

Le responsable souhaite afficher automatiquement l'état du stock.

Travail demandé

Créer un Pipe nommé :

stock

Règles

Quantité	Affichage
0	Rupture de stock
1 à 5	Stock faible
>5	Disponible

Données

20

4

0

Affichage attendu

Disponible

Stock faible

Rupture de stock

TP 8 – Gestion des commandes

Une plateforme de commerce électronique souhaite informer le client de l'état de sa commande.

Travail demandé

Créer un Pipe nommé :

commande

Règles

Valeur	Résultat
0	En préparation
1	Expédiée
2	Livrée
3	Annulée

Affichage attendu

0

↓

En préparation

TP 9 – Gestion des abonnements

Une plateforme de streaming souhaite afficher le type d'abonnement de ses utilisateurs.

Travail demandé

Créer un Pipe nommé :

abonnement

Règles

Valeur	Résultat
B	Basic
S	Standard
P	Premium

Affichage attendu

B

↓

Basic

TP 10 – Hôpital

Le service des urgences souhaite afficher le niveau de priorité d'un patient selon un score de gravité.

Travail demandé

Créer un Pipe nommé :

priorite

Règles

Score	Priorité
≥90	Urgence absolue
≥70	Urgence élevée
≥40	Urgence normale
<40	Faible priorité

TP 11 – Gestion des salaires

Le directeur souhaite savoir quels employés recevront une prime annuelle.

Travail demandé

Créer un Pipe nommé :

prime

Règles

Salaire	**Résultat**
≥4000 €	Prime accordée
<4000 €	Pas de prime

Affichage attendu

3500 €

↓

Pas de prime

5200 €

↓

Prime accordée

TP 12 – Plateforme de formation

Une plateforme d'apprentissage souhaite afficher le niveau d'un étudiant selon son pourcentage de progression.

Travail demandé

Créer un Pipe nommé :

niveau

Règles

Progression	Résultat
100 %	Formation terminée
≥75 %	Niveau avancé
≥50 %	Niveau intermédiaire
<50 %	Niveau débutant

Projet Final :

Gestion des employés d'une entreprise

Vous êtes développeur Angular dans une entreprise spécialisée dans les ressources humaines.

Votre mission consiste à développer une page permettant d'afficher les informations des employés de manière claire et professionnelle.

Pour éviter de répéter les mêmes traitements dans plusieurs composants, votre responsable vous demande de créer plusieurs **Pipes personnalisés**.

Données

employes = [

{

nom:"Ahmed",

age:35,

salaire:4500,

telephone:"0499123456",

email:"ahmed@gmail.com",

note:18,

stockConge:24

```
},

{
nom:"Sara",
age:22,
salaire:2800,
telephone:"0488456789",
email:"sara@hotmail.com",
note:13,
stockConge:5
},

{
nom:"Ali",
age:17,
salaire:1800,
telephone:"0477123456",
email:"ali@yahoo.com",
note:8,
stockConge:0
}

];
```

Travail demandé

Développez les Pipes suivants :

1. **mentionPipe** : transforme une note en mention (*Très Bien*, *Bien*, *Assez Bien*, *Échec*).
2. **telephonePipe** : masque tous les chiffres sauf les quatre derniers.
3. **emailPipe** : masque une partie de l'adresse e-mail.
4. **primePipe** : indique si l'employé bénéficie d'une prime.
5. **agePipe** : affiche *Mineur*, *Adulte* ou *Senior*.

6. **congePipe** : affiche :
    - **Aucun congé restant** si le solde est de 0.
    - **Peu de jours restants** si le solde est inférieur ou égal à 5.
    - **Solde de congés suffisant** au-delà de 5 jours.

Réponses

TP 1 – Application de gestion des étudiants

Vous êtes développeur Angular dans une école.

Le directeur souhaite afficher le texte **"Étudiant :"** devant le nom de chaque étudiant.

Au lieu d'écrire ce texte dans tous les composants, vous décidez de créer un **Pipe personnalisé**.

Travail demandé

Créer un Pipe nommé :

ng g p etudiant

Le Pipe devra transformer :

Ahmed

en

Étudiant : Ahmed

Étape 1 : Générer le Pipe

Dans le terminal :

ng g p etudiant

Angular crée automatiquement

etudiant.pipe.ts

Étape 2 : Modifier le Pipe

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


@Pipe({
  name: 'etudiant'
```

```
})
export class EtudiantPipe implements PipeTransform {

 transform(value: string): string {

   return "Étudiant : " + value;

 }

}
```

Étape 3 : Importer le Pipe

Angular 21 (Standalone)

```
import { Component } from '@angular/core';
import { EtudiantPipe } from './etudiant.pipe';

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

 nom1="Ahmed";

 nom2="Sara";

 nom3="Ali";

}
```

HTML

<h2>Liste des étudiants</h2>

{{ nom1 | etudiant }}

{{ nom2 | etudiant }}

{{ nom3 | etudiant }}

TP 2 – Gestion des employés

Vous développez une application RH.

Le responsable souhaite afficher la fonction de chaque employé avant son nom.

Au lieu d'écrire ce texte partout dans le HTML, vous créez un Pipe.

Travail demandé

Créer

ng g p fonction

Le Pipe recevra un paramètre.

Exemple

{{ nom | fonction:'Développeur' }}

Affichage :

Développeur : Ahmed

Étape 1

Créer le Pipe

ng g p fonction

Étape 2

Modifier

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

@Pipe({
 name: 'fonction'
})
export class FonctionPipe implements PipeTransform {

 transform(value: string, poste: string): string {

  return poste + " : " + value;

 }

}
```

Composant

```
import { Component } from '@angular/core';
import { FonctionPipe } from './fonction.pipe';

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

 nom="Ahmed";
```

}

HTML

{{ nom | fonction:'Développeur' }}

{{ nom | fonction:'Chef de projet' }}

{{ nom | fonction:'Designer' }}

Un Pipe peut recevoir plusieurs paramètres.

Syntaxe

{{ variable | pipe:parametre }}

TP 3 – Plateforme E-commerce

Vous développez une boutique en ligne.

Le client souhaite afficher automatiquement le symbole € après chaque prix.

Vous décidez de créer un Pipe.

Travail demandé

Créer

ng g p prix

Pipe

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

@Pipe({
  name: 'prix'
})
export class PrixPipe implements PipeTransform {
```

```
 transform(value: number): string {

   return value + " €";

 }

}
```

Composant

```
import { Component } from '@angular/core';
import { PrixPipe } from './prix.pipe';

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

 ordinateur=1200;

 telephone=850;

 tablette=499;

}
```

HTML

```
<h2>Produits</h2>
```

Ordinateur : {{ ordinateur | prix }}

Téléphone : {{ telephone | prix }}

Tablette : {{ tablette | prix }}

Affichage

Ordinateur : 1200 €

Téléphone : 850 €

Tablette : 499 €

TP 4 – Gestion des notes

Vous développez une application pour une université.

Les enseignants ne souhaitent plus afficher uniquement les notes.

Ils veulent également afficher la mention correspondante.

Vous allez créer un Pipe.

Travail demandé

Créer

ng g p mention

Règles

Note	Mention
≥16	Très Bien
≥14	Bien
≥12	Assez Bien
≥10	Passable
<10	Échec

Pipe

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

@Pipe({
 name: 'mention'
})
export class MentionPipe implements PipeTransform {

 transform(note: number): string {

  if(note>=16){

   return "Très Bien";

  }

  if(note>=14){

   return "Bien";

  }

  if(note>=12){

   return "Assez Bien";

  }

  if(note>=10){
```

```
    return "Passable";

  }

  return "Échec";

 }

}
```

Composant

```
import { Component } from '@angular/core';
import { MentionPipe } from './mention.pipe';

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

 note1=18;

 note2=15;

 note3=13;

 note4=11;
```

```
 note5=8;

}
```

HTML

```
<h2>Résultats des étudiants</h2>

<p>{{ note1 }} → {{ note1 | mention }}</p>

<p>{{ note2 }} → {{ note2 | mention }}</p>

<p>{{ note3 }} → {{ note3 | mention }}</p>

<p>{{ note4 }} → {{ note4 | mention }}</p>

<p>{{ note5 }} → {{ note5 | mention }}</p>
```

Affichage :

18 → Très Bien

15 → Bien

13 → Assez Bien

11 → Passable

8 → Échec

TP 5 – Application bancaire : Masquer un numéro de compte

Vous êtes développeur dans une banque.

Pour des raisons de sécurité, le numéro de compte d'un client ne doit jamais être affiché entièrement à l'écran.

Seuls les **4 derniers caractères** doivent rester visibles.

Vous devez créer un Pipe personnalisé qui masque automatiquement le début du numéro de compte.

Travail demandé

Créer un Pipe nommé :

ng g p compte

Le Pipe devra transformer :

BE68539007547034

en

**********7034

Étape 1 : Générer le Pipe

ng g p compte

Angular crée automatiquement :

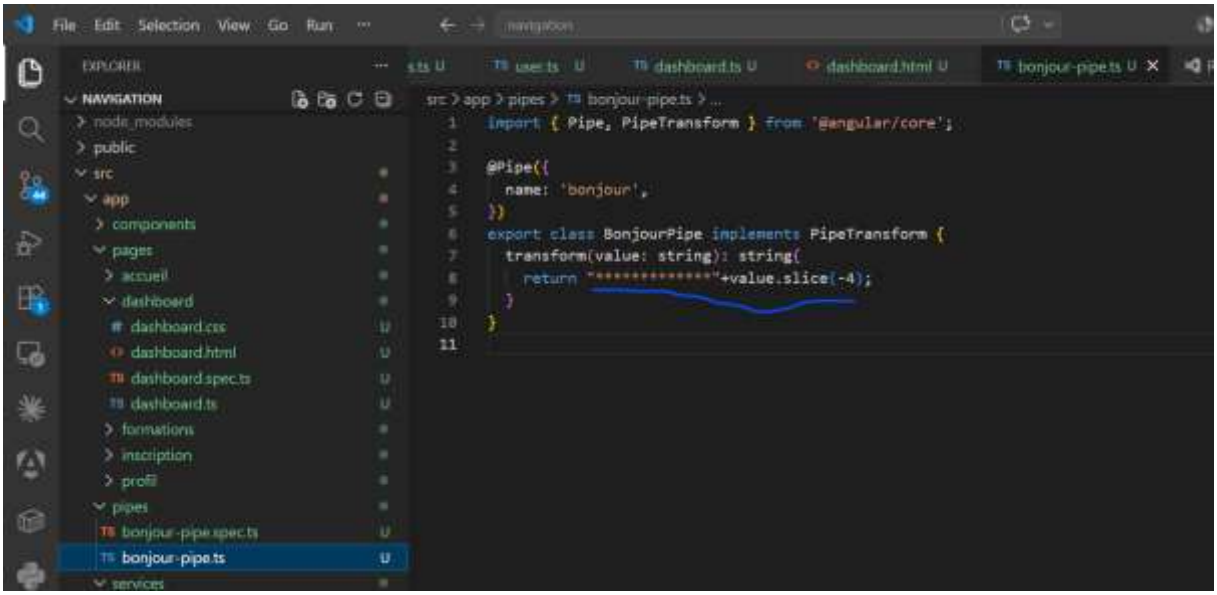
compte.pipe.ts

Étape 2 : Modifier le Pipe

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


@Pipe({
  name: 'compte'
})
export class ComptePipe implements PipeTransform {


  transform(value: string): string {


    return "**********" + value.slice(-4);
```

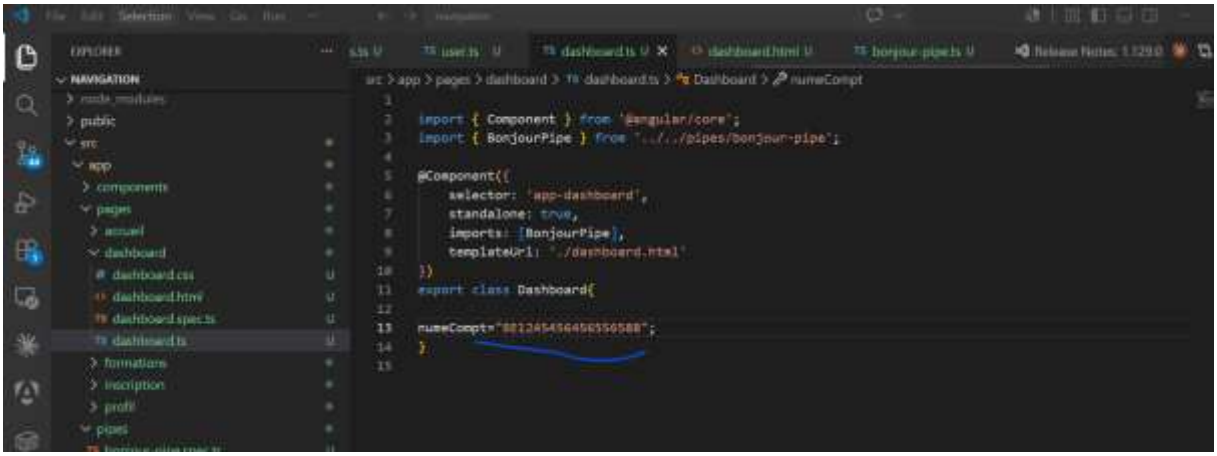
}

}



Composant

```
import { Component } from '@angular/core';
import { ComptePipe } from './compte.pipe';

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

 compte1 = "BE68539007547034";
 compte2 = "BE12345678901234";

}
```

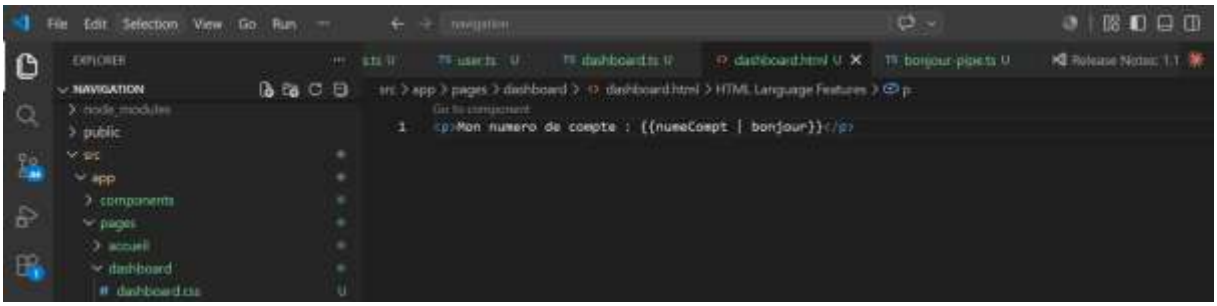


HTML

<h2>Comptes bancaires</h2>

{{ compte1 | compte }}

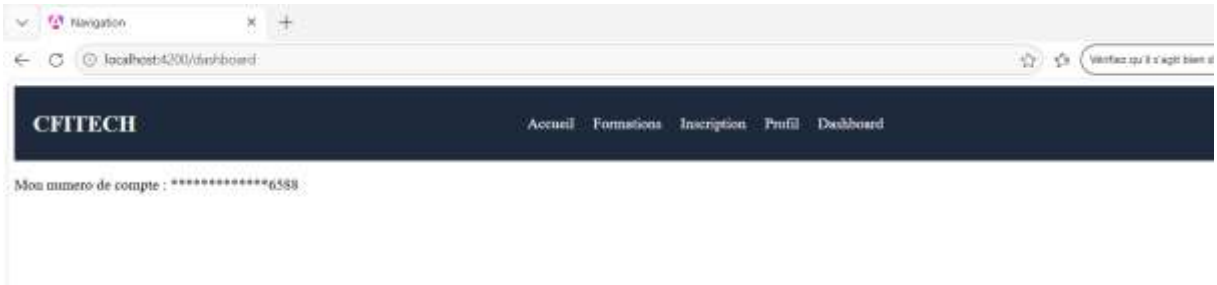
{{ compte2 | compte }}



Affichage :

**********7034

**********1234



Le Pipe protège les informations sensibles.

Cette technique est très utilisée dans les banques, les assurances et les applications de paiement.

TP 6 – Gestion des Ressources Humaines

Une entreprise souhaite afficher automatiquement le statut d'un employé selon son âge.

Vous devez créer un Pipe permettant d'afficher :

- Mineur
- Adulte
- Senior

Travail demandé

Créer un Pipe nommé :

ng g p age

Règles

Âge	Résultat
moins de 18	Mineur
de 18 à 59	Adulte
60 et plus	Senior

Pipe

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


@Pipe({
  name: 'age'
})
export class AgePipe implements PipeTransform {


  transform(age: number): string {
```

```
    if (age < 18) {

      return "Mineur";

    }

    if (age < 60) {

      return "Adulte";

    }

    return "Senior";

  }

}
```

Composant

```
import { Component } from '@angular/core';
import { AgePipe } from './age.pipe';

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

```
  age1 = 16;

  age2 = 35;

  age3 = 67;

}
```

HTML

```
<h2>Employés</h2>

<p>Ahmed : {{ age1 | age }}</p>

<p>Sara : {{ age2 | age }}</p>

<p>Ali : {{ age3 | age }}</p>
```

Affichage :

Ahmed : Mineur

Sara : Adulte

Ali : Senior

TP 7 – Gestion de Stock

Vous développez un logiciel de gestion de stock pour une entreprise.

Le responsable souhaite voir immédiatement si un produit est disponible.

Vous devez créer un Pipe qui interprète automatiquement la quantité restante.

Travail demandé

Créer un Pipe nommé :

ng g p stock

Règles

Quantité	Résultat
0	Rupture de stock
1 à 5	Stock faible
plus de 5	Disponible

Pipe

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

@Pipe({
  name: 'stock'
})
export class StockPipe implements PipeTransform {

  transform(qte: number): string {

    if (qte === 0) {

      return "Rupture de stock";

    }

    if (qte <= 5) {

      return "Stock faible";
```

```
  }

  return "Disponible";

 }

}
```

Composant

```
import { Component } from '@angular/core';
import { StockPipe } from './stock.pipe';

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

 ordinateur = 15;

 telephone = 3;

 imprimante = 0;

}
```

HTML

```
<h2>Stock</h2>

<p>Ordinateur : {{ ordinateur | stock }}</p>

<p>Téléphone : {{ telephone | stock }}</p>

<p>Imprimante : {{ imprimante | stock }}</p>
```

Affichage :

Ordinateur : Disponible

Téléphone : Stock faible

Imprimante : Rupture de stock

TP 8 – Plateforme E-commerce : Suivi des commandes

Vous développez une application de vente en ligne.

Chaque commande possède un statut enregistré sous forme de nombre.

Pour que les utilisateurs comprennent facilement l'état de leur commande, vous allez créer un Pipe.

Travail demandé

Créer un Pipe nommé :

ng g p commande

Règles

Valeur	Résultat
0	En préparation
1	Expédiée

Valeur	Résultat
2	Livrée
3	Annulée

Pipe

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

@Pipe({
 name: 'commande'
})
export class CommandePipe implements PipeTransform {

 transform(etat: number): string {

  switch (etat) {

   case 0:
    return "En préparation";

   case 1:
    return "Expédiée";

   case 2:
    return "Livrée";

   case 3:
    return "Annulée";

   default:
    return "État inconnu";
```

```
  }

 }

}
```

Composant

```
import { Component } from '@angular/core';
import { CommandePipe } from './commande.pipe';

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

 commande1 = 0;

 commande2 = 1;

 commande3 = 2;

 commande4 = 3;

}
```

HTML

```
<h2>Suivi des commandes</h2>

<p>Commande 1 : {{ commande1 | commande }}</p>

<p>Commande 2 : {{ commande2 | commande }}</p>

<p>Commande 3 : {{ commande3 | commande }}</p>

<p>Commande 4 : {{ commande4 | commande }}</p>
```

Affichage :

Commande 1 : En préparation

Commande 2 : Expédiée

Commande 3 : Livrée

Commande 4 : Annulée

TP 9 – Sécuriser l'affichage des adresses e-mail

Vous travaillez pour une plateforme de commerce électronique.

Pour protéger les données personnelles des utilisateurs, le responsable informatique vous demande de ne plus afficher les adresses e-mail en clair.

L'utilisateur doit reconnaître son adresse, mais une partie doit être masquée.

Travail demandé

Créer un Pipe nommé :

ng g p email

Le Pipe devra transformer :

ahmed@gmail.com

en

a******@gmail.com

Étape 1 : Générer le Pipe

ng g p email

Angular crée automatiquement :

email.pipe.ts

Étape 2 : Modifier le Pipe

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

@Pipe({
  name: 'email'
})
export class EmailPipe implements PipeTransform {

  transform(value: string): string {

    const parties = value.split('@');

    return parties[0].charAt(0) + "******@" + parties[1];

  }

}
```

Supposons :

ahmed@gmail.com

Le code

value.split('@')

retourne

["ahmed","gmail.com"]

Puis

parties[0].charAt(0)

retourne

a

Le résultat final devient

a******@gmail.com

Composant

```
import { Component } from '@angular/core';
import { EmailPipe } from './email.pipe';

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

  email1="ahmed@gmail.com";

  email2="sara@hotmail.com";

}
```

HTML

```
<h2>Clients</h2>

<p>{{ email1 | email }}</p>
```

{{ email2 | email }}

Affichage :

a******@gmail.com

s******@hotmail.com

TP 10 – Plateforme de Streaming

Vous développez une plateforme similaire à Netflix.

Chaque utilisateur possède un type d'abonnement enregistré dans la base de données sous forme d'une lettre.

Vous devez afficher un texte compréhensible.

Travail demandé

Créer un Pipe nommé

ng g p abonnement

Règles

Code	Résultat
B	Basic
S	Standard
P	Premium

Pipe

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

@Pipe({
  name: 'abonnement'
})
export class AbonnementPipe implements PipeTransform {
```

```
 transform(value: string): string {

  switch(value){

    case 'B':
      return "Basic";

    case 'S':
      return "Standard";

    case 'P':
      return "Premium";

    default:
      return "Inconnu";

  }

 }

}
```

Composant

```
import { Component } from '@angular/core';
import { AbonnementPipe } from './abonnement.pipe';

@Component({
 selector: 'app-root',
 standalone: true,
 imports: [AbonnementPipe],
 templateUrl: './app.html'
```

```
})
export class AppComponent {

 client1="B";

 client2="S";

 client3="P";

}
```

HTML

```
<h2>Abonnements</h2>

<p>{{ client1 | abonnement }}</p>

<p>{{ client2 | abonnement }}</p>

<p>{{ client3 | abonnement }}</p>
```

Affichage :

Basic

Standard

Premium

TP 11 – Gestion des Primes

Vous développez une application RH.

Le directeur souhaite savoir rapidement quels employés recevront une prime annuelle.

Travail demandé

Créer un Pipe nommé

ng g p prime

Règles

Salaire	Résultat
≥4000	Prime accordée
<4000	Pas de prime

Pipe

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

@Pipe({
 name:'prime'
})
export class PrimePipe implements PipeTransform {

 transform(salaire:number):string{

   if(salaire>=4000){

     return "Prime accordée";

   }

   return "Pas de prime";
```

```
 }

}
```

Composant

```
import { Component } from '@angular/core';
import { PrimePipe } from './prime.pipe';

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

 salaire1=2500;

 salaire2=4200;

}
```

HTML

```
<h2>Primes</h2>

<p>Ahmed : {{ salaire1 | prime }}</p>

<p>Sara : {{ salaire2 | prime }}</p>
```

Affichage :

Ahmed : Pas de prime

Sara : Prime accordée

TP 12 – Plateforme de Formation

Vous développez une plateforme de formation en ligne.

Les responsables pédagogiques souhaitent afficher automatiquement le niveau d'un étudiant selon son pourcentage de progression.

Travail demandé

Créer un Pipe nommé

ng g p niveau

Règles

Progression	Résultat
100 %	Formation terminée
75 à 99 %	Niveau avancé
50 à 74 %	Niveau intermédiaire
0 à 49 %	Niveau débutant

Pipe

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


@Pipe({
 name:'niveau'
})
export class NiveauPipe implements PipeTransform{

 transform(progression:number):string{

  if(progression===100){
```

```
    return "Formation terminée";

  }

  if(progression>=75){

    return "Niveau avancé";

  }

  if(progression>=50){

    return "Niveau intermédiaire";

  }

  return "Niveau débutant";

 }

}
```

Composant

```
import { Component } from '@angular/core';
import { NiveauPipe } from './niveau.pipe';

@Component({
 selector:'app-root',
 standalone:true,
 imports:[NiveauPipe],
 templateUrl:'./app.html'
```

```
})
export class AppComponent{

 progression1=100;

 progression2=82;

 progression3=60;

 progression4=25;

}
```

HTML

```
<h2>Progression des étudiants</h2>

<p>Étudiant 1 : {{ progression1 | niveau }}</p>

<p>Étudiant 2 : {{ progression2 | niveau }}</p>

<p>Étudiant 3 : {{ progression3 | niveau }}</p>

<p>Étudiant 4 : {{ progression4 | niveau }}</p>
```

Affichage :

Étudiant 1 : Formation terminée

Étudiant 2 : Niveau avancé

Étudiant 3 : Niveau intermédiaire

Étudiant 4 : Niveau débutant

Module 3 : Communication entre composants

Objectifs:

- Comprendre pourquoi les composants doivent communiquer.
- Envoyer des données d'un composant parent vers un composant enfant avec input().
- Envoyer des données d'un composant enfant vers un composant parent avec output().
- Utiliser model() pour réaliser une liaison bidirectionnelle (two-way binding) entre composants.
- Accéder à un composant enfant avec ViewChild.
- Réutiliser du contenu HTML grâce à la projection de contenu (<ng-content>).

Plan du module

1. Pourquoi les composants doivent communiquer ?

2. input()

3. output()

4. model()

5. ViewChild

6. Content Projection

7. TP complet : Gestion d'une boutique

Pourquoi les composants doivent communiquer ?

Avant de montrer input(), il faut répondre à une question.

Pourquoi un composant doit-il communiquer avec un autre ?

Prenons une application e-commerce.

AppComponent

├— HeaderComponent

├— MenuComponent

├— ProduitComponent

├— PanierComponent

└— FooterComponent

Chaque partie de l'écran est un composant.

Mais ces composants ne vivent pas isolés.

Ils doivent échanger des informations.

Par exemple :

Le composant **Produit** connaît le prix d'un produit.

Le composant **Panier** doit connaître ce prix lorsqu'on clique sur **Ajouter au panier**.

Les composants doivent donc communiquer.

Les différents sens de communication

Parent

↓

Enfant

→ input()

Enfant

↑

Parent

→ output()

Parent ⇄ Enfant

→ model()

Parent

↓

ViewChild

↓

Enfant

Parent

↓

HTML

↓

ng-content

Ex : Boutique en ligne

On construit une vraie application.

Par exemple

App

contient

Header

Liste des produits

Panier

Footer

Les composants devront communiquer.

Ainsi, les stagiaires comprendront pourquoi Angular a créé :

- input()
- output()
- model()
- ViewChild
- Content Projection

Partie 1 : input()

1. Pourquoi utiliser input() ?

Avant de parler d'Angular, imaginons une situation réelle.

Vous développez une application de vente en ligne.

La page affiche une liste de produits.

PC Portable

1200 €

[Voir le détail]

Téléphone

850 €

[Voir le détail]

Tablette

499 €

[Voir le détail]

Pour éviter de copier le même code plusieurs fois, vous créez un composant :

ProduitComponent

Mais une question se pose.

Comment le composant va connaître :

- le nom ?
- le prix ?
- l'image ?

Puisque chaque produit est différent.

C'est exactement le rôle de **input()**.

Définition

input() est une fonction (API) fournie par Angular, permet au composant parent d'envoyer des données à un composant enfant.

Autrement dit,

le parent **donne** une information à son enfant.

Schéma

Parent

(AppComponent)

|

| input()

↓

ProduitComponent

Le flux est toujours :

Parent

↓

Enfant

Exemple de la vie courante

Imaginez une famille.

Papa

↓

Enfant

Le papa donne :

- un cadeau
- de l'argent
- un conseil

L'enfant reçoit.

L'enfant ne donne rien.

C'est exactement le principe de **input()**.

1er exemple

Nous voulons afficher

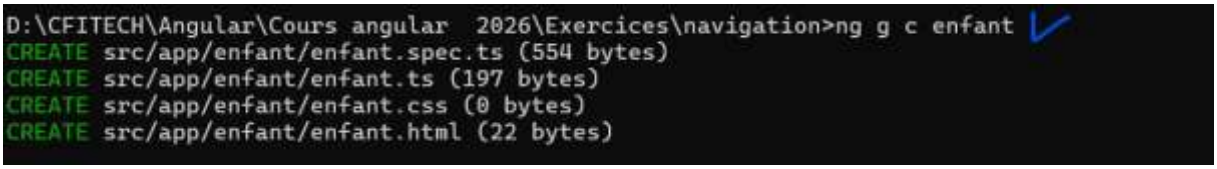
Bonjour Ahmed

grâce à un composant.

Étape 1

Créer un composant

ng g c enfant



Angular crée

enfant.ts

enfant.html

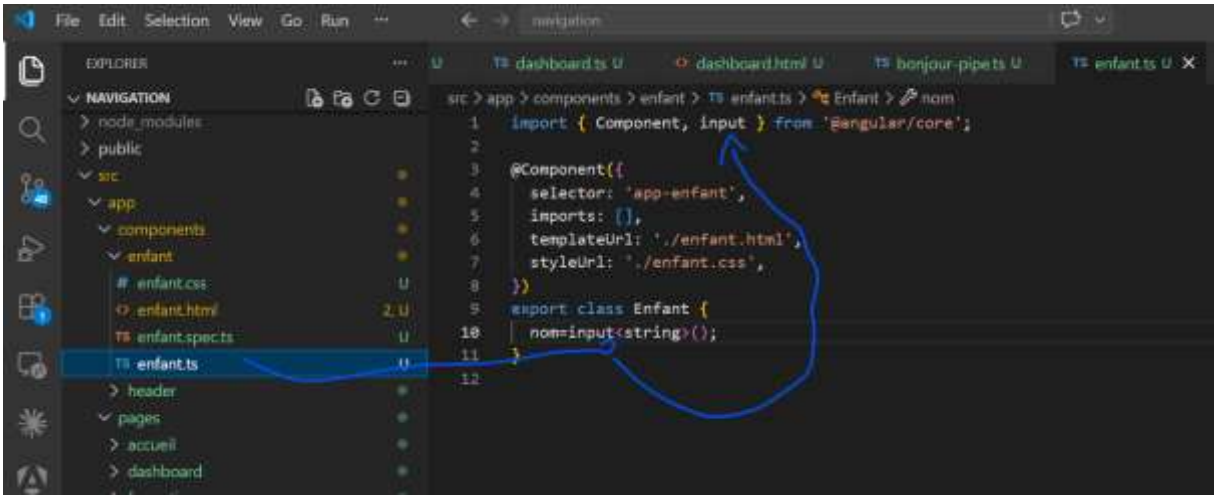
Étape 2

Dans

enfant.ts

écrire

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


@Component({
 selector: 'app-enfant',
 standalone: true,
 templateUrl: './enfant.component.html'
})
export class EnfantComponent {


 nom = input<string>();


}
```



Explication

Cette ligne

nom = input<string>();

signifie

Ce composant attend une information appelée **nom** venant de son parent.

Pour l'instant

il ne connaît aucune valeur.

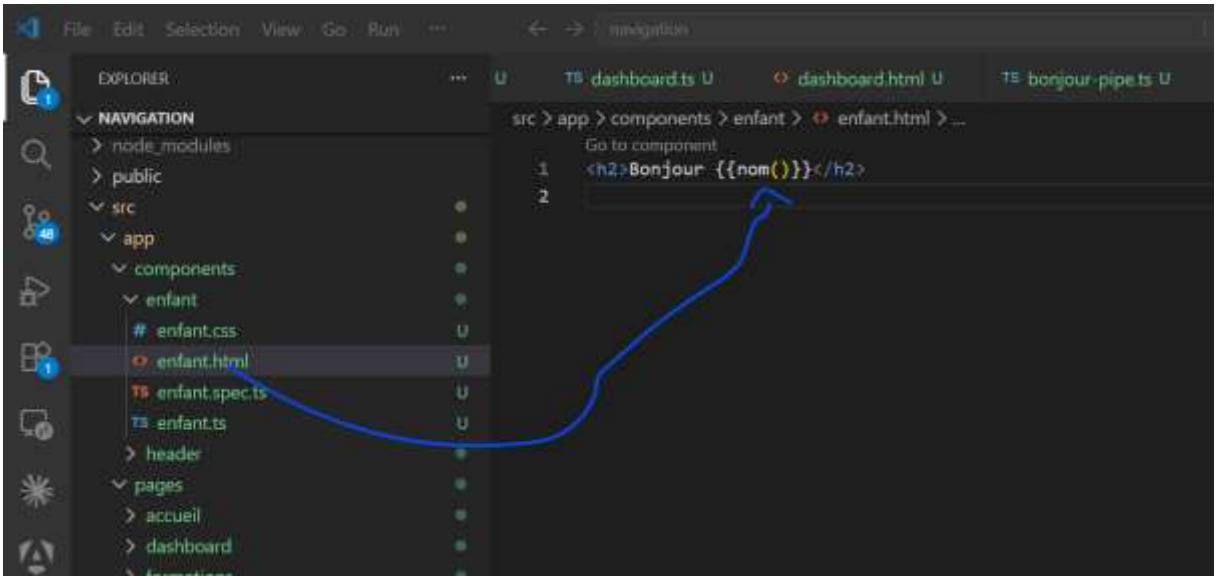
Il attend.

Étape 3

Dans

enfant.html

<h2>Bonjour {{ nom() }}</h2>



Pourquoi

nom()

et pas

nom

?

Parce que depuis Angular 17,

input() retourne un **Input Signal**.

On lit donc sa valeur avec des parenthèses.

Comme un Signal.

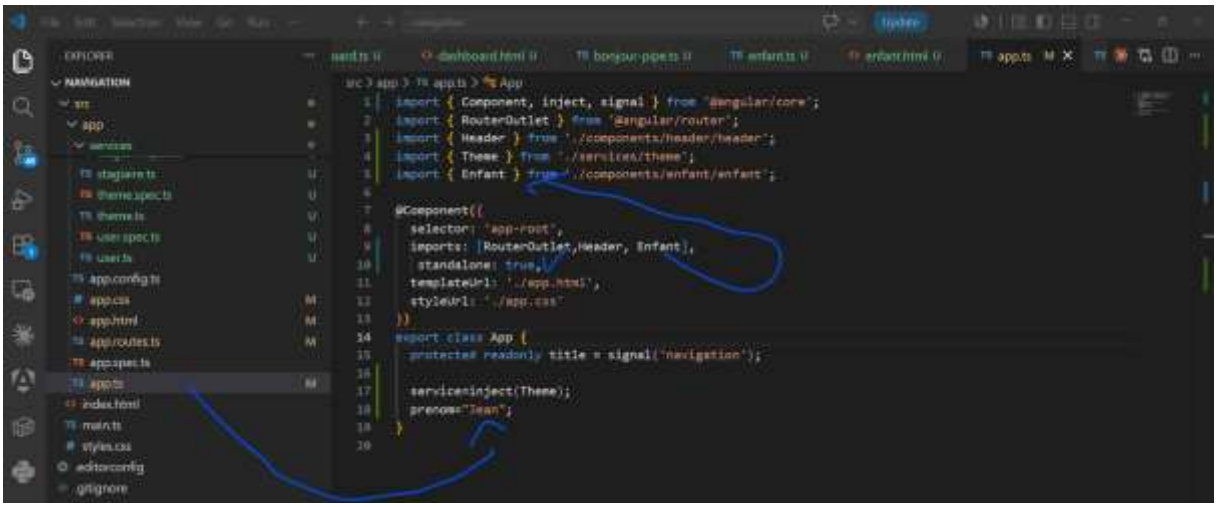
Étape 4

Dans

app.ts

```
import { Component } from '@angular/core';
import { Enfant} from './enfant/enfant.component';

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

```
  templateUrl: './app.component.html'
})
export class AppComponent {

  prenom = "Ahmed";

}
```

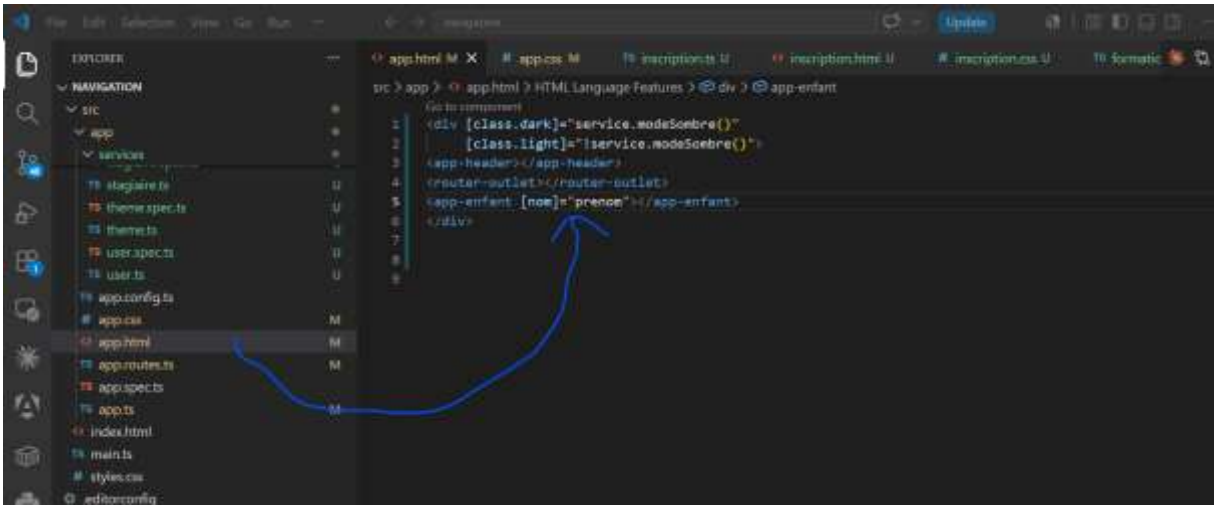


Étape 5

Dans

app.component.html

<app-enfant [nom]="prenom"></app-enfant>



Résultat

Bonjour Ahmed

Que se passe-t-il ?

Le parent possède

prenom="Ahmed";

Le parent envoie

Ahmed

au composant enfant.

L'enfant reçoit

Ahmed

dans

nom=input<string>();

Puis

{{ nom() }}

affiche

Ahmed

Visualisation

AppComponent

prenom="Ahmed"

|

|

[nom]="prenom"

|

▼

EnfantComponent

nom=input()

↓

Ahmed

↓

Bonjour Ahmed

Pourquoi utilise-t-on des crochets ?

Regardons

<app-enfant [nom]="prenom">

Les crochets

[]

signifient

Je n'envoie pas du texte.

J'envoie la valeur d'une variable.

Sans crochets

<app-enfant nom="prenom">

Angular envoie

prenom

(le texte)

Avec les crochets

[nom]="prenom"

Angular envoie

Ahmed

(la valeur de la variable).

Deuxième exemple

Le même composant

mais plusieurs utilisateurs.

<app-enfant [nom]="'Ahmed'"></app-enfant>

<app-enfant [nom]="'Sara'"></app-enfant>

<app-enfant [nom]="'Ali'"></app-enfant>

Résultat

Bonjour Ahmed

Bonjour Sara

Bonjour Ali

Pourquoi créer un composant ?

Sans composant

on écrirait

<h2>Bonjour Ahmed</h2>

<h2>Bonjour Sara</h2>

<h2>Bonjour Ali</h2>

Le code est répété.

Avec un composant

on écrit une seule fois.

Angular le réutilise.

Ex professionnel

Gestion des employés.

Le parent possède

employes=[

{

nom:"Ahmed",

poste:"Développeur"

},

{

nom:"Sara",

poste:"Designer"

},

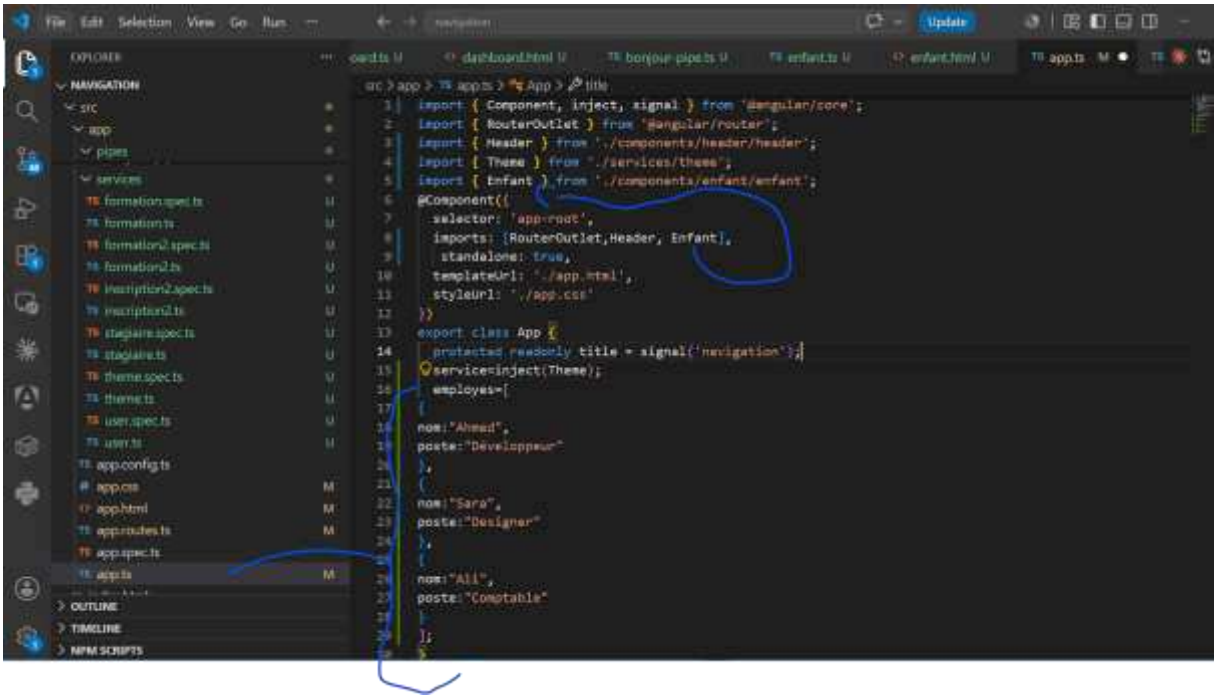
{

nom:"Ali",

poste:"Comptable"

}

];



Le composant

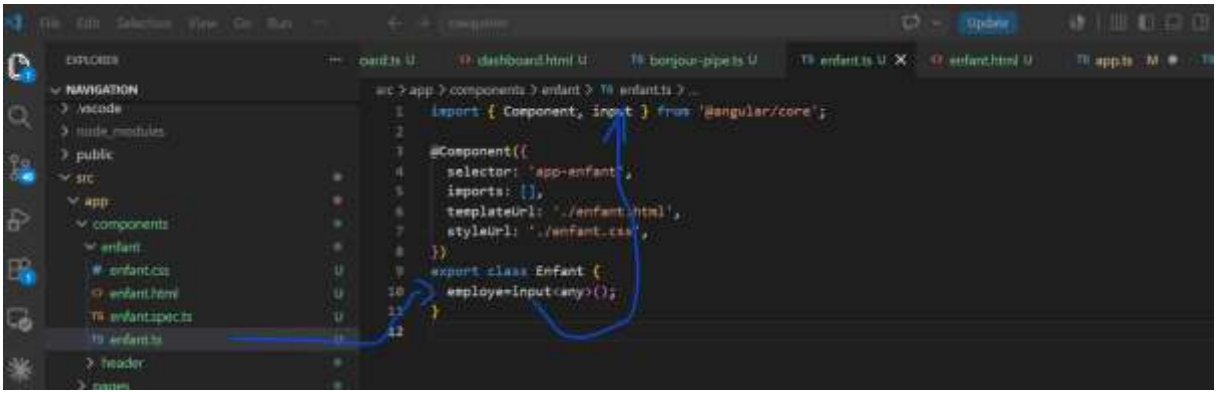
EmployeComponent

reçoit

un employé.

Dans

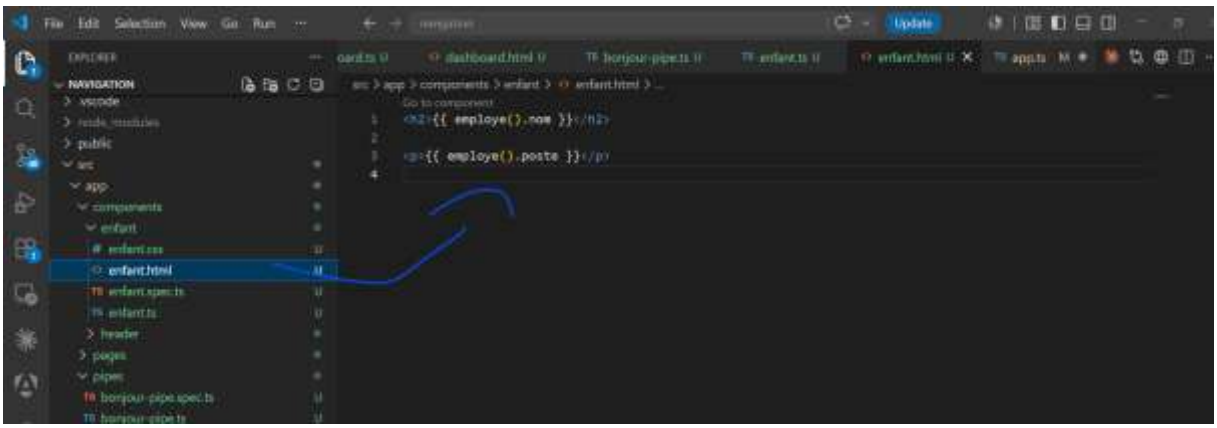
employe=input<any>();



HTML

<h2>{{ employe().nom }}</h2>

{{ employe().poste }}



Parent

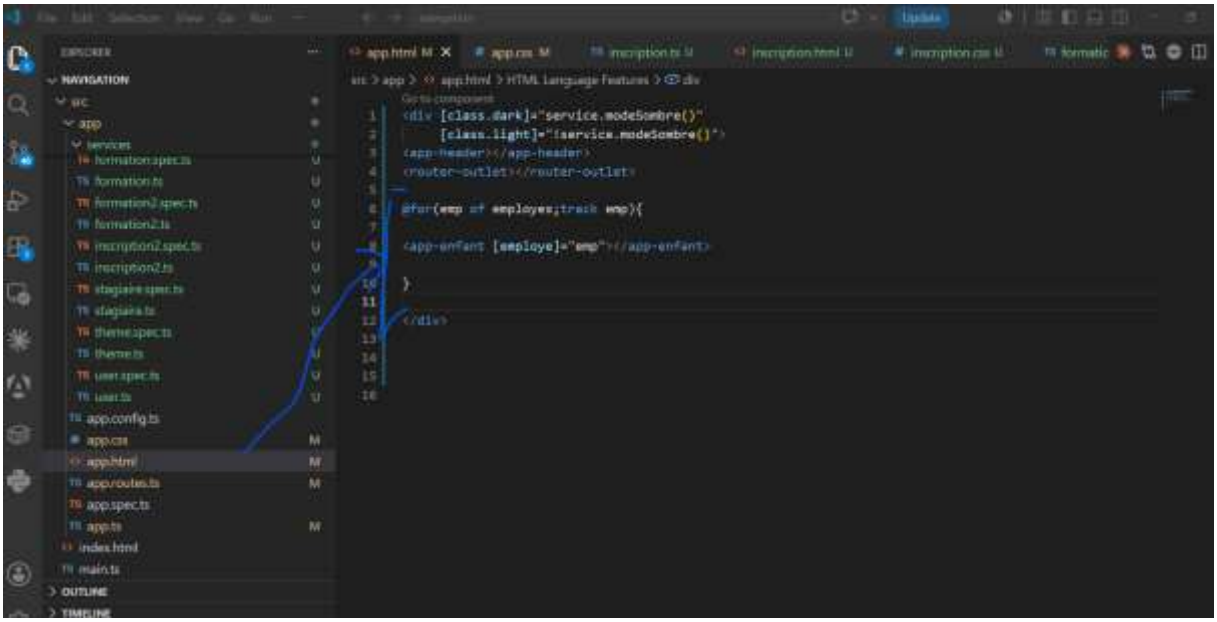
@for(emp of employes;track emp){

<app-employe

[employe]="emp">

</app-employe>

}



Résultat

Ahmed

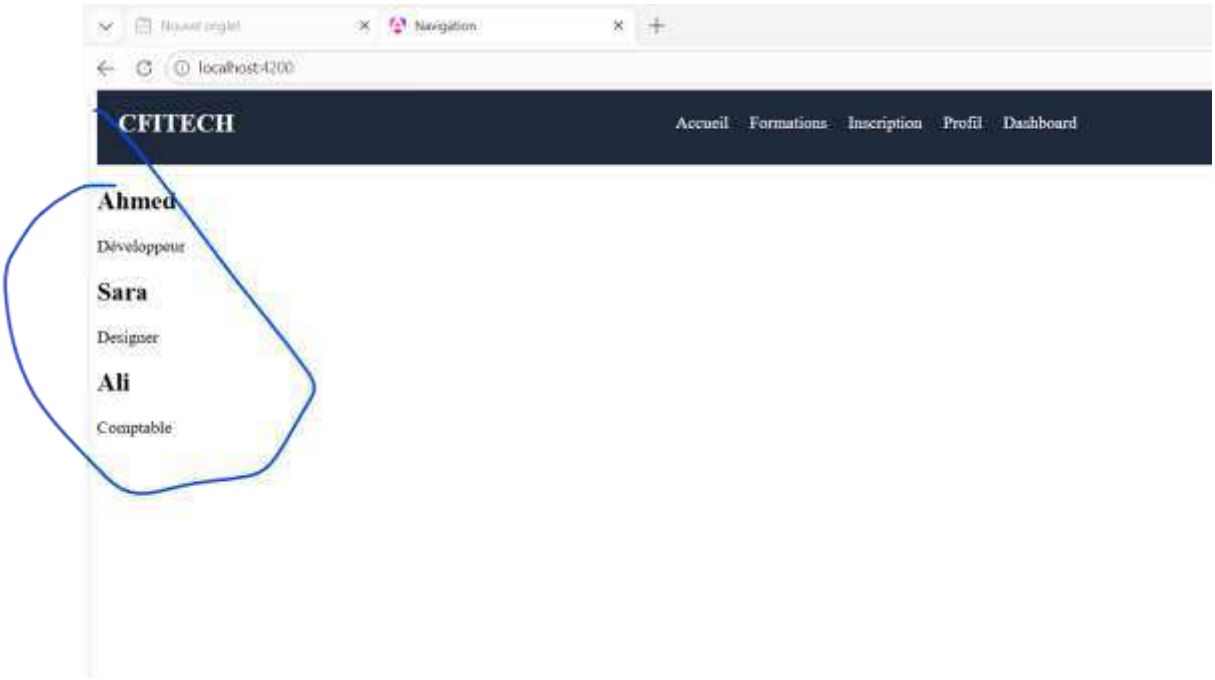
Développeur

Sara

Designer

Ali

Comptable



Exercice 1

Créer un composant

VoitureComponent

Le composant reçoit

marque

et affiche

Marque :

BMW

Tester avec

BMW

Audi

Mercedes

Exercice 2

Créer un composant

VilleComponent

Recevoir

nom

Afficher

Bienvenue à Bruxelles

Bienvenue à Paris

Bienvenue à Madrid

TP : Gestion d'une boutique

Contexte

Vous développez une application e-commerce.

Chaque produit possède :

- un nom
- un prix
- une image

Vous devez créer un composant réutilisable permettant d'afficher un produit.

Travail demandé

Créer un composant :

ng g c produit

Le composant devra recevoir les informations suivantes grâce à input() :

- nom
- prix
- image

Le composant devra afficher :

PC Portable

Prix : 1200 €

[Image]

Le composant devra être réutilisé pour afficher les produits suivants :

Nom	Prix	Image
PC Portable	1200 €	pc.jpg
Téléphone	850 €	telephone.jpg
Tablette	499 €	tablette.jpg

Résultat attendu

PC Portable

Prix : 1200 €

Téléphone

Prix : 850 €

Tablette

Prix : 499 €

À quoi sert réellement input() ?

Lorsque l'on débute avec Angular, il est fréquent de se demander pourquoi utiliser input() alors qu'il existe déjà les formulaires. En réalité, **les formulaires et input() ne répondent pas au même besoin**. Ils interviennent à des moments différents de la vie d'une application.

Un **formulaire** est utilisé lorsque **l'utilisateur doit saisir des informations**. Par exemple, lors d'une inscription, d'une connexion, d'un formulaire de contact ou de la création d'un produit, c'est l'utilisateur qui entre les données dans les champs de saisie. Angular récupère ensuite ces informations grâce à ngModel ou aux Reactive Forms (FormControl, FormGroup, etc.).

Par exemple :

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

Ici, c'est l'utilisateur qui tape son nom.

Utilisateur

Formulaire

Composant Angular

Le formulaire est donc la **porte d'entrée des données dans l'application**.

Que deviennent ensuite ces données ?

Une fois les informations enregistrées (dans une base de données, via une API ou un service Angular), l'utilisateur n'a plus besoin de les saisir une seconde fois. L'application les possède déjà.

À partir de ce moment, le rôle des formulaires est terminé.

C'est ici que **input() devient indispensable**.

Le rôle de input() est de **faire circuler les données entre les composants**. Plus précisément, il permet à un **composant parent** d'envoyer des données à un **composant enfant**.

Le schéma devient alors :

Base de données

Service Angular

Composant Parent

Composant Enfant (input)

Autrement dit, input() ne sert pas à récupérer des informations auprès de l'utilisateur, mais à transmettre des informations déjà existantes à un autre composant.

Ex concret : Facebook

Lorsque vous ouvrez Facebook, l'application affiche votre photo, votre prénom, votre liste d'amis et vos publications.

Est-ce que Facebook vous demande de remplir un formulaire à chaque fois que vous ouvrez votre profil ?

Bien sûr que non.

Vos informations existent déjà dans la base de données de Facebook. Elles sont récupérées par l'application puis envoyées aux différents composants.

Par exemple :

Serveur Facebook

AppComponent

ProfilComponent

Le composant ProfilComponent reçoit les informations grâce à input().

utilisateur = input<User>();

Puis il les affiche :

<h2>{{ utilisateur().nom }}</h2>

<img [src]="utilisateur().photo">

Le composant ne connaît pas le nom de l'utilisateur. Il reçoit simplement les données envoyées par son parent.

Ex concret : Amazon

Lorsque vous consultez Amazon, vous voyez immédiatement une liste de produits.

PC Portable

799 €

Clavier

39 €

Souris

19 €

L'utilisateur n'a rempli aucun formulaire pour afficher ces produits.

Les produits ont déjà été récupérés depuis la base de données. Le composant parent les transmet ensuite au composant ProduitComponent.

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

```
  <app-produit [produit]="produit"></app-produit>
}
```

Chaque carte produit reçoit un objet différent grâce à input().

Le composant ProduitComponent peut alors afficher :

- le nom,
- le prix,
- la photo,
- la description.

Le même composant est réutilisé des centaines de fois avec des données différentes.

Ex concret : YouTube

Lorsque vous ouvrez YouTube, vous voyez immédiatement plusieurs vidéos.

Vidéo 1

Vidéo 2

Vidéo 3

...

Encore une fois, aucun formulaire n'est utilisé.

Les vidéos sont récupérées depuis le serveur puis envoyées au composant VideoComponent.

```
@for(video of videos; track video.id) {
  <app-video [video]="video"></app-video>
}
```

Le composant reçoit les informations grâce à input().

Exemple concret : une application de gestion d'école

Imaginons une application de gestion des étudiants.

Étape 1 : inscription

Le secrétaire remplit un formulaire.

Nom : Ahmed

Prénom : Ali

Classe : BTS 2

Le formulaire sert uniquement à enregistrer les informations.

Étape 2 : consultation

Le directeur ouvre la liste des étudiants.

Ahmed

Sara

Ali

Jean

Il ne remplit aucun formulaire.

Les données sont déjà enregistrées.

Le composant parent affiche alors :

@for (e of etudiants; track e.id) {

<app-etudiant [etudiant]="e"></app-etudiant>

}

Chaque composant reçoit un étudiant différent grâce à input().

Étape 3 : affichage du profil

Lorsque le directeur clique sur Ahmed, le composant parent envoie l'étudiant sélectionné au composant de profil.

<app-profil [etudiant]="selection"></app-profil>

Le composant ProfilComponent affiche ensuite toutes les informations sans utiliser de formulaire.

Quand utilise-t-on un formulaire ?

On utilise un formulaire lorsque **l'utilisateur doit saisir des informations**.

Exemples :

- Connexion
- Inscription
- Formulaire de contact
- Création d'un produit
- Ajout d'un étudiant
- Modification d'un profil
- Recherche

Dans tous ces cas, les données proviennent de l'utilisateur.

Quand utilise-t-on input() ?

On utilise input() lorsque **les données existent déjà** et qu'il faut simplement les transmettre d'un composant à un autre.

Exemples :

- Afficher le profil d'un utilisateur.
- Afficher une carte produit.
- Afficher une vidéo YouTube.
- Afficher un étudiant.
- Afficher une facture.
- Afficher un article de blog.
- Afficher les informations d'une commande.
- Afficher une météo.

Dans tous ces cas, les données proviennent généralement :

- d'une base de données,
- d'une API REST,
- d'un service Angular,
- ou d'un composant parent.

Résumé

Le formulaire et input() sont deux mécanismes complémentaires.

Le **formulaire** permet de faire entrer les données dans l'application. Il intervient lorsque l'utilisateur saisit des informations.

input(), quant à lui, intervient **après** que les données existent déjà. Son rôle est de permettre au composant parent d'envoyer ces données à un composant enfant afin qu'elles puissent être affichées ou réutilisées.

En résumé :

- **Formulaire** → communication **Utilisateur → Application**.
- **input()** → communication **Composant parent → Composant enfant**.

C'est cette séparation des responsabilités qui rend les applications Angular modulaires, réutilisables et faciles à maintenir.

Partie 2 : output()

Objectifs

- Comprendre pourquoi output() existe.
- Comprendre la différence entre input() et output().
- Créer un événement personnalisé.
- Envoyer des données d'un composant enfant vers un composant parent.
- Utiliser output() avec plusieurs exemples concrets.

1. Qu'est-ce que output() ?

Dans Angular, **output() est également une fonction (API) fournie par Angular, tout comme input(), qui permet à un composant enfant d'envoyer un événement au composant parent**.

Autrement dit :

- input() : le parent envoie des données à l'enfant.
- output() : l'enfant envoie un message (ou des données) au parent.

On parle alors de **communication Enfant → Parent**.

Pourquoi output() existe-t-il ?

Lorsqu'on utilise input(), le composant enfant reçoit des informations.

Par exemple :

<app-profil [nom]="prenom"></app-profil>

Le parent dit :

Voici ton prénom.

L'enfant affiche simplement cette information.

Mais imaginons maintenant que le composant enfant possède un bouton.

<button>Supprimer</button>

Lorsque l'utilisateur clique dessus :

Qui doit supprimer l'utilisateur ?

Le composant enfant ?

Non.

C'est généralement le composant parent qui possède la liste des utilisateurs.

L'enfant doit donc prévenir le parent.

C'est exactement le rôle de output().

2. Comparaison entre input() et output()

input()	**output()**
Parent → Enfant	Enfant → Parent
Transmet des données	Transmet un événement
Le parent donne une information	L'enfant avertit le parent
Exemple : afficher un nom	Exemple : bouton Supprimer

3. Analogie de la vie courante

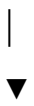
Imaginez un professeur et un étudiant.

Le professeur dit :

Fais l'exercice numéro 2.

L'information va :

Professeur



Étudiant

C'est exactement le rôle de input().

Une fois l'exercice terminé, l'étudiant répond :

Monsieur, j'ai terminé.

L'information repart dans l'autre sens.

Étudiant

Professeur

C'est exactement le rôle de output().

4. Exemple concret : Amazon

Chaque produit possède un bouton.

PC Portable

799 €

[Ajouter au panier]

Qui gère le panier ?

Le composant Produit ?

Non.

Le panier appartient au composant parent.

Le composant enfant fait simplement :

Le client a cliqué sur "Ajouter au panier".

Le parent reçoit cet événement et ajoute le produit.

5. Exemple concret : Facebook

Chaque publication possède un bouton.

J'aime

Lorsque l'utilisateur clique dessus :

Le composant Publication prévient le composant Parent.

Le parent incrémente ensuite le nombre de likes.

6. Ex concret : Gestion des étudiants

Chaque étudiant possède un bouton.

Ahmed

[Supprimer]

Lorsque l'utilisateur clique :

L'enfant dit :

Le bouton Supprimer vient d'être cliqué.

Le parent reçoit l'information puis supprime Ahmed de la liste.

7. Comment fonctionne output() ?

Le fonctionnement se déroule en quatre étapes.

Étape 1

Le parent affiche le composant enfant.

<app-enfant></app-enfant>

Étape 2

L'enfant crée un événement.

supprimer = output<void>();

Ici, on déclare un événement nommé supprimer.

Étape 3

Lorsque l'utilisateur clique :

<button (click)="supprimer.emit()">

  Supprimer

</button>

L'enfant envoie un événement.

La méthode **emit()** sert à **envoyer (émettre) un événement** au composant parent(**émettre**, **envoyer**, **diffuser**.).

Étape 4

Le parent écoute cet événement.

```
<app-enfant
  (supprimer)="supprimerUtilisateur()">
</app-enfant>
```

Dès que l'enfant déclenche l'événement, le parent exécute :

```
supprimerUtilisateur(){
  console.log("Utilisateur supprimé");
}
```

8. output() ou EventEmitter ?

Ancienne méthode (Angular <17)

Pendant longtemps, Angular utilisait :

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

Puis

```
this.supprimer.emit();
```

Nouvelle méthode (Angular 17+)

Aujourd'hui on utilise :

```
supprimer = output<void>();
```

Puis

```
this.supprimer.emit();
```

Le fonctionnement est identique.

La syntaxe est simplement plus moderne.

9. Qu'est-ce qu'un événement ?

Un événement est simplement un message.

Exemples :

- bouton cliqué

- utilisateur connecté
- produit ajouté
- étudiant supprimé
- formulaire envoyé
- vidéo terminée
- paiement effectué

output() sert justement à envoyer ces messages.

10. Envoyer uniquement un événement

valider = output<void>();

<button (click)="valider.emit()">

Valider

</button>

Le parent :

<app-enfant

(valider)="confirmation()">

</app-enfant>

11. Envoyer une donnée

On peut également envoyer une information.

nom = output<string>();

Puis

this.nom.emit("Ahmed");

Le parent reçoit :

<app-enfant

(nom)="afficherNom($event)">

</app-enfant>

afficherNom(nom:string){

console.log(nom);

}

Console

Ahmed

12. Que représente $event ?

$event contient la donnée envoyée par l'enfant.

Exemple :

L'enfant :

selection = output<string>();

selection.emit("Angular");

Le parent :

<app-enfant

(selection)="recevoir($event)">

</app-enfant>

Ici,

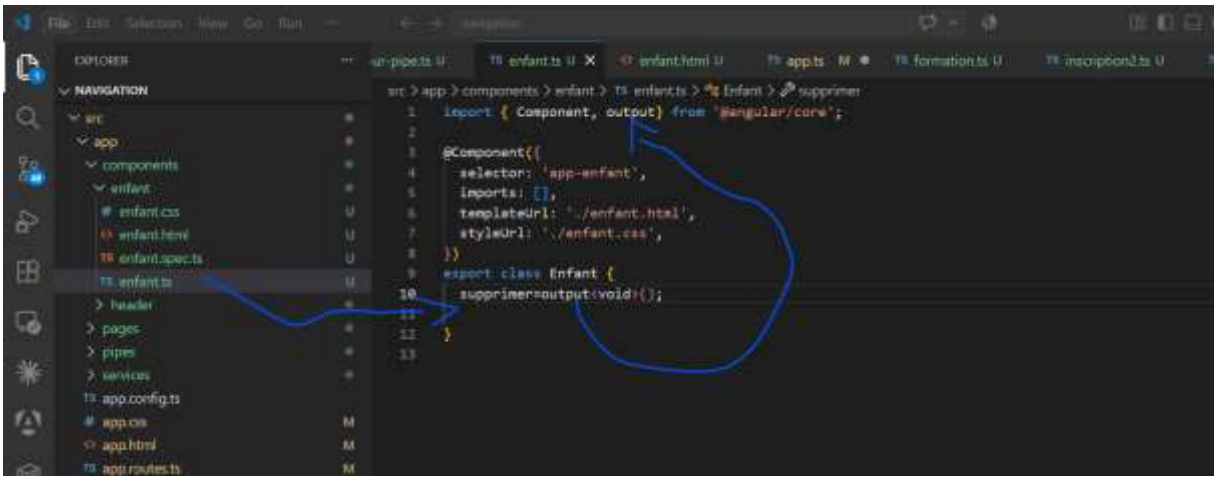
$event = "Angular"

13. Exemple complet

Enfant

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

@Component({
  selector:'app-enfant',
  standalone:true,
  templateUrl:'./enfant.html'
})
export class Enfant{

  supprimer = output<void>();
```

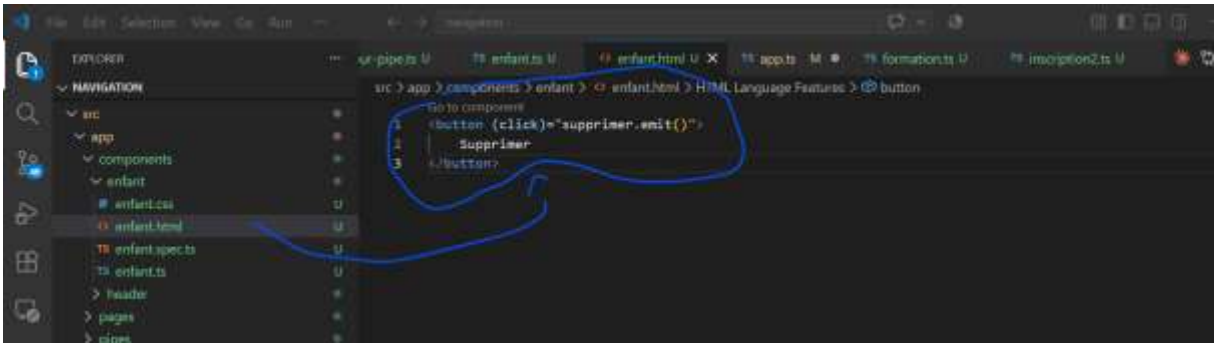
}



<button (click)="supprimer.emit()">

Supprimer

</button>

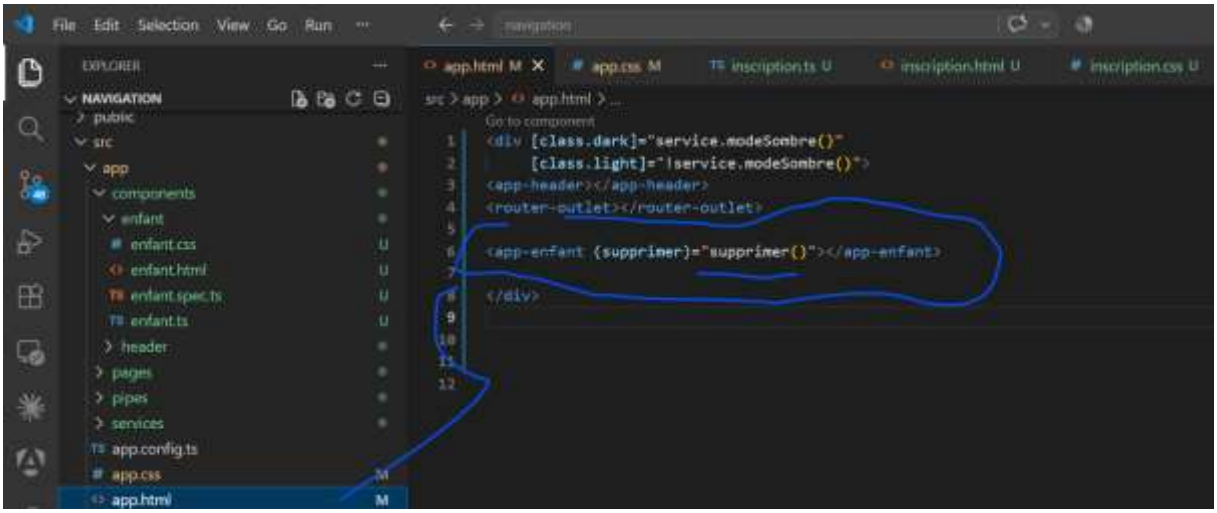


Parent

<app-enfant

(supprimer)="supprimerEtudiant()">

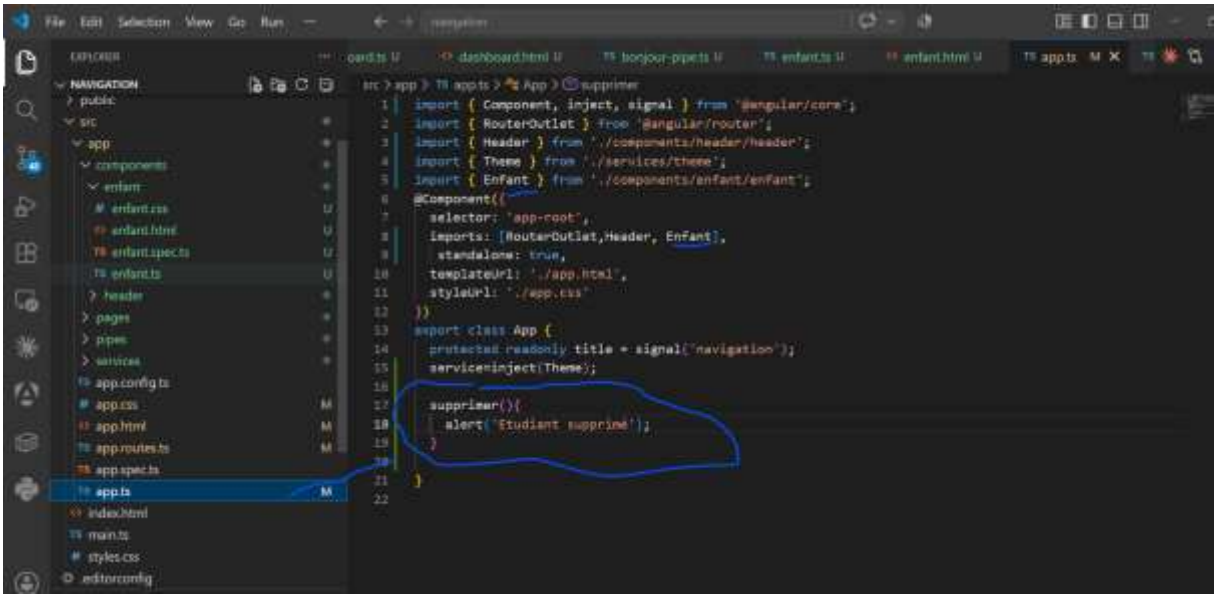
</app-enfant>



supprimerEtudiant(){

alert("Étudiant supprimé");

}



14. Exemple avec transmission d'un objet

Enfant

etudiant = output<any>();

supprimer(){

this.etudiant.emit({

id:1,

nom:"Ahmed"

});

}

Parent

<app-enfant

(etudiant)="supprimer($event)">

</app-enfant>

supprimer(e:any){

console.log(e);

}

Console

{

id:1,

nom:"Ahmed"

}

input()	**output()**
Parent → Enfant	Enfant → Parent
Reçoit des données	Envoie un événement
[nom]="prenom"	(supprimer)="..."

input()	**output()**
Utilise input()	Utilise output()

TP : Gestion d'une liste de stagiaires

Créer deux composants :

- ListeStagiaires (parent)
- CarteStagiaire (enfant)

Le composant parent possède une liste de stagiaires.

Chaque stagiaire est affiché grâce au composant enfant.

Le composant enfant affiche :

- Nom
- Formation
- Un bouton **Supprimer**

Lorsque l'utilisateur clique sur **Supprimer**, le composant enfant ne supprime pas lui-même le stagiaire. Il envoie un événement au parent grâce à output().

Le parent reçoit l'événement, identifie le stagiaire concerné grâce à $event, puis le retire de la liste.

Affichage attendu

Avant :

Ahmed [Supprimer]

Sara [Supprimer]

Jean [Supprimer]

Après un clic sur **Supprimer** pour "Sara" :

Ahmed [Supprimer]

Jean [Supprimer]

Taches :

- Créer un composant parent et un composant enfant.
- Utiliser input() pour transmettre un objet stagiaire du parent vers l'enfant.
- Utiliser output() pour envoyer un événement de suppression de l'enfant vers le parent.
- Utiliser $event pour récupérer les données envoyées.
- Modifier une liste avec filter() ou splice() après réception de l'événement.

Étape 1 : Créer les composants

Créer les deux composants suivants :

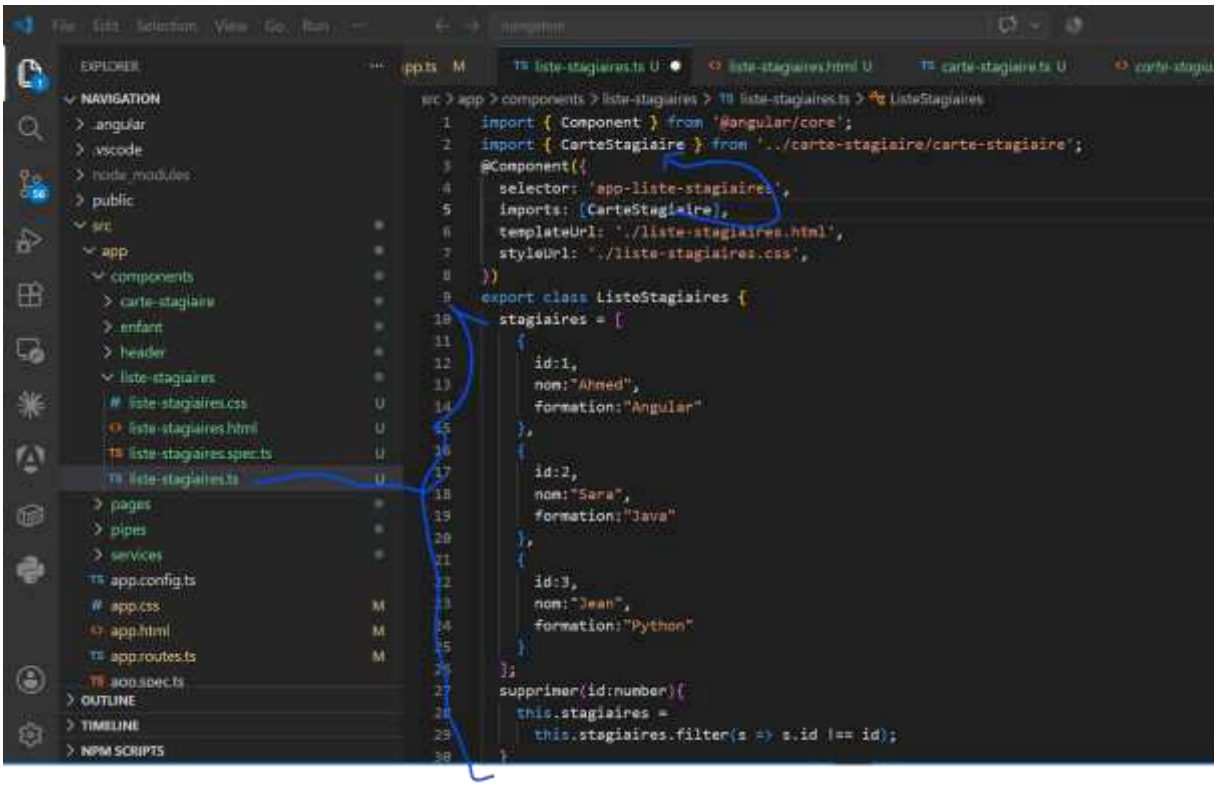
ng generate component components/liste-stagiaires

ng generate component components/carte-stagiaire

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g c components/liste-stagiaires
CREATE src/app/components/liste-stagiaires/liste-stagiaires.spec.ts (618 bytes)
CREATE src/app/components/liste-stagiaires/liste-stagiaires.ts (236 bytes)
CREATE src/app/components/liste-stagiaires/liste-stagiaires.css (0 bytes)
CREATE src/app/components/liste-stagiaires/liste-stagiaires.html (32 bytes)

D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g c components/carte-stagiaire
CREATE src/app/components/carte-stagiaire/carte-stagiaire.spec.ts (611 bytes)
CREATE src/app/components/carte-stagiaire/carte-stagiaire.ts (232 bytes)
CREATE src/app/components/carte-stagiaire/carte-stagiaire.css (0 bytes)
CREATE src/app/components/carte-stagiaire/carte-stagiaire.html (31 bytes)

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

Étape 2 : Créer la liste des stagiaires (Parent)

Dans **liste-stagiaires.ts**



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

import { CarteStagiaire } from '../carte-stagiaire/carte-stagiaire';

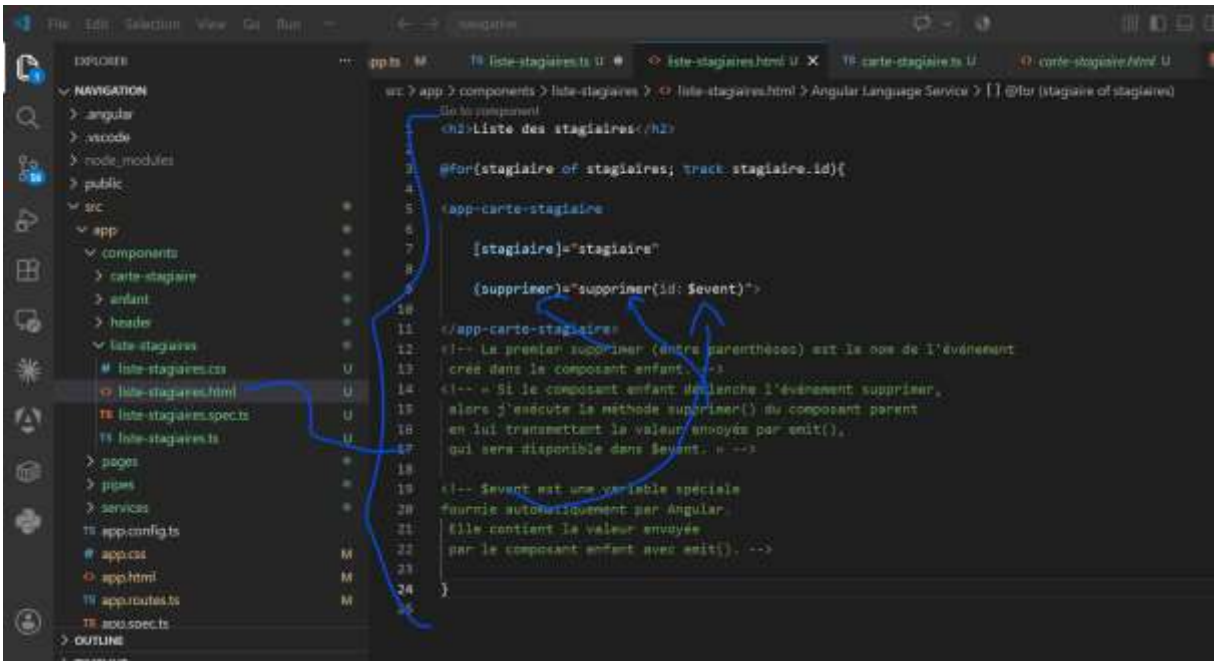
@Component({

selector: 'app-liste-stagiaires',

```
 imports: [CarteStagiaire],
 templateUrl: './liste-stagiaires.html',
 styleUrl: './liste-stagiaires.css',
})
export class ListeStagiaires {
 stagiaires = [
  {
   id:1,
   nom:"Ahmed",
   formation:"Angular"
  },
  {
   id:2,
   nom:"Sara",
   formation:"Java"
  },
  {
   id:3,
   nom:"Jean",
   formation:"Python"
  }
 ];
 supprimer(id:number){
  this.stagiaires =
   this.stagiaires.filter(s => s.id !== id);
 }
}
```

Étape 3 : Afficher les stagiaires

Dans **liste-stagiaires.html**



<h2>Liste des stagiaires</h2>

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

<app-carte-stagiaire

[stagiaire]="stagiaire"

(supprimer)="supprimer($event)">

</app-carte-stagiaire>

<!-- Le premier supprimer (entre parenthèses) est le nom de l'événement
créé dans le composant enfant. -->

<!-- « Si le composant enfant déclenche l'événement supprimer,
alors j'exécute la méthode supprimer() du composant parent
en lui transmettant la valeur envoyée par emit(),
qui sera disponible dans $event. » -->

<!-- $event est une variable spéciale

fournie automatiquement par Angular.

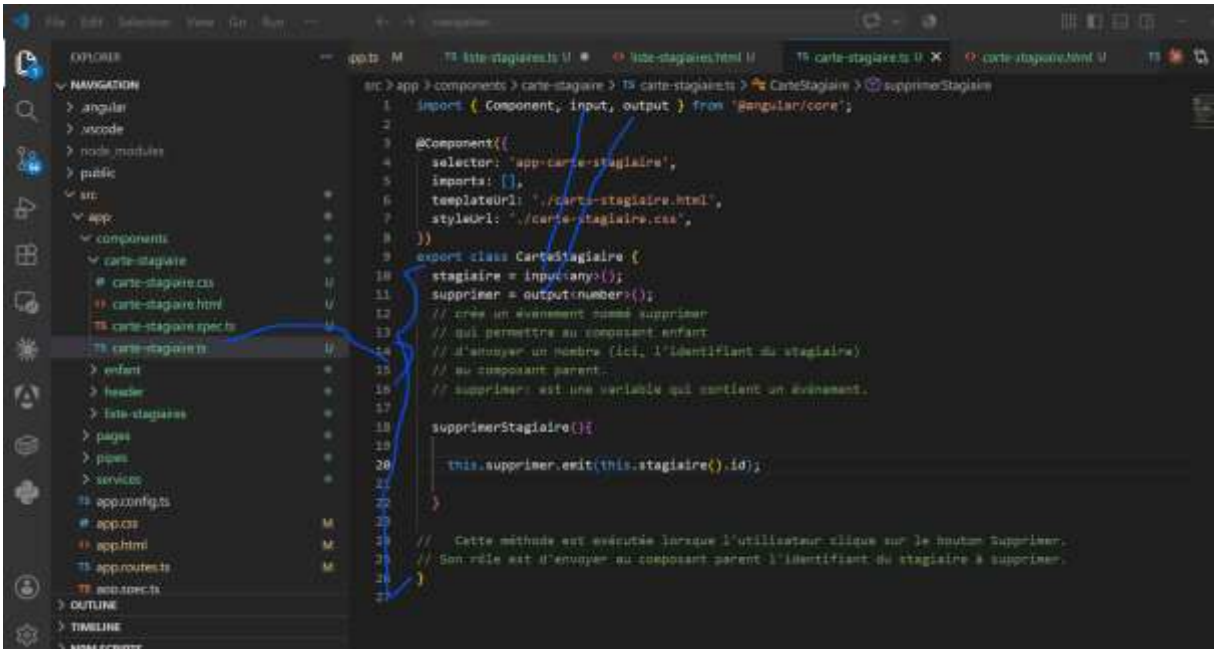
Elle contient la valeur envoyée

par le composant enfant avec emit(). -->

}

Étape 4 : Créer le composant enfant

Dans **carte-stagiaire.ts**



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

@Component({

selector: 'app-carte-stagiaire',

imports: [],

templateUrl: './carte-stagiaire.html',

styleUrl: './carte-stagiaire.css',

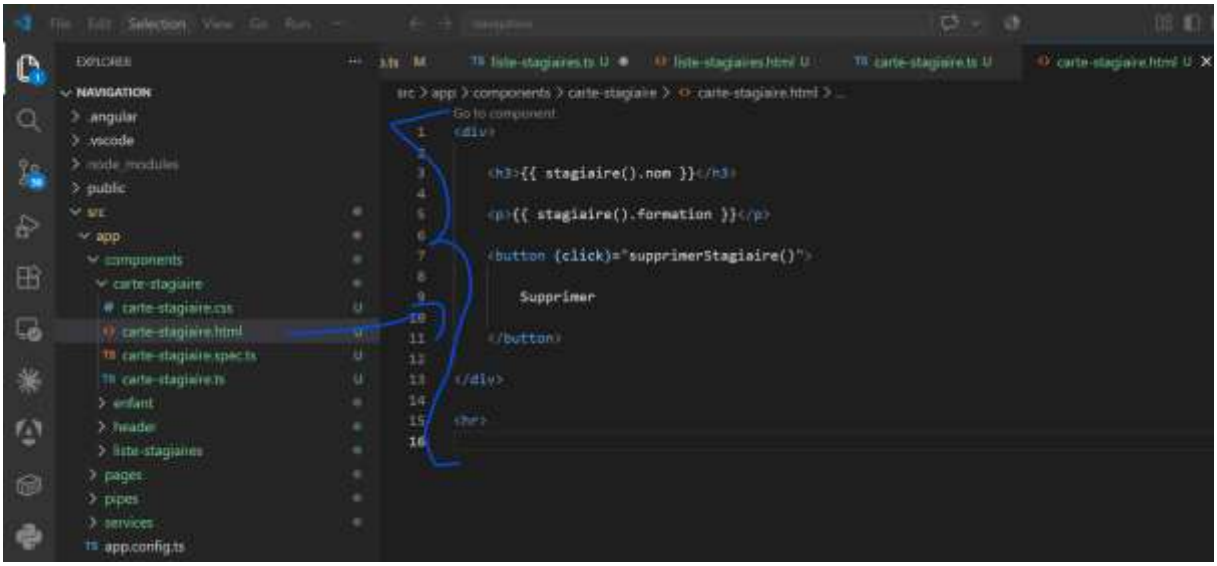
})

export class CarteStagiaire {

```
 stagiaire = input<any>();
 supprimer = output<number>();
 // crée un événement nommé supprimer
 // qui permettra au composant enfant
 // d'envoyer un nombre (ici, l'identifiant du stagiaire)
 // au composant parent.
 // supprimer: est une variable qui contient un événement.

 supprimerStagiaire(){

   this.supprimer.emit(this.stagiaire().id);

 }

//   Cette méthode est exécutée lorsque l'utilisateur clique sur le bouton Supprimer.
// Son rôle est d'envoyer au composant parent l'identifiant du stagiaire à supprimer.
}
```

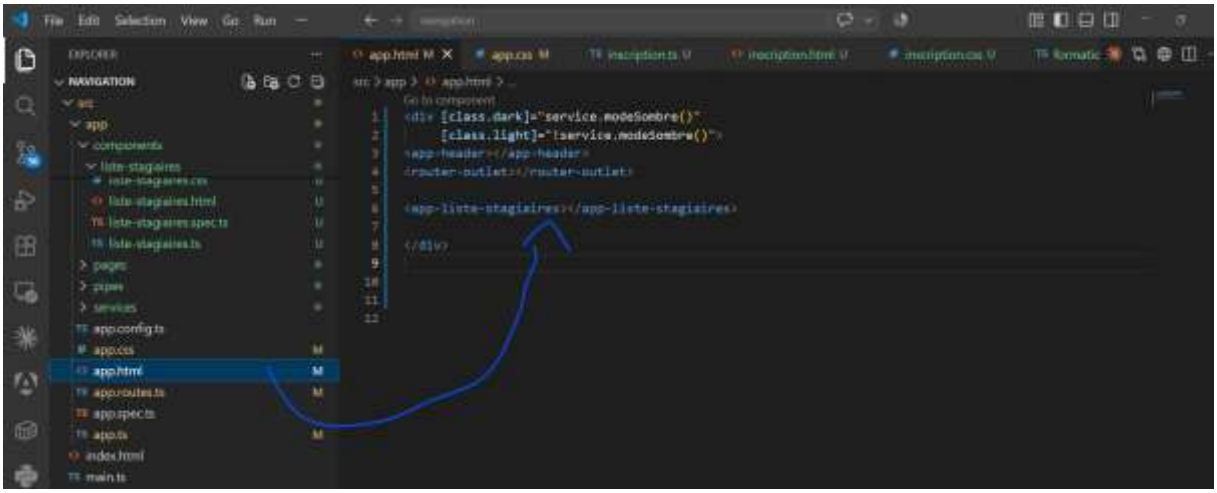
Étape 5 : Afficher une carte

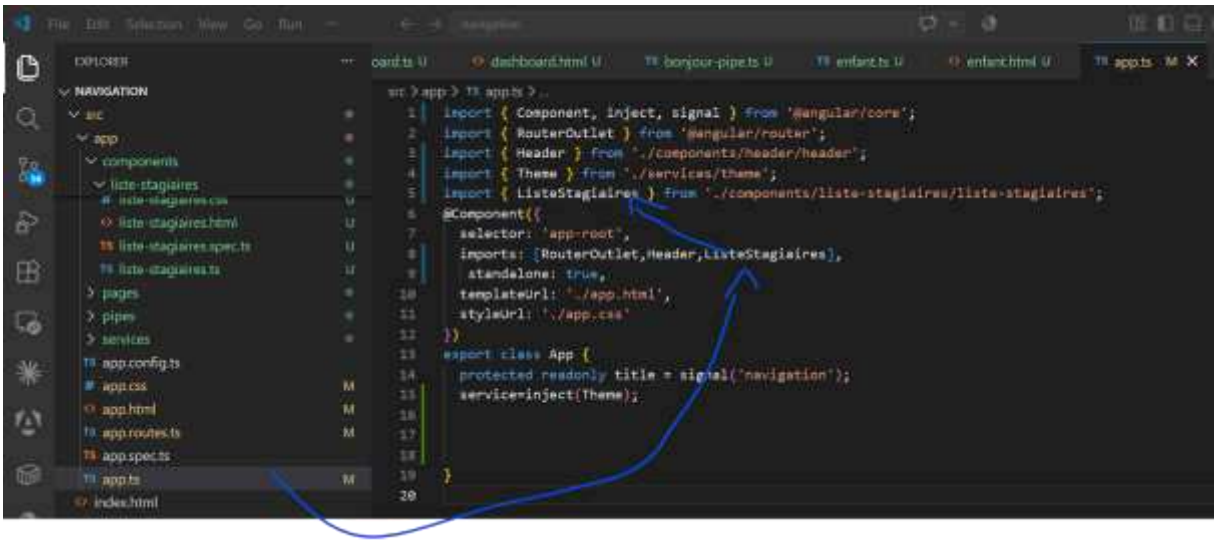
Dans **carte-stagiaire.html**




```
<div>

  <h3>{{ stagiaire().nom }}</h3>

  <p>{{ stagiaire().formation }}</p>

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

    Supprimer

  </button>

</div>

<hr>
```

Dans le composant principal :





Partie 3 : model()

Ex concret : Modifier le prénom d'un utilisateur

Nous avons :

- un **composant Parent** qui possède le prénom de l'utilisateur ;
- un **composant Enfant** qui affiche un champ de saisie pour modifier ce prénom.

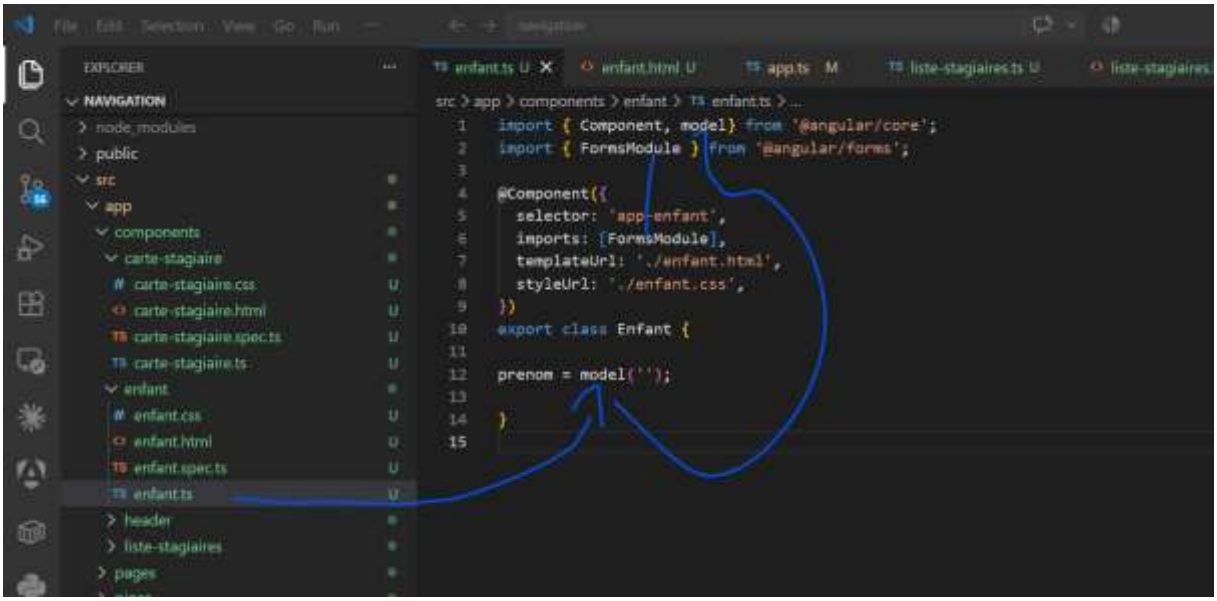
L'objectif est que **lorsque l'enfant modifie le prénom, le parent soit automatiquement mis à jour.**

Avec model()

Angular fait tout automatiquement.

Enfant

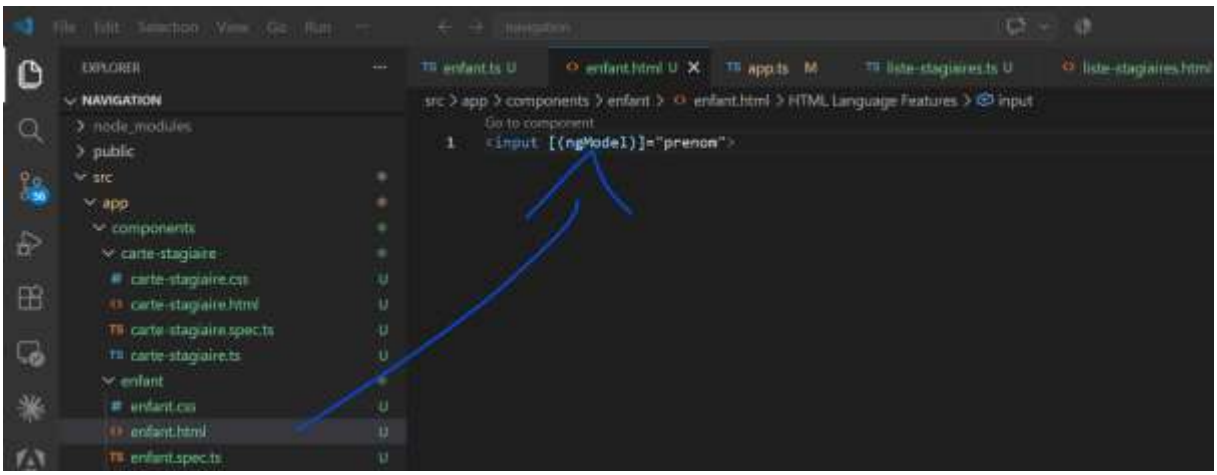
Ts :



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

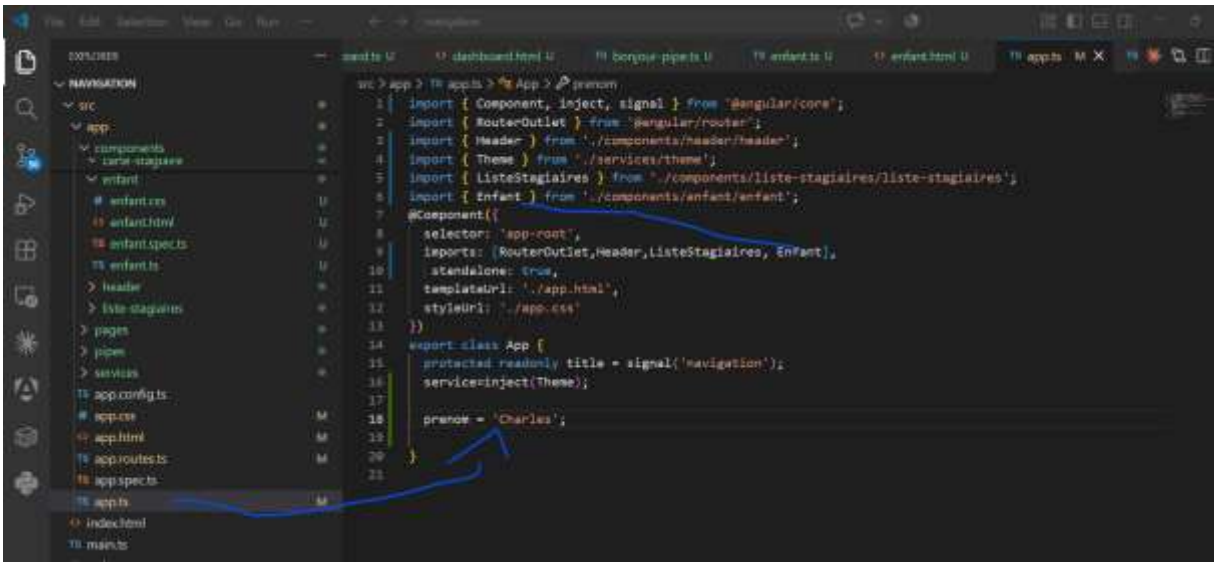
prenom = model('');

html :



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

Parent

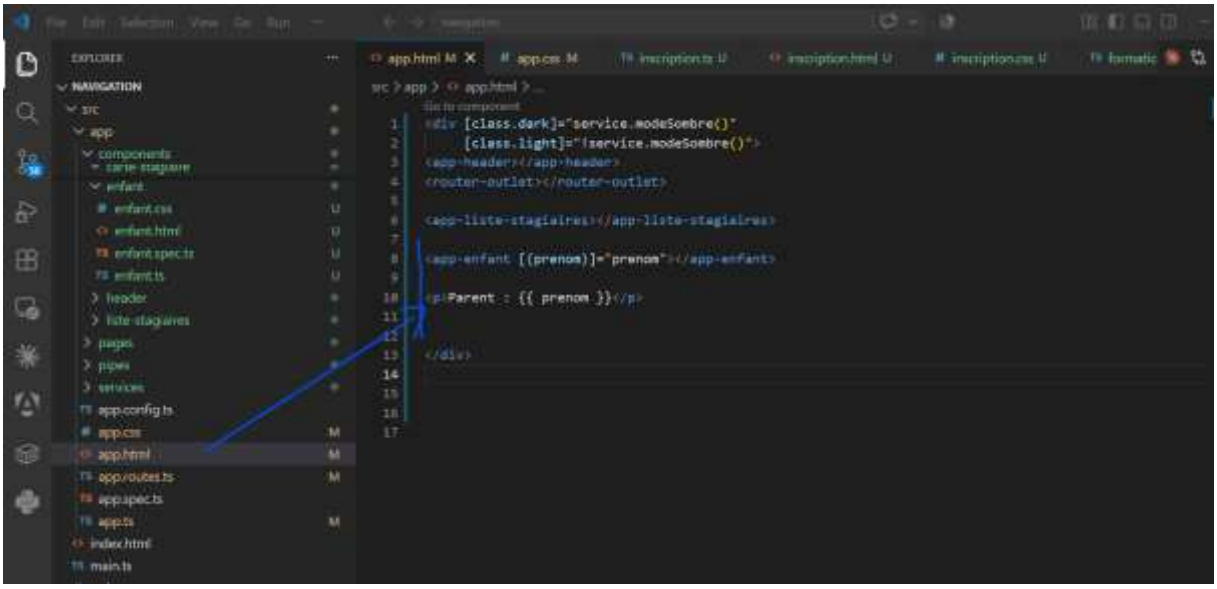


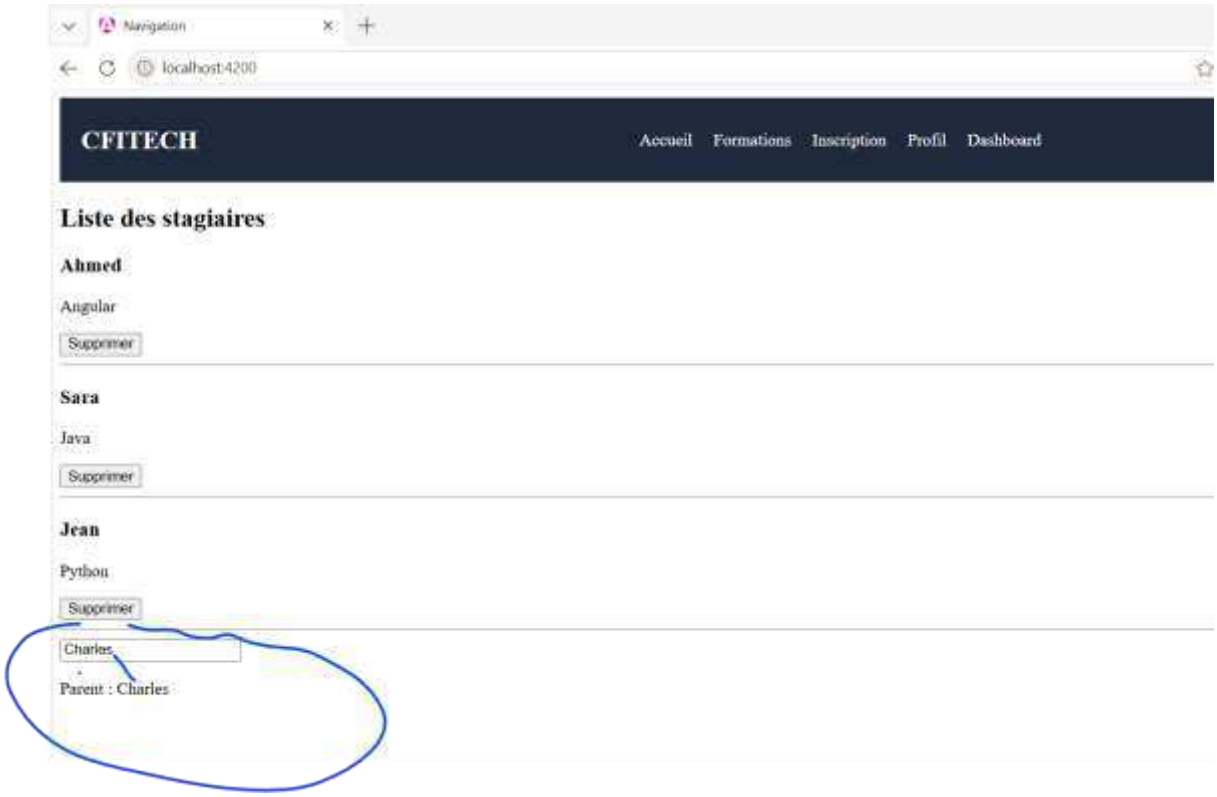
prenom = 'Charles';

html :

<app-enfant [(prenom)]="prenom"></app-enfant>

Parent : {{ prenom }}





Partie 4 : ViewChild()

ViewChild() permet à un composant parent d'obtenir une référence vers un élément présent dans son template.

Cela peut être :

- un composant enfant ;
- un élément HTML ;

Ex : Formulaire de connexion

Imaginons une page de connexion.

Le composant **LoginFormComponent** contient :

- un champ email ;
- un champ mot de passe ;
- une méthode pour vider le formulaire ;
- une méthode pour mettre le focus sur le champ email.

Le **ParentComponent** veut contrôler ce formulaire.

Étape 1 – Le composant enfant

login-form.component.ts

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

@Component({
 selector: 'app-login-form',
 standalone: true,
 template: `
  <input
   type="email"
   [(ngModel)]="email"
   placeholder="Email">

  <br><br>

  <input
   type="password"
   [(ngModel)]="password"
   placeholder="Mot de passe">
 `
})
export class LoginFormComponent {

 email = 'alice@email.com';
 password = '123456';

}
```

Le formulaire fonctionne.

Mais le parent ne peut pas accéder à ses données.

Étape 2 – Le parent

```
<h2>Connexion</h2>
```

<app-login-form></app-login-form>

<button>Afficher l'email</button>

Question :

Comment récupérer l'email depuis le parent ?

On utilise **ViewChild()**.

Étape 3 – Accéder au composant

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

import { LoginFormComponent } from './login-form.component';

@Component({

selector: 'app-parent',

standalone: true,

imports: [LoginFormComponent],

templateUrl: './parent.component.html'

})

export class ParentComponent {

loginForm = viewChild(LoginFormComponent);

}

En Angular moderne, on utilise **viewChild()** (fonction) plutôt que le décorateur @ViewChild.

Étape 4 – Accéder à une variable

Le composant enfant possède :

email = 'alice@email.com';

Le parent peut lire cette valeur.

showEmail() {

console.log(this.loginForm()?.email);

}

Le ?. (opérateur de chaînage optionnel) signifie :

« Si l'objet situé à gauche existe, alors continue et accède à la propriété. Sinon, ne provoque pas d'erreur et retourne undefined. »

HTML

```
<button (click)="showEmail()">
  Afficher l'email
</button>
```

Résultat

Console

alice@email.com

Le parent a accédé directement à une variable de l'enfant.

Étape 5 – Modifier une variable

Le parent peut également écrire :

```
fillEmail() {

  this.loginForm()!.email = 'admin@gmail.com';

}
```

L'opérateur **!** est utilisé lorsque le développeur est **certain qu'une variable ou un objet n'est ni null ni undefined**.

HTML

```
<button (click)="fillEmail()">
  Remplir l'email
</button>
```

Résultat

Avant

alice@email.com

Après

admin@gmail.com

Le champ du formulaire est mis à jour automatiquement.

Étape 6 – Appeler une méthode

Ajoutons une méthode dans l'enfant.

```
clearForm() {

 this.email = '';
 this.password = '';

}
```

Le parent peut l'appeler.

```
clear() {

 this.loginForm()?.clearForm();

}
```

HTML

```
<button (click)="clear()">
 Vider le formulaire
</button>
```

Résultat

Avant

Email : alice@email.com

Password : 123456

Après

Email :

Password :

Le parent exécute une méthode de l'enfant.

Étape 7 – Accéder à un input HTML

Dans le composant enfant

```
<input
 #emailInput
 type="email"
 [(ngModel)]="email">
```

```
import { ElementRef, viewChild } from '@angular/core';


emailInput =
 viewChild<ElementRef<HTMLInputElement>>('emailInput');
```

On ajoute une méthode.

```
focusEmail() {

 this.emailInput()?.nativeElement.focus();


}
```

Étape 8 – Le parent appelle cette méthode

```
focus() {

 this.loginForm()?.focusEmail();


}
```

```
<button (click)="focus()">
 Focus Email
</button>
```

Quand on clique sur le bouton,

le curseur est placé directement dans le champ email.

Résumé

Avec

loginForm = viewChild(LoginFormComponent);

le parent peut :

Lire une variable

this.loginForm()?.email;

Modifier une variable

this.loginForm()!.email = 'admin@gmail.com';

Appeler une méthode

this.loginForm()?.clearForm();

Déclencher une action

this.loginForm()?.focusEmail();

viewChild() permet au **parent** d'obtenir une **référence directe** sur son composant enfant.

Partie 5 : Content Projection (ng-content)

<ng-content> permet d'insérer dans un composant le contenu envoyé par son parent.

Exconcret : Une boutique en ligne

Nous développons une boutique e-commerce.

La page affiche plusieurs produits.

Chaque produit est présenté dans une **carte**.

Par exemple :

```
+----------------------------------+
|  iPhone 16                |
| Prix : 999 €              |
| [Ajouter au panier]          |
+----------------------------------+


+----------------------------------+
|  MacBook Air              |
| Prix : 1299 €             |
| [Ajouter au panier]          |
+----------------------------------+
```

+----------------------------------+

| AirPods Pro |

| Prix : 249 € |

| [Ajouter au panier] |

+----------------------------------+

On remarque que :

- toutes les cartes ont le même style ;
- seul le contenu change.

Le problème

Sans composant réutilisable, on écrit plusieurs fois le même HTML.

```
<div class="card">
  <h2>iPhone 16</h2>
  <p>999 €</p>
  <button>Ajouter au panier</button>
</div>

<div class="card">
  <h2>MacBook Air</h2>
  <p>1299 €</p>
  <button>Ajouter au panier</button>
</div>

<div class="card">
  <h2>AirPods Pro</h2>
  <p>249 €</p>
  <button>Ajouter au panier</button>
</div>
```

Beaucoup de code est dupliqué.

La solution

On crée un composant :

ProductCard

Mais une question se pose :

Comment faire pour que chaque carte affiche un contenu différent ?

C'est exactement le rôle de **<ng-content>**.

Étape 1 : Création de la carte

card.component.html

```
<div class="card">
  <ng-content></ng-content>
</div>
```

La carte connaît uniquement son apparence.

Elle ne connaît pas le produit.

Étape 2 : Utilisation

Pour le premier produit :

```
<app-card>

  <h2>iPhone 16</h2>

  <p>999 €</p>

  <button>Ajouter au panier</button>

</app-card>
```

Affichage :

```
+------------------------+

iPhone 16
```

999 €

[Ajouter au panier]

+------------------------+

Deuxième produit

On réutilise exactement le même composant.

<app-card>

  <h2>MacBook Air</h2>

  1299 €

  <button>Ajouter au panier</button>

</app-card>

Aucune modification du composant.

Troisième produit

<app-card>

  <h2>AirPods Pro</h2>

  249 €

  <button>Ajouter au panier</button>

</app-card>

Encore une fois, la même carte est utilisée.

Que fait <ng-content> ?

Lorsque Angular rencontre :

<app-card>

  <h2>iPhone 16</h2>

  999 €

  <button>Ajouter au panier</button>

</app-card>

Il remplace automatiquement :

<ng-content></ng-content>

par :

<h2>iPhone 16</h2>

999 €

<button>Ajouter au panier</button>

Le résultat devient :

  <h2>iPhone 16</h2>

  999 €

  <button>Ajouter au panier</button>

Réutilisation

Le même composant peut afficher :

Un produit

```
<app-card>

  <h2>iPhone 16</h2>

  <p>999 €</p>

</app-card>
```

Un utilisateur

```
<app-card>

  <h2>Alice Martin</h2>

  <p>Développeuse Angular</p>

</app-card>
```

Une formation

```
<app-card>

  <h2>Angular 21</h2>

  <p>Formation avancée</p>

  <button>S'inscrire</button>

</app-card>
```

Le composant **Card** ne change jamais.

Seul le contenu projeté change.

<ng-content> permet de créer des composants **génériques et réutilisables**.

Pourquoi créer plusieurs composants alors qu'on pourrait tout faire dans un seul ?

Lorsqu'on découvre Angular, une question revient souvent : **pourquoi créer un composant parent et un composant enfant alors que toutes les données sont souvent créées dans le composant parent ?** Dans les exemples de cours, il est vrai que les données sont généralement écrites directement dans le composant parent pour simplifier l'apprentissage. Cela peut donner l'impression que la communication entre composants est inutile et que tout pourrait être réalisé dans un seul composant.

En réalité, ces exemples sont volontairement simplifiés. Dans une application professionnelle, les données ne sont presque jamais créées directement dans un composant. Elles proviennent généralement d'une **base de données**, d'une **API REST** ou d'un **service Angular**. Le composant parent récupère ces données puis les distribue aux différents composants enfants qui sont chargés de les afficher.

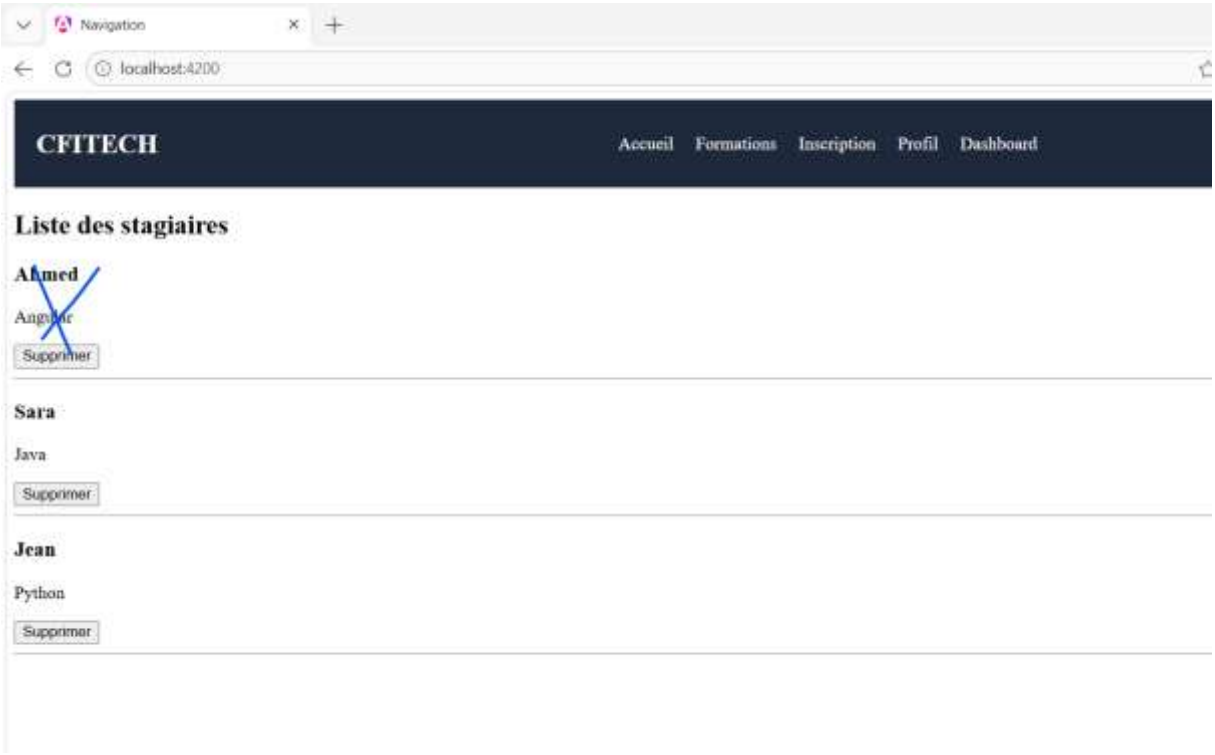
Cependant, la véritable raison de créer plusieurs composants n'est pas uniquement le partage des données. L'objectif principal est de **séparer les responsabilités**. Chaque composant doit avoir une mission bien précise. Par exemple, un composant peut être chargé de récupérer la liste des étudiants, un autre d'afficher la fiche d'un étudiant, un autre encore de gérer la barre de recherche ou la pagination. Cette organisation rend l'application plus claire, plus facile à comprendre, plus simple à maintenir et permet de réutiliser les mêmes composants à plusieurs endroits.

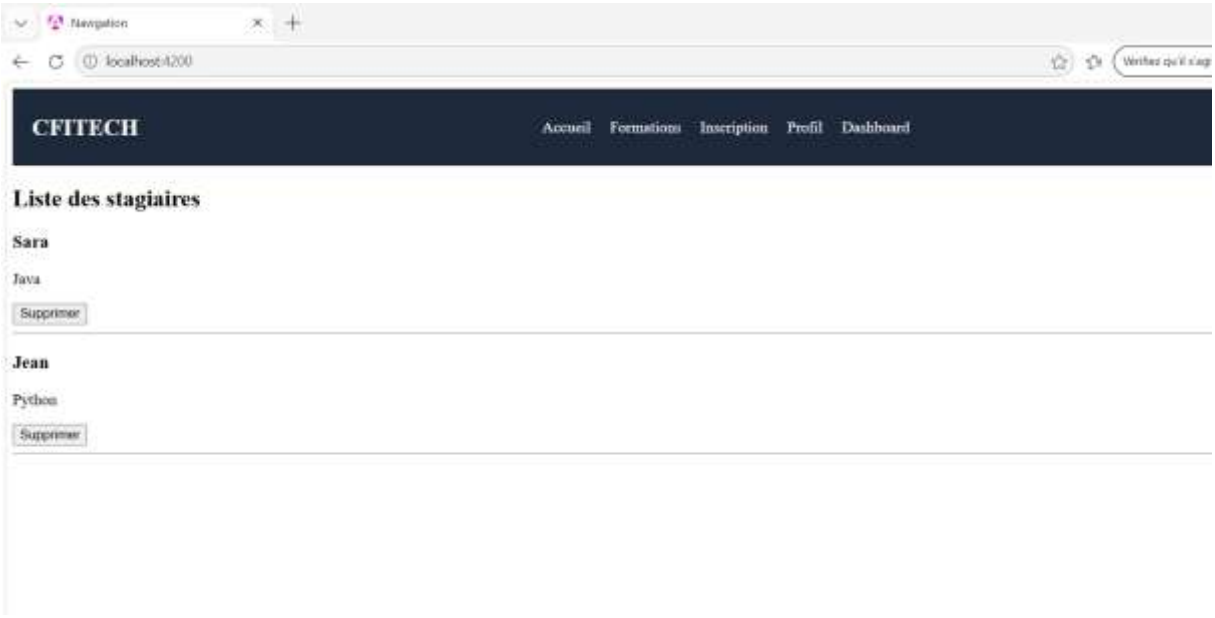
Prenons l'exemple d'une application de gestion des étudiants. Le composant **ListeStagiairesComponent** récupère la liste complète des stagiaires depuis un service. En revanche, il ne s'occupe pas de l'affichage détaillé de chaque stagiaire. Cette tâche est confiée au composant **CarteStagiaireComponent**, qui est uniquement responsable de l'affichage d'une carte. Si demain on souhaite afficher les stagiaires sur une autre page de l'application, il suffira de réutiliser **CarteStagiaireComponent** sans avoir à recopier son code.

Une fois que l'application est découpée en plusieurs composants, ceux-ci deviennent indépendants. Ils doivent alors échanger des informations pour fonctionner ensemble. C'est précisément à ce moment qu'interviennent **input()** et **output()**. Le composant parent transmet des données à l'enfant grâce à **input()**, tandis que le composant enfant informe son parent qu'une action a été réalisée grâce à **output()**.

On peut comparer cela au fonctionnement d'une entreprise. Le directeur pourrait effectuer lui-même toutes les tâches, mais ce serait inefficace. Il répartit donc le travail entre plusieurs services : les ressources humaines, la comptabilité, le service commercial, etc. Chaque service possède une responsabilité bien définie. Comme ils travaillent ensemble, ils doivent communiquer en permanence. Angular fonctionne exactement selon le même principe : les composants sont spécialisés, puis ils communiquent lorsqu'ils en ont besoin.

En conclusion, **les composants ne sont pas créés pour communiquer entre eux**. Ils sont créés pour **découper l'application en plusieurs parties ayant chacune une responsabilité précise**. La communication avec **input()** et **output()** n'est qu'une conséquence de cette séparation. C'est cette architecture qui permet de développer des applications Angular professionnelles, modulaires, réutilisables et faciles à faire évoluer.


Navigation
localhost:4200
CFITECH
Accueil
Formations
Inscription
Profil
Dashboard
Liste des stagiaires
Ahmed
Angular
Supprimer
Sara
Java
Supprimer
Jean
Python
Supprimer


Navigation
localhost:4200
CFITECH
Accueil
Formations
Inscription
Profil
Dashboard
Liste des stagiaires
Sara
Java
Supprimer
Jean
Python
Supprimer

Module 4 — HTTP Client

1. Qu'est-ce que HTTP ?

Dans une application Angular, les données ne sont généralement pas écrites directement dans le composant.

Elles sont stockées dans un **backend**.

Par exemple :

Angular

│

│ HTTP

v

API REST

│

v

Base de données

Exemple professionnel :

Angular

│

│ GET /formations

v

Backend

│

v

MySQL

Angular demande les formations au serveur.

Le serveur récupère les données dans la base de données et les renvoie à Angular.

2. Qu'est-ce que HttpClient ?

HttpClient est l'outil Angular qui permet de communiquer avec une API HTTP.

Il permet notamment de faire :

GET → récupérer

POST → ajouter

PUT → modifier complètement

PATCH → modifier partiellement

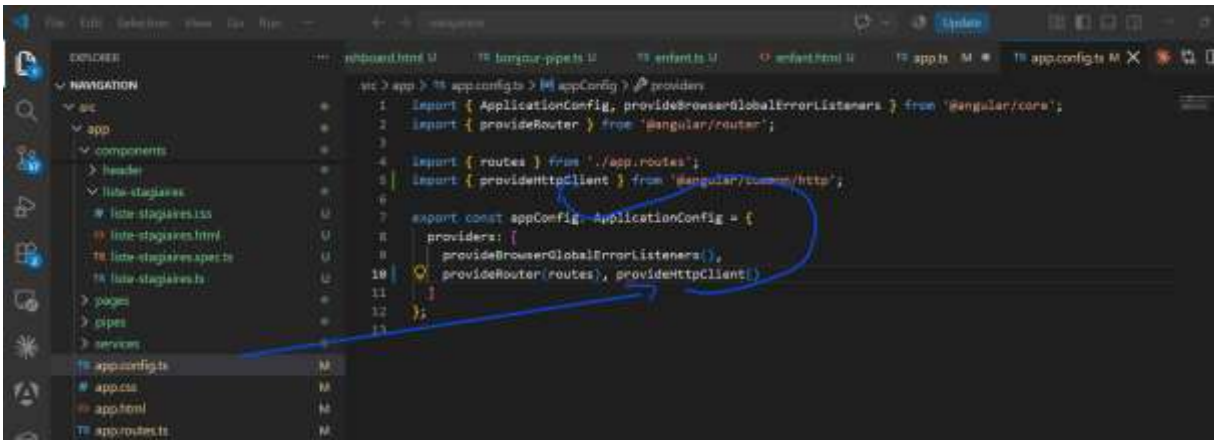
DELETE → supprimer

3. provideHttpClient()

Avec Angular moderne, on configure HTTP dans app.config.ts.

app.config.ts

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

export const appConfig: ApplicationConfig = {

 providers: [

  provideHttpClient()

 ]

};
```



La ligne :

provideHttpClient()

dit à Angular :

« Mon application a besoin du client HTTP. »

Sans cette configuration, l'injection de HttpClient ne fonctionnera pas correctement.

4. Notre API

Pour réaliser le TP, nous allons utiliser une API REST de démonstration.

Par exemple :

https://jsonplaceholder.typicode.com

Nous allons travailler avec :

/posts

Chaque élément possède par exemple :

```
{
  "id": 1,
  "title": "Angular",
  "body": "Formation Angular"
}
```

Nous allons considérer que :

- title = nom de la formation
- body = description

Pour un vrai projet professionnel, on utiliserait évidemment une API développée pour l'application, avec une vraie base de données.

5. Création du service

Nous allons créer :

formation.service.ts

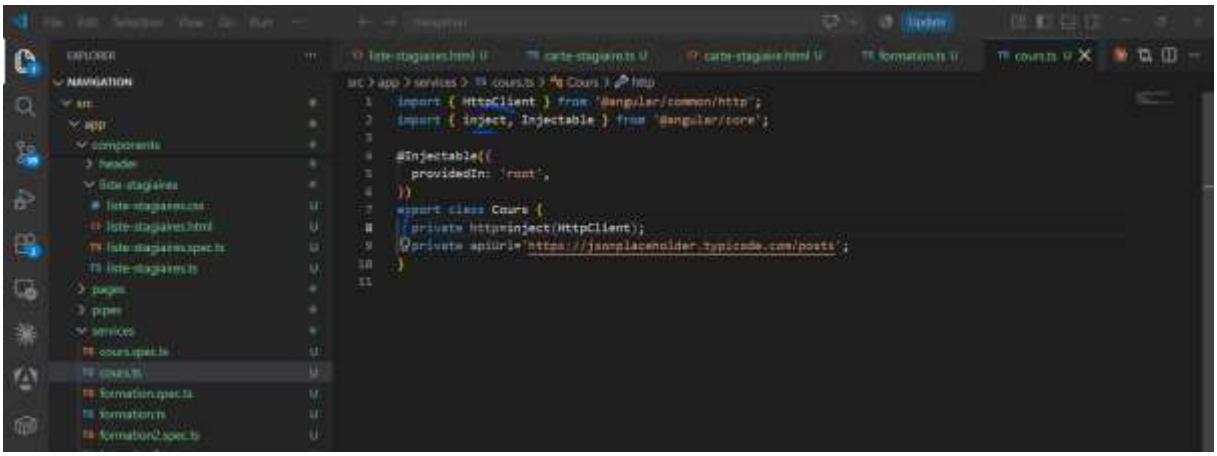
Le service sera responsable de toutes les communications avec l'API.

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);

  private apiUrl =
    'https://jsonplaceholder.typicode.com/posts';

}
```

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



6. Comprendre inject(HttpClient)

Cette ligne :

private http = inject(HttpClient);

permet d'obtenir une instance de HttpClient.

Nous pouvons ensuite utiliser :

this.http.get()

this.http.post()

this.http.put()

this.http.patch()

this.http.delete()

7. GET — récupérer les données

Le premier besoin est d'afficher les formations.

Nous allons créer :

getFormations()

Service

```
getFormations() {

 return this.http.get<any[]>(
  this.apiUrl
 );

}
```

Cette méthode demande les données à l'API.

```
this.http.get<any[]>(
 this.apiUrl
);
```

signifie :

« Envoie une requête GET à cette URL et attends un tableau de données. »

Le serveur peut répondre :

```
[
 {
  "id": 1,
  "title": "Angular",
  "body": "Formation Angular"
 },
 {
  "id": 2,
  "title": "React",
  "body": "Formation React"
```

```
 }
]
```

8. Utiliser GET dans le composant

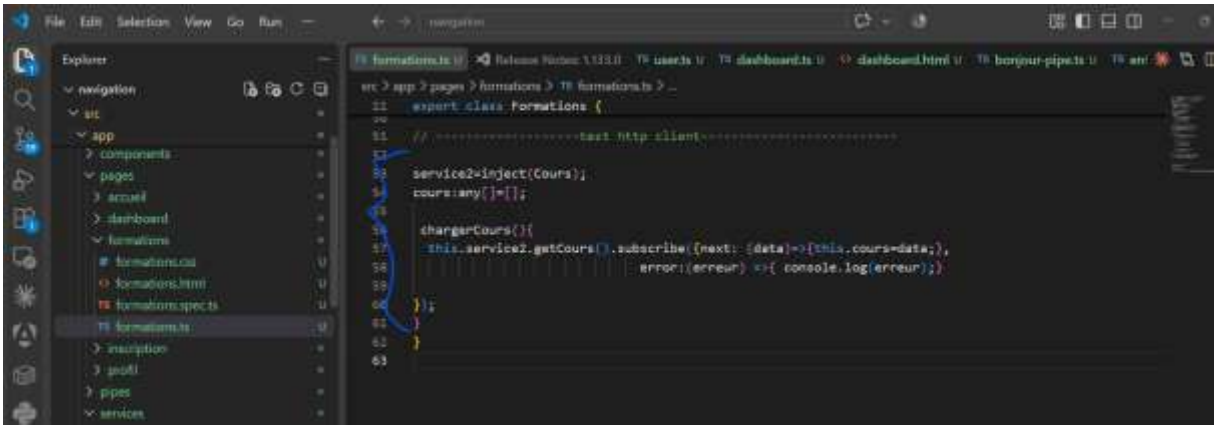
formation.ts

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

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

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

 service = inject(FormationService);

 formations: any[] = [];

 chargerFormations() {

  this.service.getFormations()
   .subscribe({

    Next: (data) => {

     this.formations = data;
```

```
      },

      error: (erreur) => {

        console.log(erreur);

      }

    });

 }

}
```

```
service2=inject(Cours);
cours:any[]=[];

 chargerCours(){
  this.service2.getCours().subscribe({next: (data)=>{this.cours=data;},
                        error:(erreur) =>{ console.log(erreur);}

});
}
```

```
import { Cours } from '../../services/cours';
```



9. subscribe() ?

La méthode subscribe() joue un rôle essentiel avec les requêtes HTTP Angular. Lorsque le composant appelle this.service2.getCours(), la requête est envoyée vers l'API, mais la réponse n'arrive pas immédiatement : elle peut prendre quelques millisecondes ou plus. subscribe() permet donc de s'abonner à la réponse et de dire à Angular : « lorsque l'API répond, exécute ce code ». Si la requête réussit, Angular exécute next et place les données reçues dans data, puis this.cours = data permet de les stocker dans le composant. Si la requête échoue, Angular exécute error afin de gérer le problème. On peut donc retenir que getCours() prépare et retourne la requête HTTP, tandis que subscribe() permet d'attendre et de récupérer réellement le résultat de cette requête pour pouvoir le traiter.

10. next

next: (data) => {

this.formations = data;

}

next est exécuté lorsque la requête réussit.

Par exemple, le serveur renvoie :

[

{

"id": 1,

"title": "Angular"

}

]

Alors :

data

contient cette réponse.

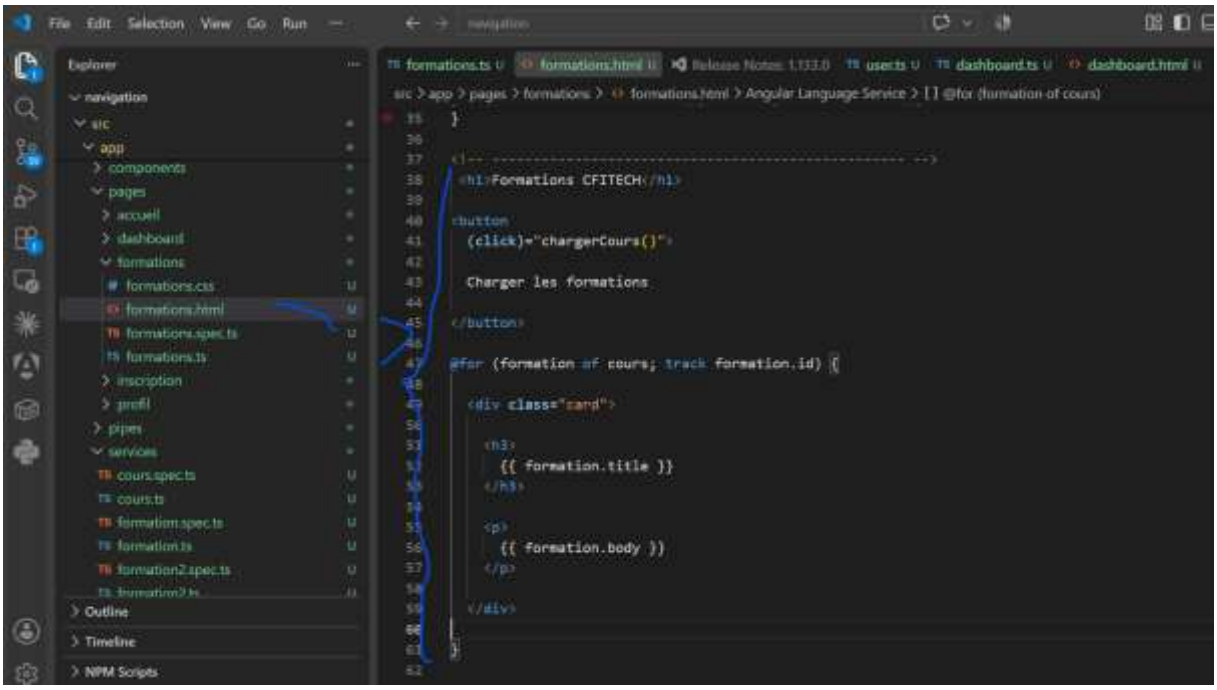
On fait donc :

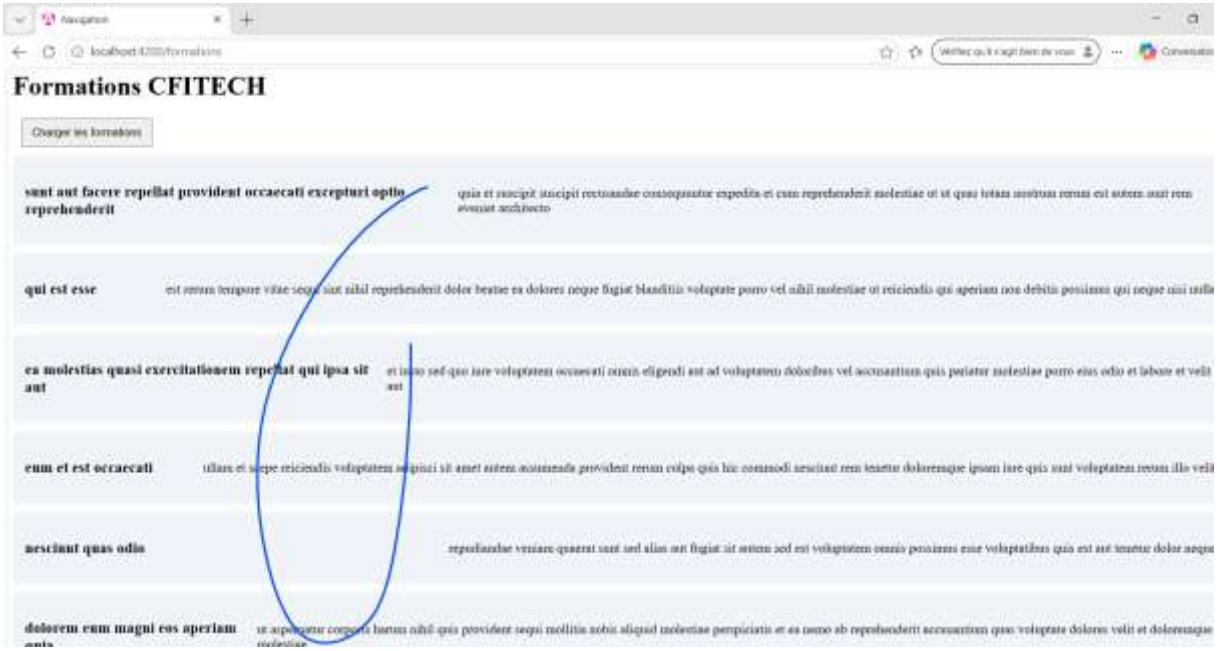
this.formations = data;

11. Affichage HTML

```
<h1>Formations CFITECH</h1>

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

 Charger les formations

</button>

@for (formation of formations; track formation.id) {

 <div class="card">

  <h3>
   {{ formation.title }}
  </h3>

  <p>
   {{ formation.body }}
  </p>

 </div>
```

}



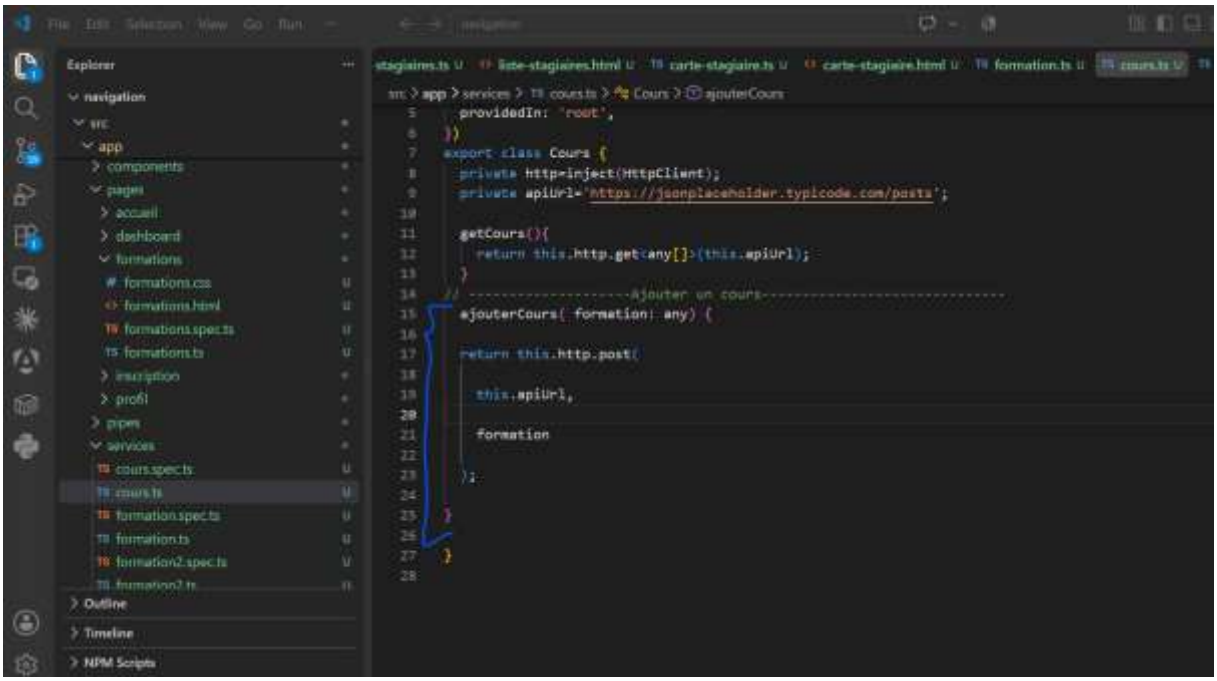


12. POST — ajouter une formation

POST permet généralement de **créer une nouvelle ressource**.

1. Le service prépare l'ajout:

```
ajouterFormation(formation: any) {

 return this.http.post(

   this.apiUrl,

   formation

 );

}
```



Nous envoyons :

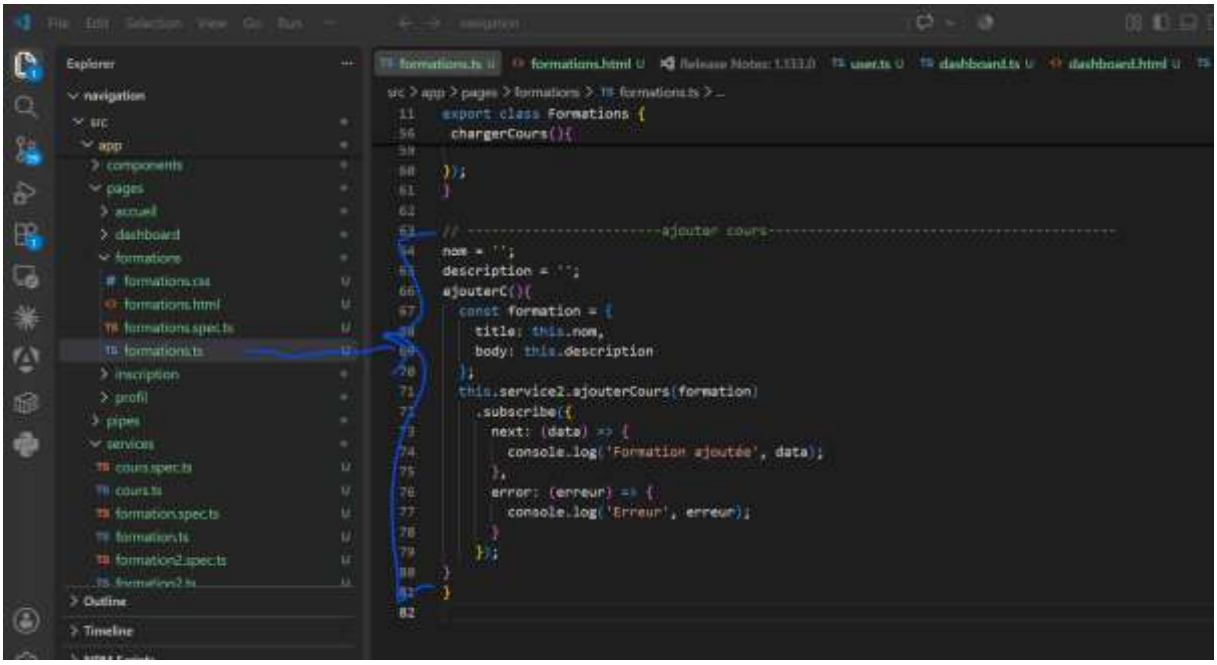
```
{
 title: 'Angular',
 body: 'Formation Angular'
}
```

vers l'API.

2. Le composant appelle cette méthode

Par exemple :

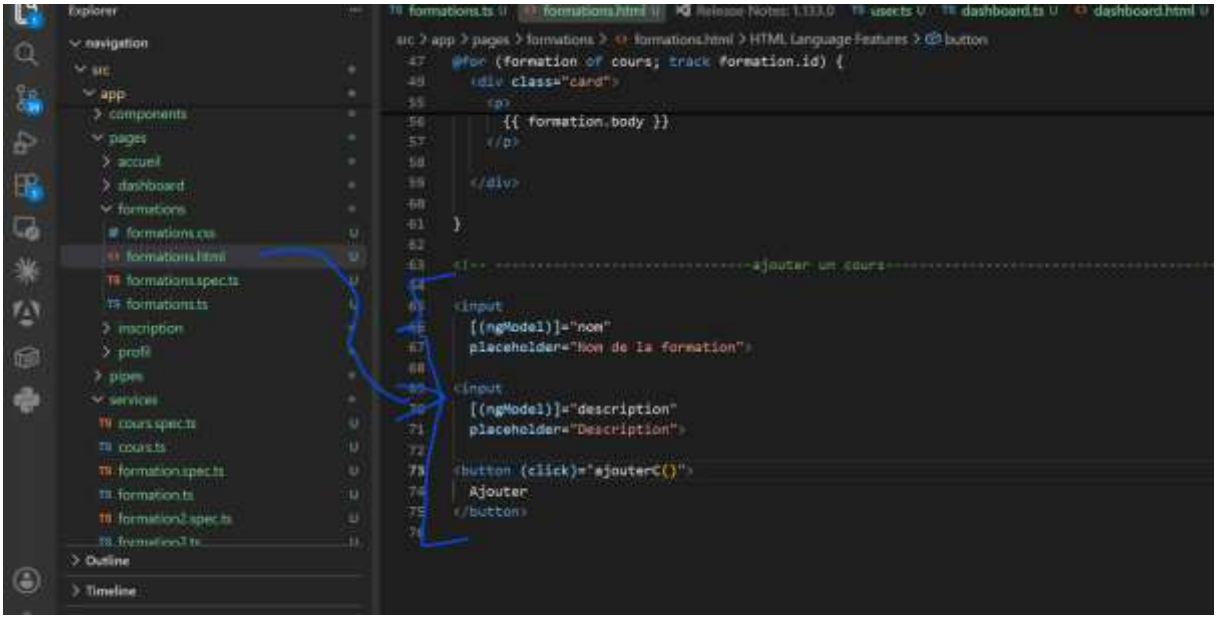
```
ajouter() {

 const formation = {
  title: this.nom,
  body: this.description
 };

 this.service.ajouterFormation(formation)
  .subscribe({

    next: (data) => {

      console.log('Formation ajoutée', data);

    },

    error: (erreur) => {

      console.log('Erreur', erreur);

    }

  });

}
```



C'est **le composant qui déclenche réellement l'opération** lorsque l'utilisateur clique sur le bouton.

3. Le HTML permet à l'utilisateur de saisir et de lancer l'ajout

```
<input
 [(ngModel)]="nom"
 placeholder="Nom de la formation">

<input
 [(ngModel)]="description"
 placeholder="Description">

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



13. PUT — modifier complètement

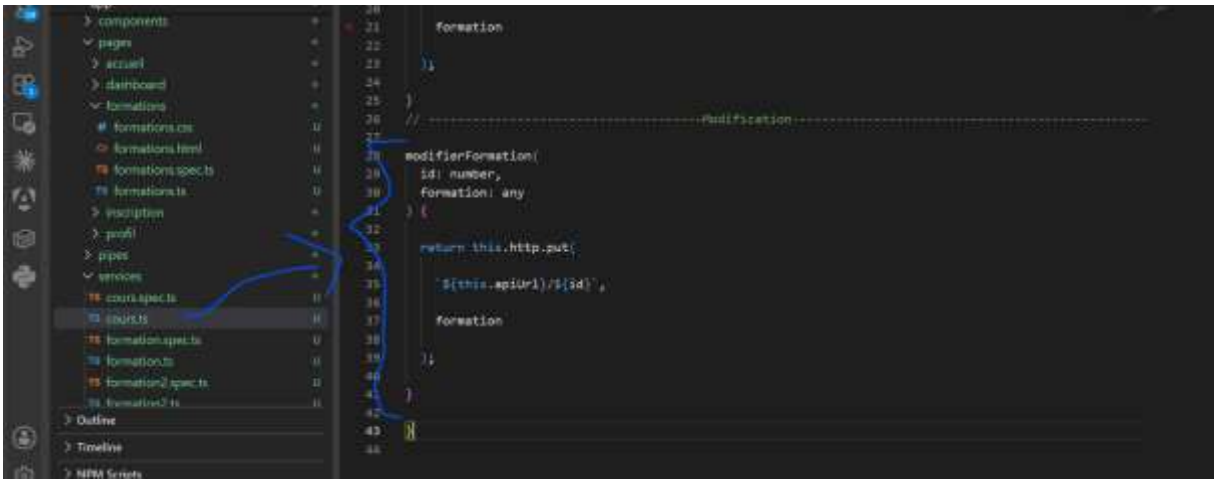
PUT est utilisé lorsqu'on souhaite remplacer complètement une ressource.

Service :

```
modifierFormation(
 id: number,
 formation: any
){

 return this.http.put(

  `${this.apiUrl}/${id}`,

  formation
```

);

}



Exemple :

PUT /posts/5

Composant ts :

// Formation que nous voulons modifier

idModification = 5;

// Nouvelles informations

title = '';

description2 = '';

modifier() {

const formation = {

title: this.title,

body: this.description2

};

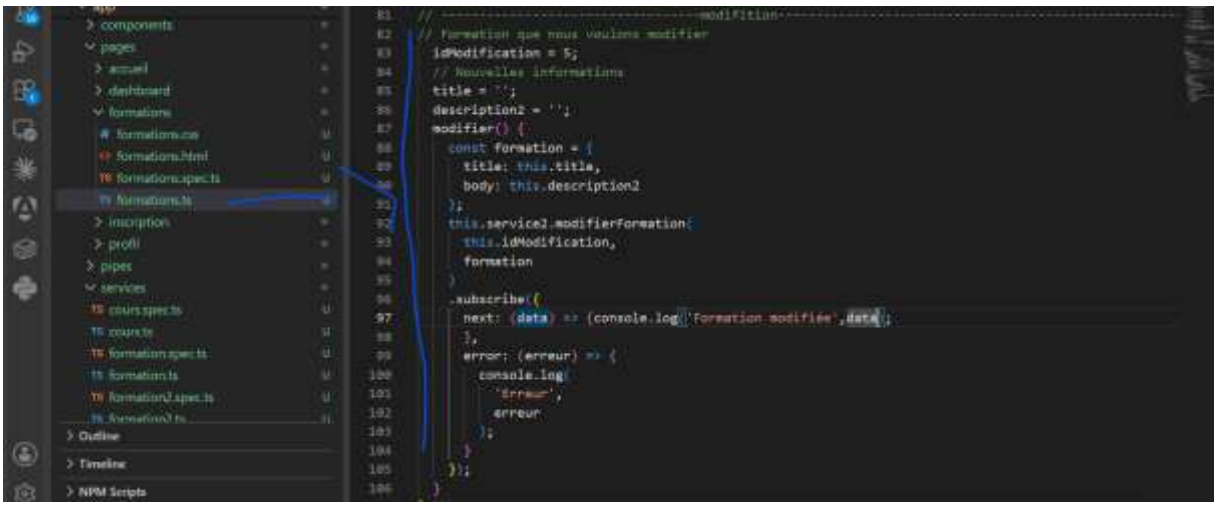
this.service2.modifierFormation(

this.idModification,

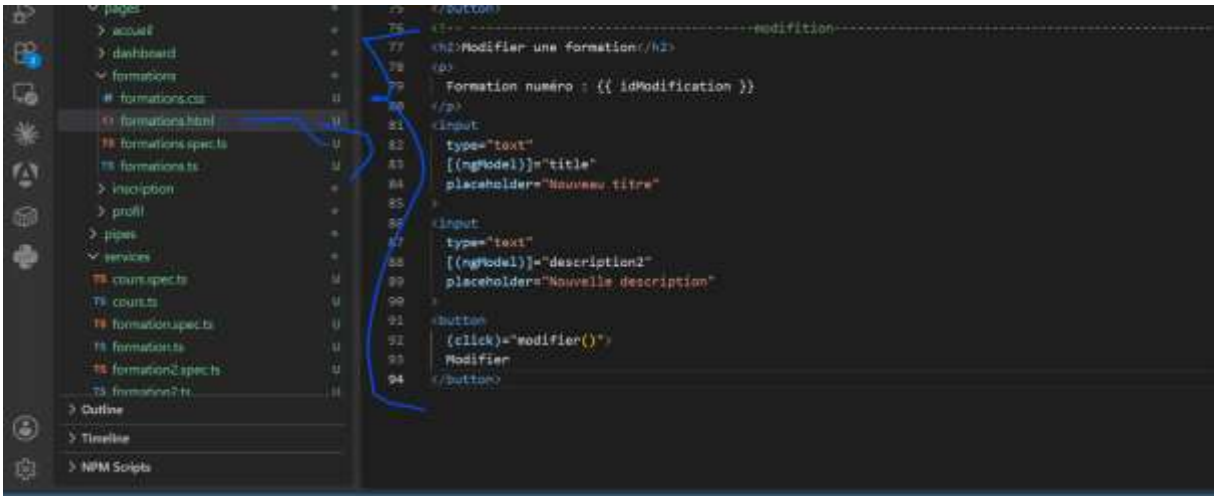
formation

)

```
  .subscribe({
    next: (data) => {console.log('Formation modifiée',data);
    },
    error: (erreur) => {
      console.log(
        'Erreur',
        erreur
      );
    }
  });
}
```



Html :



La différence principale entre **PUT** et **PATCH**:

- **PUT** → on remplace généralement **toute la ressource**.
- **PATCH** → on modifie **seulement une partie** de la ressource.

Ex concret

Supposons que l'API possède cette formation :

```
{
 "id": 5,
 "title": "Angular",
 "description": "Formation Angular débutant",
 "duree": "3 jours"
}
```

PUT : remplacer complètement

Si on veut modifier la formation avec PUT, on envoie généralement **toutes les informations** :

```
this.http.put(
  `${this.apiUrl}/5`,
 {
   title: "Angular avancé",
   description: "Formation Angular niveau avancé",
   duree: "5 jours"
 }
);
```

PATCH : modifier seulement une partie

Si on veut uniquement changer le titre :

```
this.http.patch(
  `${this.apiUrl}/5`,
```

```
 {
   title: "Angular avancé"
 }
);
```

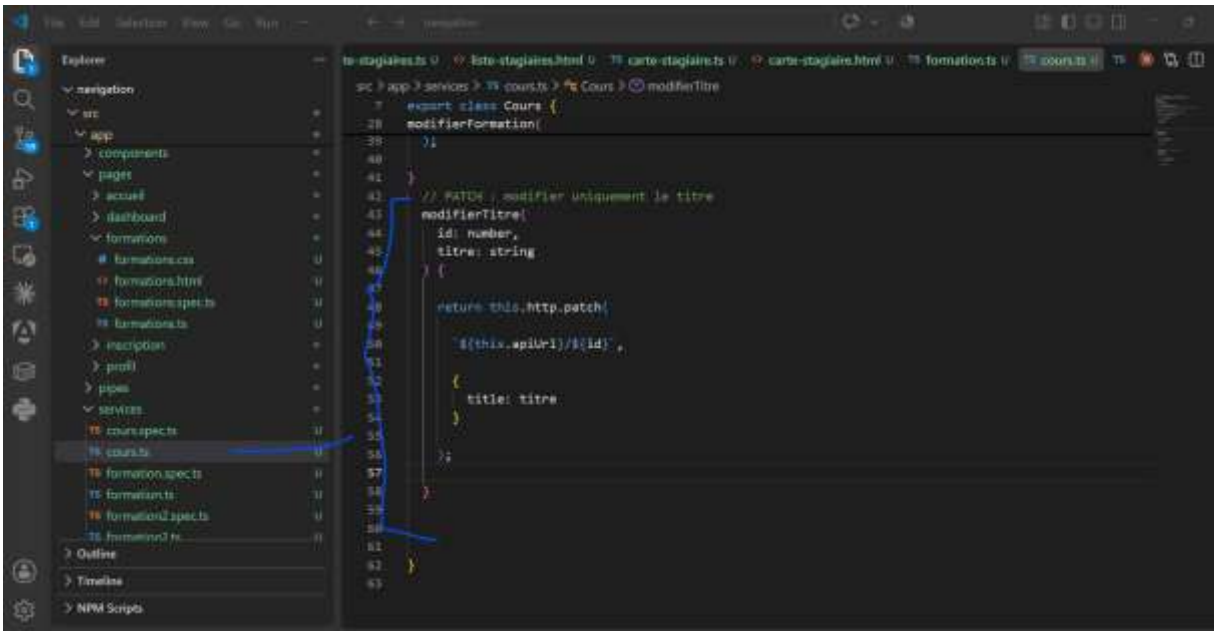
14. PATCH — modifier partiellement

1. Service — cours.ts

Le service contient la communication avec l'API.

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

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

 private http = inject(HttpClient);

 private apiUrl =
  'https://jsonplaceholder.typicode.com/posts';



 // PATCH : modifier uniquement le titre
 modifierTitre(
  id: number,
  titre: string
 ) {

  return this.http.patch(

   `${this.apiUrl}/${id}`,
```

```
    {
      title: titre
    }

  );

 }

}
```



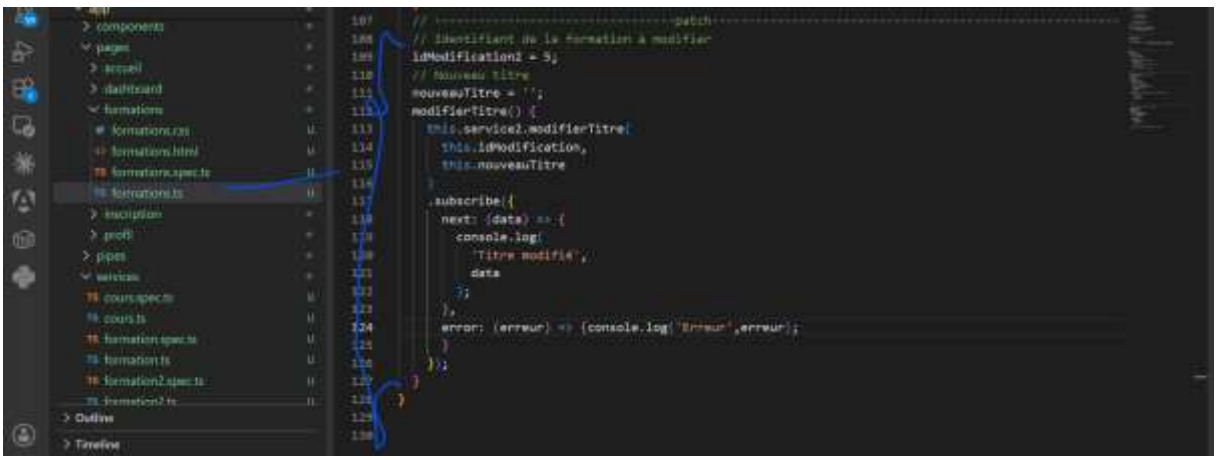
2. Composant — formations.ts

Maintenant, le composant va permettre à l'utilisateur de saisir le nouveau titre.

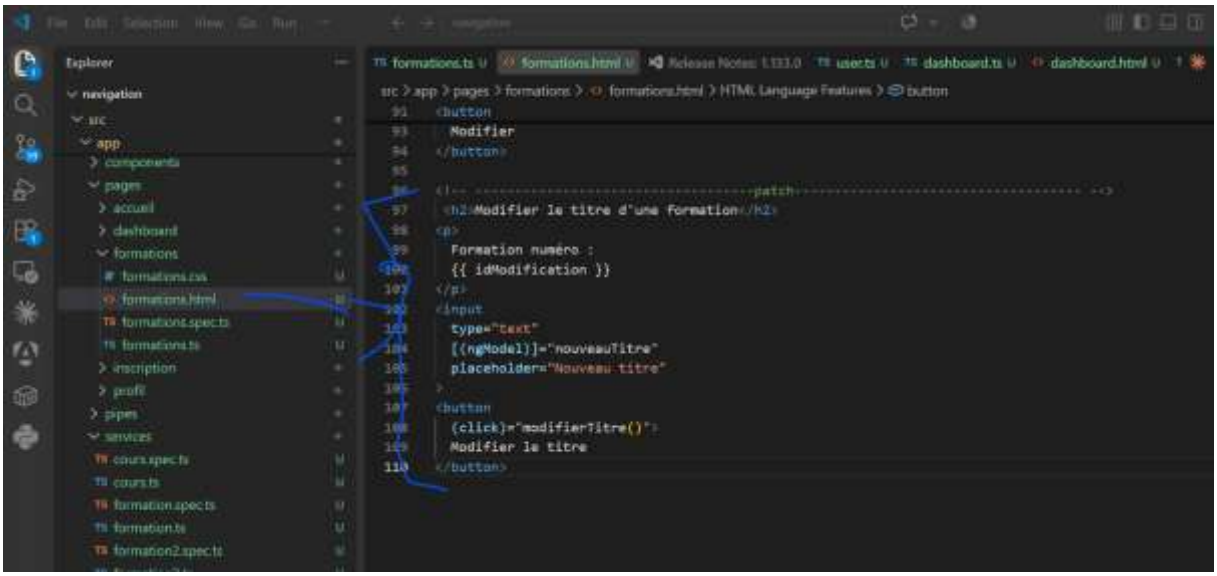
```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { Cours } from '../../services/cours';

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

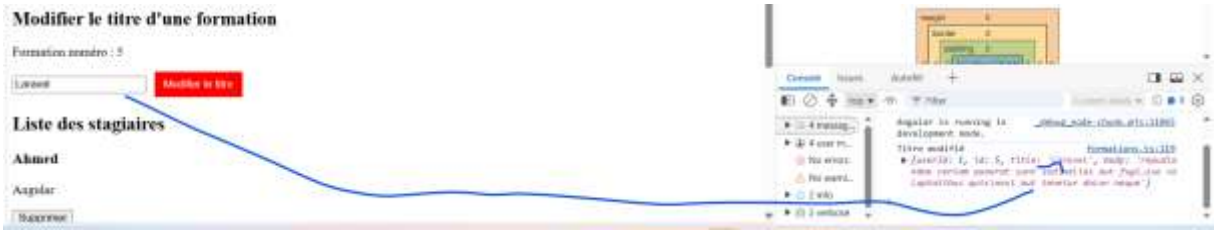
```
 imports: [FormsModule],
 templateUrl: './formations.html',
 styleUrl: './formations.css'
})
export class FormationsComponent {

 service = inject(Cours);

 // Identifiant de la formation à modifier
 idModification = 5;

 // Nouveau titre
 nouveauTitre = '';



 modifierTitre() {

  this.service.modifierTitre(

   this.idModification,

   this.nouveauTitre

  )
  .subscribe({

   next: (data) => {

     console.log(
      'Titre modifié',
      data
```

```
      );

    },

    error: (erreur) => {

      console.log(
        'Erreur',
        erreur
      );

    }

  });

 }

}
```



Html :





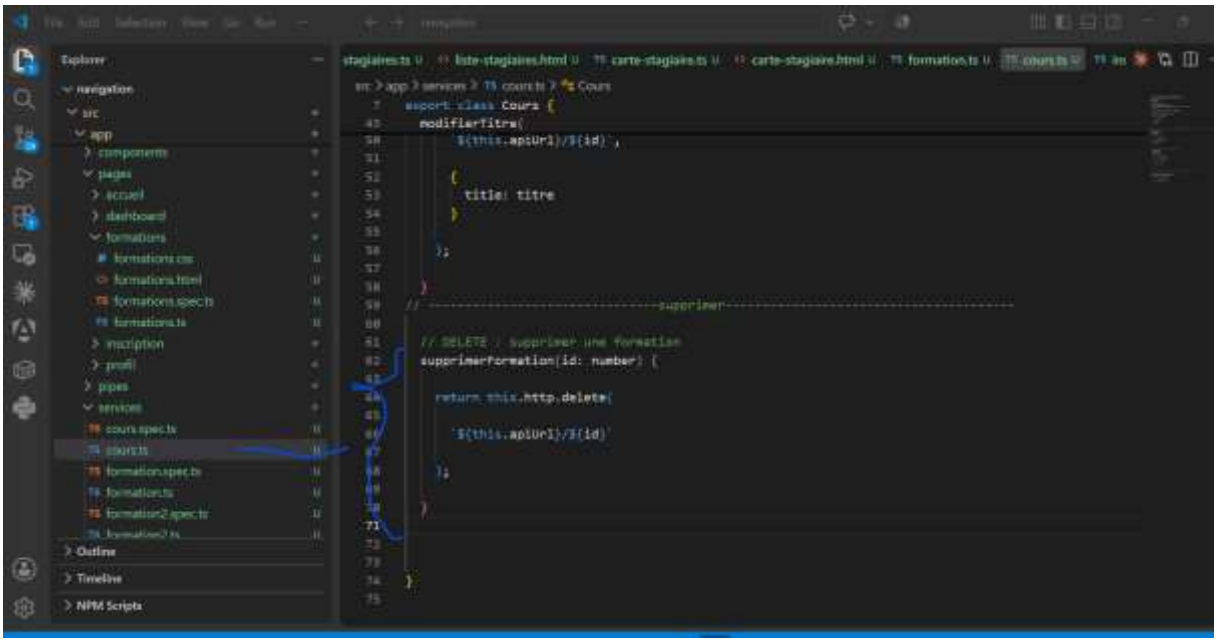
15. DELETE — supprimer

1. Service — cours.ts

Le service est responsable de communiquer avec l'API.

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

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

 private http = inject(HttpClient);

 private apiUrl =
  'https://jsonplaceholder.typicode.com/posts';
```

```
// DELETE : supprimer une formation
supprimerFormation(id: number) {

  return this.http.delete(

    `${this.apiUrl}/${id}`

  );

}

}
```



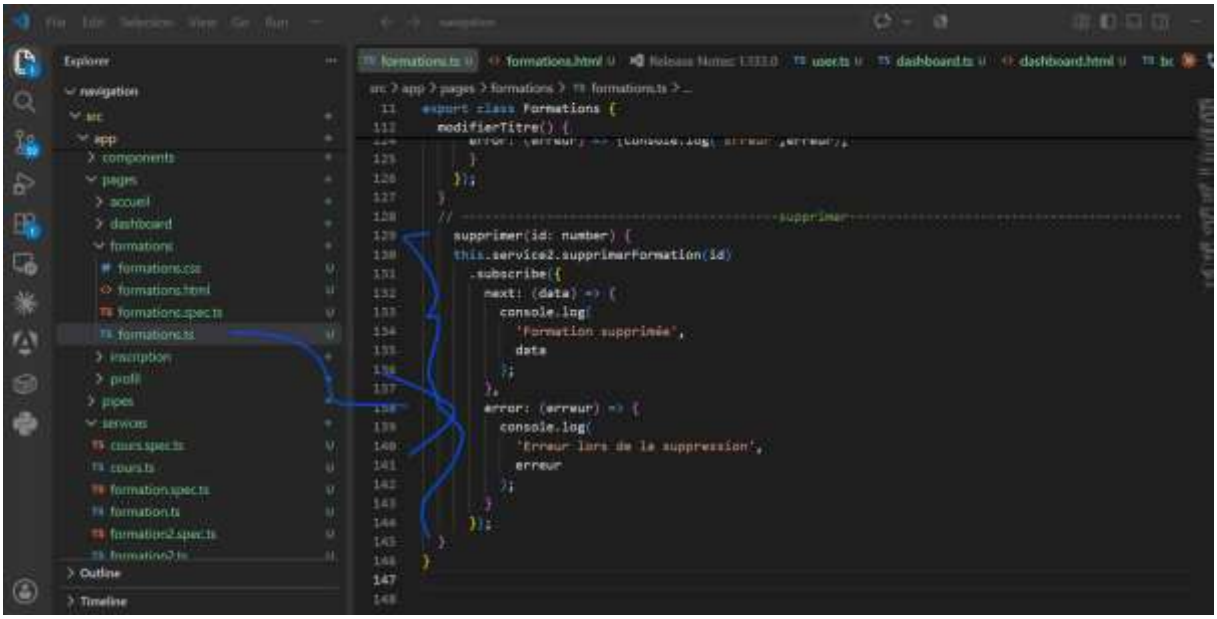
2. Composant — formations.ts

Maintenant, le composant va appeler le service lorsqu'on clique sur le bouton.

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

```
import { Cours } from '../../services/cours';

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

 service = inject(Cours);



 supprimer(id: number) {

  this.service.supprimerFormation(id)
   .subscribe({

    next: (data) => {

     console.log(
      'Formation supprimée',
      data
     );

    },

    error: (erreur) => {

     console.log(
```

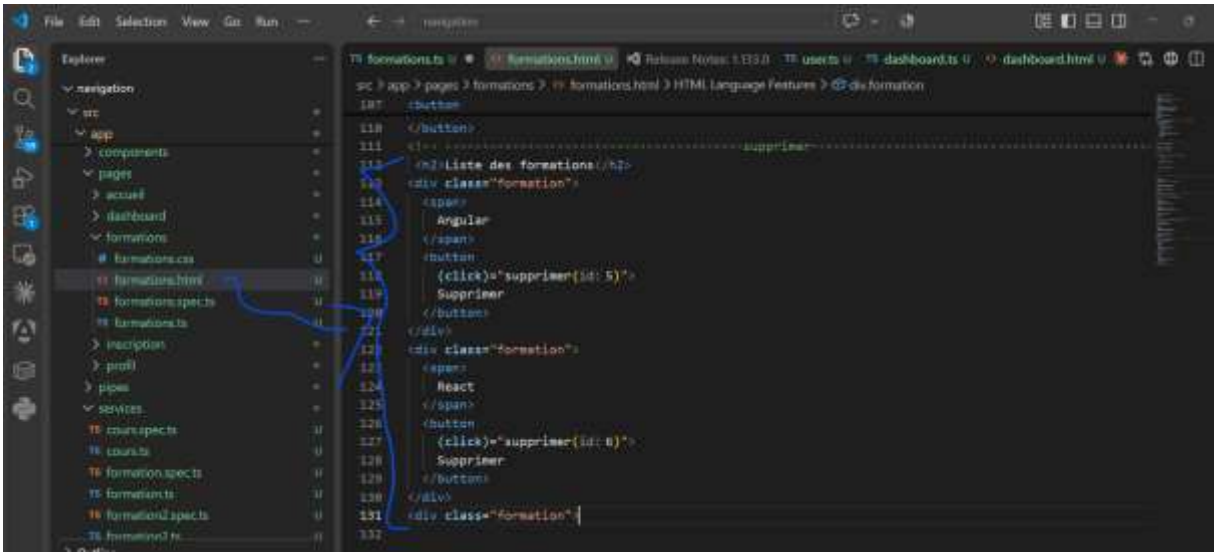
```
        'Erreur lors de la suppression',
        erreur
      );

    }

  });

 }

}
```

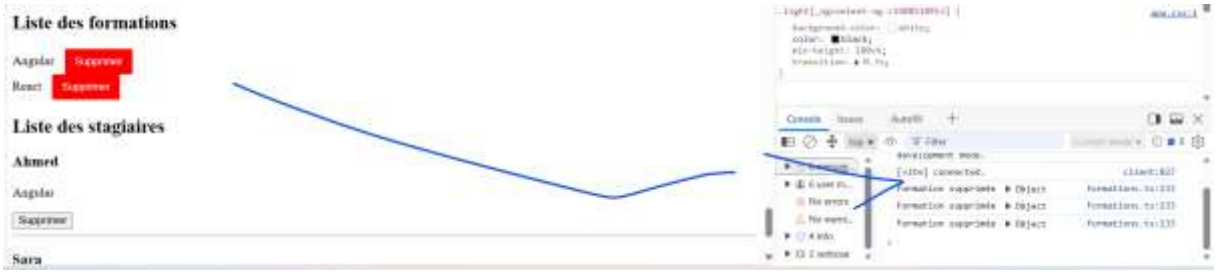


3. HTML — formations.html

```
<h2>Liste des formations</h2>

<div class="formation">

 <span>
  Angular
 </span>
```

```
 <button
  (click)="supprimer(5)">
  Supprimer
 </button>
</div>

<div class="formation">
 <span>
  React
 </span>
 <button
  (click)="supprimer(6)">
  Supprimer
 </button>
</div>

<div class="formation">
```





TP — Application Angular : Gestion d'une bibliothèque

Une bibliothèque souhaite une application Angular permettant de gérer ses livres.

Le responsable doit pouvoir :

- consulter les livres ;
- ajouter un livre ;
- modifier complètement un livre ;
- modifier uniquement son titre ;
- supprimer un livre ;
- gérer les erreurs de communication avec l'API.

Le projet doit utiliser **HttpClient et une API REST**.

1. Les données d'un livre

Chaque livre possède :

{

id: 1,

```
 titre: "Le Petit Prince",
 auteur: "Antoine de Saint-Exupéry",
 categorie: "Roman",
 annee: 1943
}
```

Créer une interface :

livre.ts

```
export interface Livre {

 id: number;

 titre: string;

 auteur: string;

 categorie: string;

 annee: number;

}
```

2. Structure du projet

Demandez aux stagiaires de créer :

```
src/
└── app/
    │
    ├── models/
    │   └── livre.ts
    │
    ├── services/
    │   └── livre.service.ts
```

```
  │
  └── pages/
      └── livres/
          ├── livres.ts
          ├── livres.html
          └── livres.css
```

3. Configuration HttpClient

Dans app.config.ts, configurer :

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

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

export const appConfig: ApplicationConfig = {

 providers: [

   provideHttpClient()

 ]
};
```

4. Service — livre.service.ts

Le service est responsable de **toutes les communications avec l'API**.

Créer :

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

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

import { Livre } from '../models/livre';
```

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

 private http = inject(HttpClient);

 private apiUrl =
  'https://jsonplaceholder.typicode.com/posts';



 // GET
 getLivres() {

  return this.http.get<Livre[]>(
   this.apiUrl
  );

 }



 // POST
 ajouterLivre(livre: Livre) {

  return this.http.post<Livre>(
   this.apiUrl,
   livre
  );

 }
```

```
// PUT
modifierLivre(
  id: number,
  livre: Livre
) {

  return this.http.put<Livre>(
    `${this.apiUrl}/${id}`,
    livre
  );

}


// PATCH
modifierTitre(
  id: number,
  titre: string
) {

  return this.http.patch<Livre>(
    `${this.apiUrl}/${id}`,
    {
      titre: titre
    }
  );

}
```

```
 // DELETE
 supprimerLivre(id: number) {

  return this.http.delete(
   `${this.apiUrl}/${id}`
  );

 }

}
```

5. GET — Afficher les livres

Dans le composant livres.ts :

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

import { LivreService } from '../../services/livre.service';

import { Livre } from '../../models/livre';

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

 service = inject(LivreService);

 livres: Livre[] = [];
```

```
erreur = '';

chargement = false;



chargerLivres() {

 this.chargement = true;

 this.service.getLivres()
  .subscribe({

    next: (data) => {

     this.livres = data;

     this.chargement = false;

    },

    error: (erreur) => {

     console.log(erreur);

     this.erreur =
      'Impossible de charger les livres.';

     this.chargement = false;

    }
```

```
    });

  }

}
```

7. Appeler automatiquement GET

On peut utiliser OnInit :

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

Puis :

```
export class LivresComponent
  implements OnInit {

  service = inject(LivreService);

  livres: Livre[] = [];

  ngOnInit() {

    this.chargerLivres();

  }

}
```

Ainsi, dès que le composant est affiché :

Composant

↓

ngOnInit()

↓

chargerLivres()

↓

Service

↓

GET

↓

API

8. HTML — Afficher les livres

Dans livres.html :

<h1> Gestion de la bibliothèque</h1>

<button

(click)="chargerLivres()">

Charger les livres

</button>

@if (chargement) {

 Chargement des livres...

}

@if (erreur) {

```
 <p class="erreur">
  {{ erreur }}
 </p>

}

<div class="liste">

 @for (livre of livres; track livre.id) {

  <div class="livre">

   <h2>
    {{ livre.titre }}
   </h2>
```

9. POST — Ajouter un livre

Ajouter dans le composant :

```
titre = '';

auteur = '';

categorie = '';

annee = 0;
```

Puis :

```
ajouterLivre() {
```

```
const livre: Livre = {

 id: 0,

 titre: this.titre,

 auteur: this.auteur,

 categorie: this.categorie,

 annee: this.annee

};



this.service.ajouterLivre(livre)
 .subscribe({

  next: (data) => {

   console.log(
    'Livre ajouté',
    data
   );

   this.livres.push(data);

  },

  error: (erreur) => {
```

```
      console.log(erreur);

      this.erreur =
        'Impossible d’ajouter le livre.';

    }

  });

}
```

10. Formulaire HTML

Comme nous utilisons ngModel, importer :

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

et :

```
imports: [FormsModule]
```

Dans le HTML :

```
<h2>Ajouter un livre</h2>

<input
 type="text"
 [(ngModel)]="titre"
 placeholder="Titre"
>

<input
 type="text"
 [(ngModel)]="auteur"
 placeholder="Auteur"
>
```

```
<input
 type="text"
 [(ngModel)]="categorie"
 placeholder="Catégorie"
>

<input
 type="number"
 [(ngModel)]="annee"
 placeholder="Année"
>

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

 Ajouter le livre

</button>
```

11. PUT — Modifier complètement

Créer :

```
modifierLivre(livre: Livre) {

 const nouveauLivre: Livre = {

   id: livre.id,

   titre: this.titre,

   auteur: this.auteur,
```

```
  categorie: this.categorie,

  annee: this.annee

 };



 this.service.modifierLivre(
  livre.id,
  nouveauLivre
 )
 .subscribe({

  next: (data) => {

   console.log(
    'Livre modifié',
    data
   );

  },

  error: (erreur) => {

   console.log(erreur);

  }

 });

}
```

12. PATCH — Modifier uniquement le titre

Créer :

```
modifierTitre(livre: Livre) {

 const nouveauTitre =
  prompt('Nouveau titre :');

 if (!nouveauTitre) {

  return;

 }



 this.service.modifierTitre(
  livre.id,
  nouveauTitre
 )
 .subscribe({

  next: (data) => {

   console.log(
    'Titre modifié',
    data
   );

  },

  error: (erreur) => {
```

```
    console.log(erreur);

  }

 });

}
```

Et dans le HTML :

```
<button
 (click)="modifierTitre(livre)">

 Modifier le titre

</button>
```

Le serveur reçoit uniquement :

```
{
 "titre": "Nouveau titre"
}
```

Donc :

PATCH est utilisé lorsque nous ne voulons modifier qu'une partie de la ressource.

13. DELETE — Supprimer

Dans le composant :

```
supprimerLivre(id: number) {

 const confirmation =
  confirm(
   'Voulez-vous vraiment supprimer ce livre ?'
  );

 if (!confirmation) {
```

```
  return;

 }


 this.service.supprimerLivre(id)
  .subscribe({

   next: () => {

    this.livres =
     this.livres.filter(
      livre => livre.id !== id
     );

   },

   error: (erreur) => {

    console.log(erreur);

    this.erreur =
     'Impossible de supprimer le livre.';

   }

  });

}
```

Dans le HTML :

```
<button
 (click)="supprimerLivre(livre.id)">

 Supprimer

</button>
```

14. CSS — livres.css

Voici un CSS simple mais propre :

```
h1 {
 text-align: center;
 margin-bottom: 30px;
}

h2 {
 margin-bottom: 15px;
}

input {
 display: block;
 width: 100%;
 max-width: 400px;
 padding: 10px;
 margin-bottom: 10px;
 border: 1px solid #ccc;
 border-radius: 6px;
}

button {
 padding: 9px 14px;
 margin: 5px;
```

```
  border: none;
  border-radius: 6px;
  cursor: pointer;
}

.liste {
  display: grid;
  grid-template-columns: repeat(
    auto-fit,
    minmax(280px, 1fr)
  );
  gap: 20px;
  margin-top: 30px;
}

.livre {
  padding: 20px;
  border-radius: 10px;
  background: #f4f4f4;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
}

.livre p {
  margin: 8px 0;
}

.erreur {
  padding: 10px;
  margin-top: 15px;
  border-radius: 6px;
  background: #ffe0e0;
```

```
  color: #b00020;
}
```

Module 5 — Introduction à RxJS

1. Qu'est-ce que RxJS ?

RxJS signifie **Reactive Extensions for JavaScript**.

C'est une bibliothèque utilisée par Angular pour travailler avec des **flux de données**, notamment lorsque les données arrivent de manière asynchrone ou peuvent changer dans le temps.

Par exemple :

- récupérer des données depuis une API ;
- écouter ce que tape un utilisateur ;
- réagir à un événement ;
- surveiller une donnée qui change ;
- enchaîner plusieurs opérations asynchrones.

2. Les deux grandes parties que nous allons étudier

Dans ce module, nous allons rencontrer deux types d'éléments.

A. Les objets qui représentent ou gèrent les flux

Observable

Subject

BehaviorSubject

ReplaySubject

Ils permettent de **représenter, émettre ou conserver des valeurs**.

B. Les opérateurs RxJS

map

filter

switchMap

mergeMap

tap

debounceTime

Les opérateurs permettent de **travailler sur les flux** : transformer, filtrer, attendre, enchaîner des opérations, etc.

3. Observable

Un **Observable** représente un **flux de données**.

Il peut émettre une ou plusieurs valeurs au cours du temps.

Un Observable, c'est comme un système de suivi qui peut nous envoyer des informations au fur et à mesure qu'elles arrivent.

Ex :livraison d'un colis

Imagine que tu commandes un colis sur Internet.

Tu ne sais pas exactement quand il va arriver. Mais tu peux **suivre son statut** :

Exemple pratique :

```
getUsers(): Observable<User[]> {
  return this.http.get<User[]>('/api/users');
}
```

Ici, getUsers() ne retourne pas directement les utilisateurs.

Il retourne :

Observable<User[]>

L'Observable va ensuite émettre les utilisateurs lorsque la réponse sera disponible.

Pour écouter ce flux :

```
this.userService.getUsers().subscribe(users => {
  console.log(users);
});
```

On peut retenir :

Observable = flux de données

subscribe() = s'abonner au flux

subscribe()

Imagine un **groupe WhatsApp** .

Le groupe envoie des messages :

Groupe

│

├── "Réunion à 10h"

├── "N'oubliez pas vos ordinateurs"

└── "La réunion est annulée"

Toi, tu décides de **suivre le groupe**.

C'est l'idée de subscribe() :

« Je m'abonne pour recevoir les informations qui arrivent. »

4. Subject

Un **Subject** est particulier car il peut à la fois :

- recevoir/envoyer des valeurs ;
- être écouté

Subject est un moyen d'envoyer une information à plusieurs personnes qui l'écoutent

Ex :

Imagine un **professeur qui annonce une information à toute la classe**.

Le professeur dit :

« Demain, le cours commence à 9h. »

Plusieurs étudiants écoutent l'annonce.

Exemple pratique :

```
const subject = new Subject<string>();
```

```
subject.subscribe(value => {
  console.log(value);
});


subject.next('Bonjour');
```

Le next() envoie une valeur et le subscribe() la reçoit.

Subject

```
  │
  ├── next() → envoyer
  │
  └── subscribe() → écouter
```

5. BehaviorSubject

Le **BehaviorSubject** ressemble au Subject, mais il possède toujours une **valeur actuelle**.

Imagine un **tableau d'affichage dans une classe** 📋.

Le professeur écrit :

« Le cours commence à 9h. »

Un étudiant arrive **plus tard**.

Même s'il n'était pas là quand le professeur a écrit l'information, il peut regarder le tableau et voir **la dernière information actuelle**.

```
const user$ = new BehaviorSubject<string>('Inconnu');
```

Sa valeur initiale est :

Inconnu

Puis :

```
user$.next('Ali');
```

La valeur actuelle devient :

Ali

Un nouvel abonné reçoit immédiatement la valeur actuelle.

BehaviorSubject

```
    │
```

└— valeur actuelle

On l'utilise souvent pour représenter un **état partagé** dans une application Angular.

6. ReplaySubject

Le **ReplaySubject** peut mémoriser plusieurs valeurs précédentes.

Imagine une **caméra qui enregistre les 3 dernières annonces** d'un professeur. 🎥

Le professeur annonce :

1 "Cours à 9h"

2 "Prenez votre ordinateur"

3 "Exercice page 20"

Un étudiant arrive **plus tard**.

Avec un ReplaySubject(3), il peut recevoir les **3 dernières informations**, même s'il n'était pas là au moment où elles ont été envoyées.

```
const history$ = new ReplaySubject<string>(2);
```

Le 2 signifie qu'il conserve les **2 dernières valeurs**.

```
history$.next('A');

history$.next('B');

history$.next('C');
```

7. Les opérateurs RxJS

Les opérateurs permettent de **transformer et contrôler un flux**.

Ils sont généralement utilisés avec :

.pipe(...)

Exemple :

```
products$.pipe(
  map(product => product.name)
);
```

8. map

map permet de **transformer une valeur**.

```
of(1, 2, 3).pipe(
  map(x => x * 2)
);
```

Résultat :

1 → 2

2 → 4

3 → 6

9. filter

filter permet de **garder uniquement certaines valeurs**.

```
of(1, 2, 3, 4).pipe(
  filter(x => x > 2)
);
```

Résultat :

3

4

10. tap

tap permet d'**observer une valeur sans la modifier**.

```
products$.pipe(
  tap(products => {
    console.log(products);
  })
);
```

Les données continuent leur chemin normalement.

Observable

↓

tap

↓

Observable

11. debounceTime

debounceTime permet d'**attendre avant de laisser passer une valeur**.

Très utile pour une recherche.

```
search$.pipe(
  debounceTime(500)
);
```

Si l'utilisateur tape :

A

Al

Ali

Alice

rapidement, on attend 500 ms après la dernière saisie avant de continuer.

A → Al → Ali → Alice

↓

500 ms

↓

recherche

12. switchMap

switchMap permet de lancer un nouvel Observable à partir d'une valeur.

Il est très utile pour les recherches.

```
search$.pipe(
  switchMap(search =>
    this.service.searchProducts(search)
  )
);
```

Si une nouvelle recherche arrive, on passe à la nouvelle opération.

"Ali" → recherche Ali

"Ali" + "ce" → recherche Alice

↓

dernière recherche

13. mergeMap

mergeMap permet également de lancer un Observable, mais il **ne remplace pas les opérations précédentes**.

```
products$.pipe(
 mergeMap(product =>
  this.service.getDetails(product.id)
 )
);
```

Plusieurs opérations peuvent donc être exécutées en parallèle.

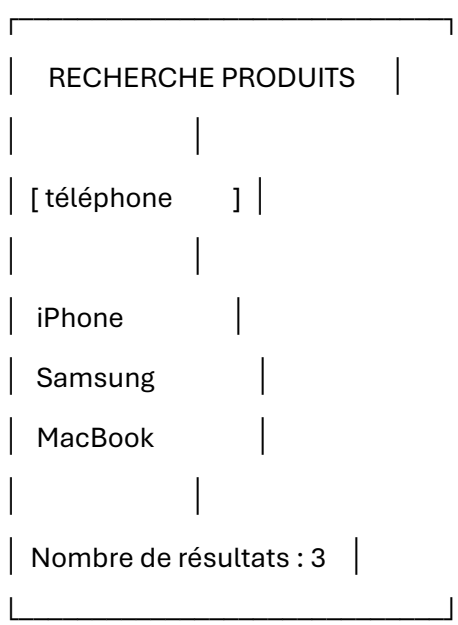
Produit 1 → requête 1 ———→ résultat

Produit 2 → requête 2 ——→ résultat

Produit 3 → requête 3 ————→ résultat

Projet : recherche de produits

Notre application fera ceci :



Nous allons apprendre RxJS **à travers ce projet**, plutôt que d'apprendre les opérateurs séparément.

1. Créer le projet

Dans le terminal :

ng new rxjs-demo

cd rxjs-demo

ng serve

Choisis les options proposées par Angular.

2. Structure du projet

Nous allons avoir :

src/app/

```
│
├── models/
│   └── product.ts
│
├── services/
│   └── productS.ts
│
├── app.ts
├── app.html
└── app.css
```

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>code .

D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g i models/product
CREATE src/app/models/product.ts (31 bytes)
```

```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng g s services/productS
CREATE src/app/services/product-s.spec.ts (348 bytes)
CREATE src/app/services/product-s.ts (121 bytes)

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

3. Le modèle Product

Crée :

src/app/models/product.ts

export interface Product {

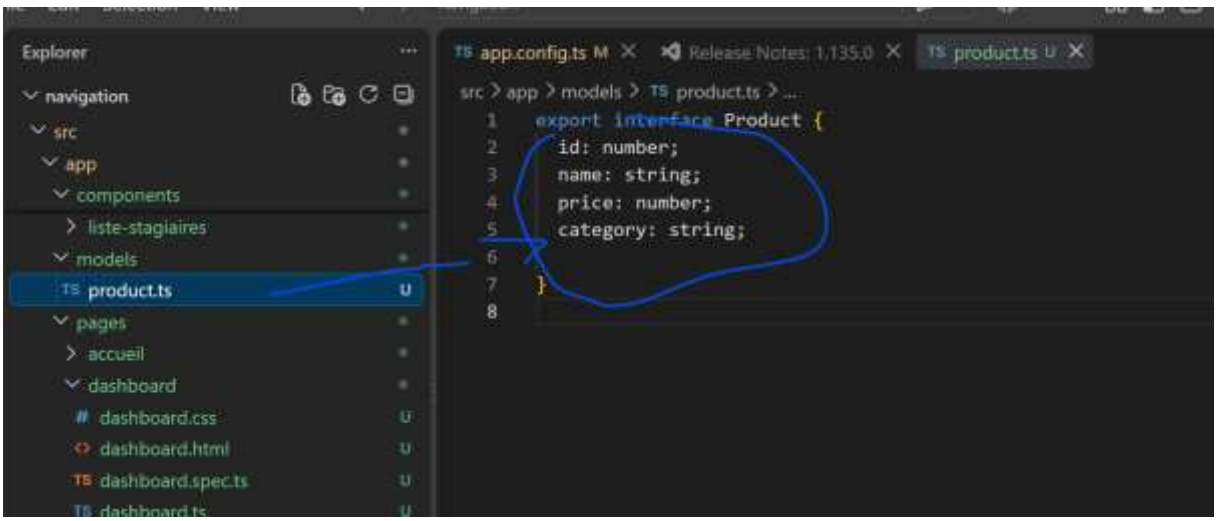
id: number;

name: string;

price: number;

category: string;

}



Nous avons donc :

Product

├── id

├── name

├── price

└── category

4. Configurer HttpClient

src/app/app.config.ts

mets :

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

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

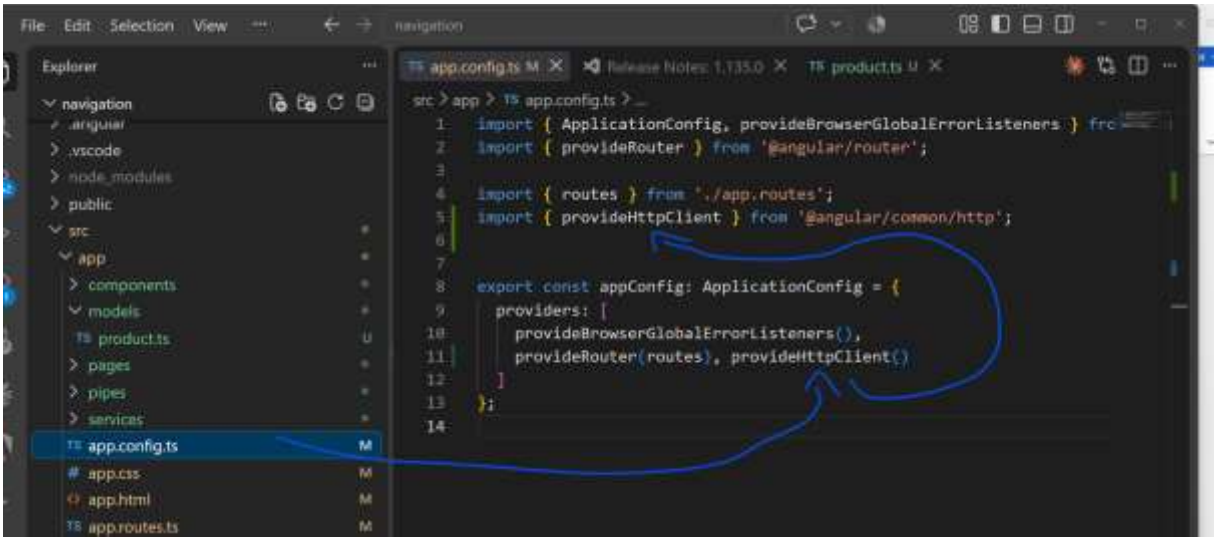
export const appConfig: ApplicationConfig = {

 providers: [

  provideHttpClient()

 ]

};



Cela permet d'utiliser HttpClient.

5. Créer le service

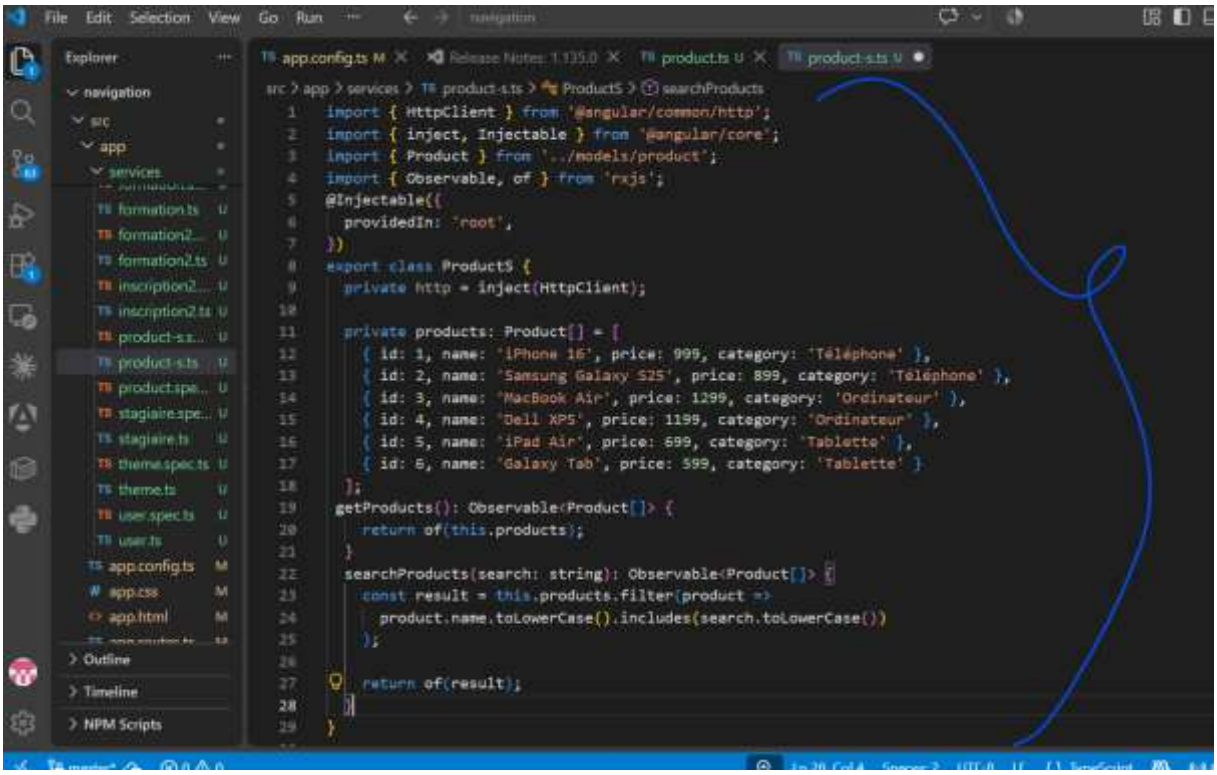
Crée :

src/app/services/productS.ts

```
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, of } from 'rxjs';
import { Product } from '../models/product';

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

 private http = inject(HttpClient);

 private products: Product[] = [
  { id: 1, name: 'iPhone 16', price: 999, category: 'Téléphone' },
  { id: 2, name: 'Samsung Galaxy S25', price: 899, category: 'Téléphone' },
  { id: 3, name: 'MacBook Air', price: 1299, category: 'Ordinateur' },
  { id: 4, name: 'Dell XPS', price: 1199, category: 'Ordinateur' },
  { id: 5, name: 'iPad Air', price: 699, category: 'Tablette' },
  { id: 6, name: 'Galaxy Tab', price: 599, category: 'Tablette' }
 ];

 getProducts(): Observable<Product[]> {
  return of(this.products);
 }

 searchProducts(search: string): Observable<Product[]> {
  const result = this.products.filter(product =>
   product.name.toLowerCase().includes(search.toLowerCase())
  );

  return of(result);
```

}

}



Of : « Prend mon tableau this.products et transforme-le en Observable pour que je puisse m'y abonner. »

6. Notre premier subscribe()

Dans app.component.ts :

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

import { ProductService } from './services/product.service';

@Component({

selector: 'app-root',

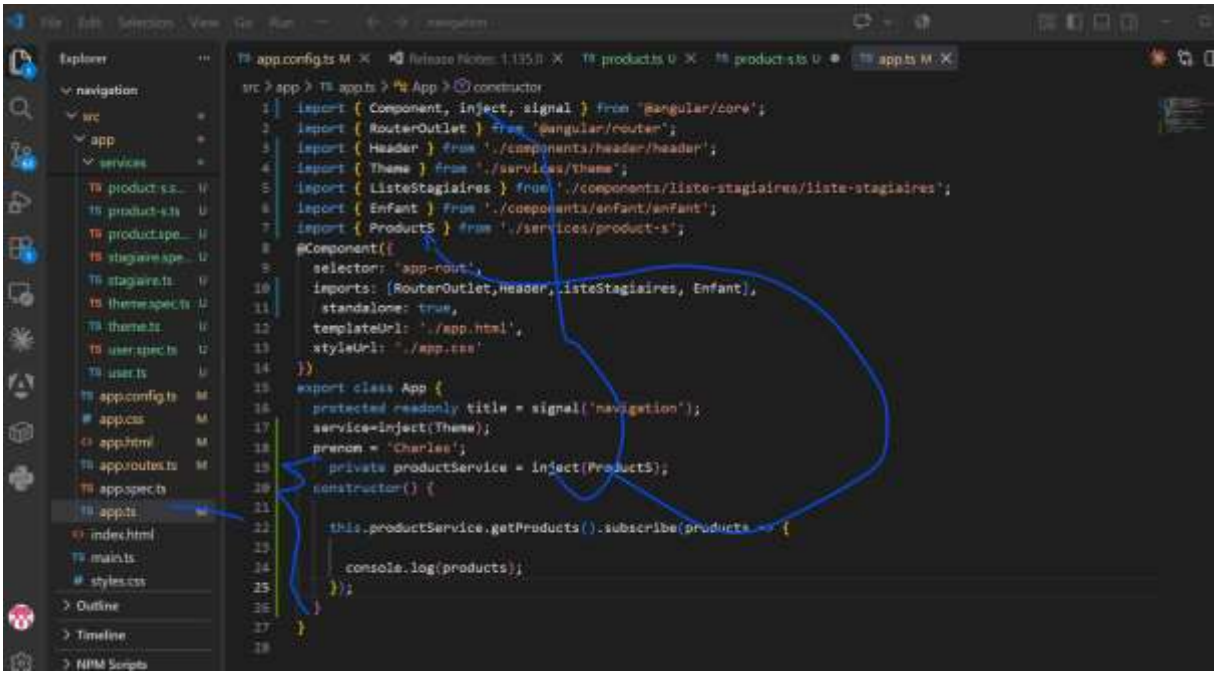
templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

```
 private productService = inject(ProductService);

 constructor() {

  this.productService.getProducts().subscribe(products => {

    console.log(products);

  });

 }
}
```



Dans la console du navigateur :

```
[
 { id: 1, name: 'iPhone 16', ... },
 { id: 2, name: 'Samsung Galaxy S25', ... },
 ...
]
```

this.productService.getProducts()

retourne :

Observable<Product[]>

Puis :

.subscribe(products => {

  ...

});

dit :

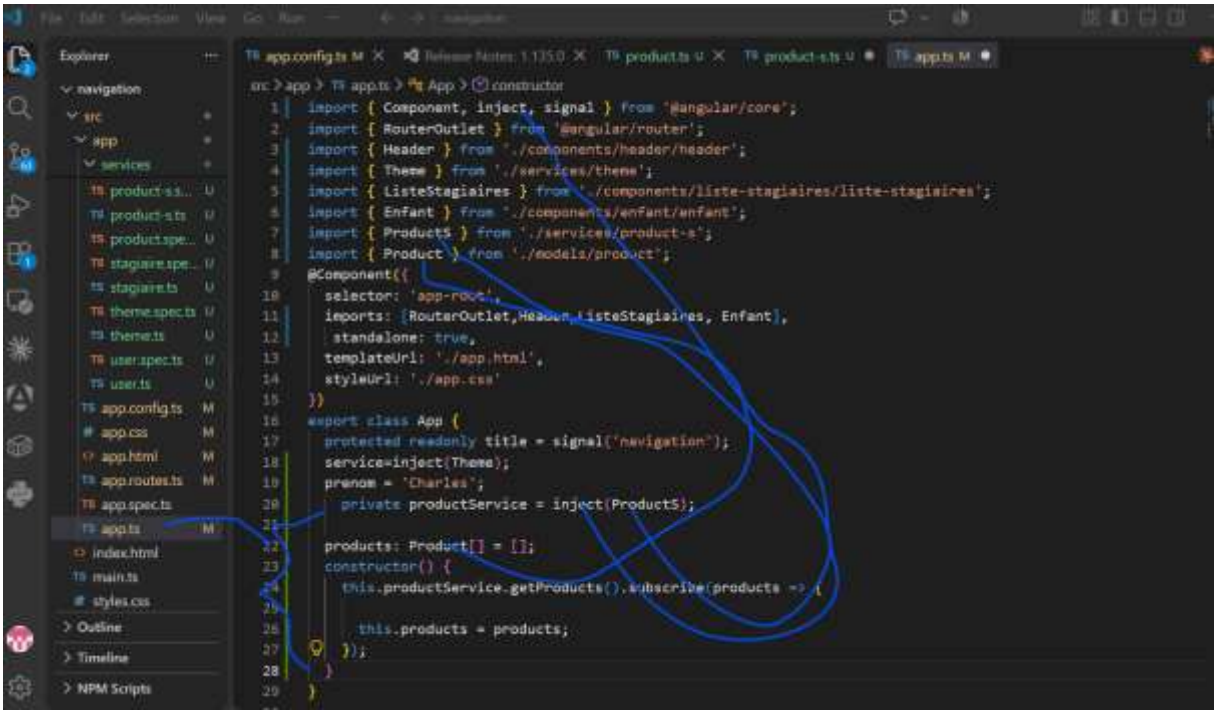
« Je m'abonne à l'Observable. Quand il émet les produits, je les récupère dans products. »

7. Afficher les produits

Modifie app.component.ts :

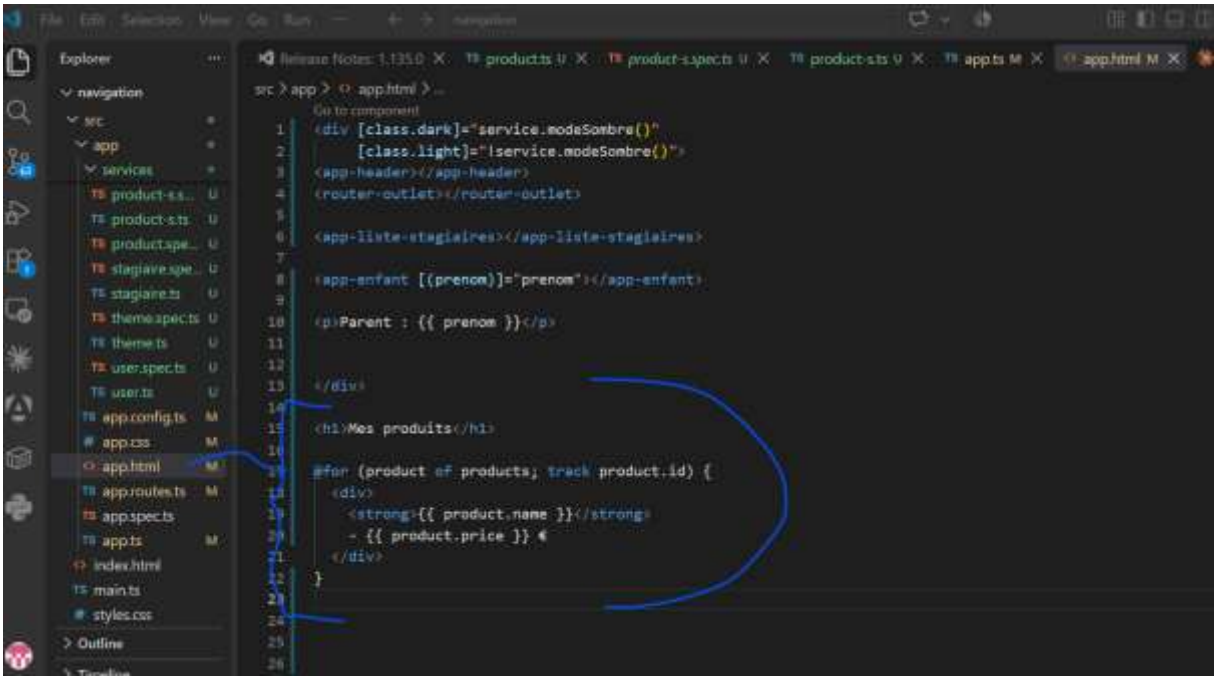
```
import { Component, inject } from '@angular/core';
import { ProductService } from './services/product.service';
import { Product } from './models/product';

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

 private productService = inject(ProductService);

 products: Product[] = [];

 constructor() {

  this.productService.getProducts().subscribe(products => {

   this.products = products;
```

```
  });


 }
}
```

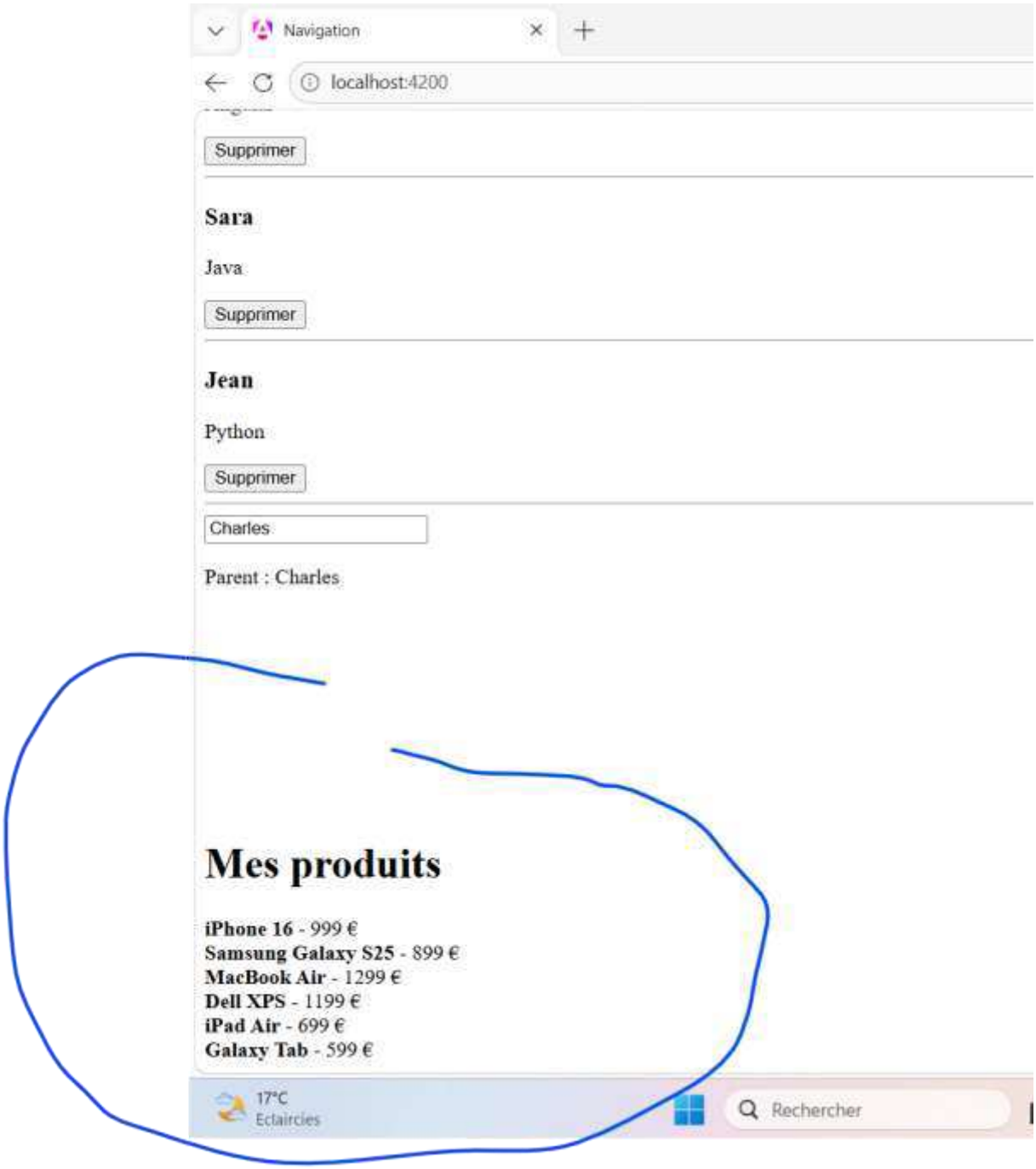


Puis app.component.html :

```
<h1>Mes produits</h1>


@for (product of products; track product.id) {
 <div>
  <strong>{{ product.name }}</strong>
  - {{ product.price }} €
 </div>
}
```




```
D:\CFITECH\Angular\Cours angular  2026\Exercices\navigation>ng serve -o
Initial chunk files | Names         | Raw size
main.js             | main          | 60.25 kB |
styles.css          | styles        | 95 bytes |

                    | Initial total | 60.34 kB

Application bundle generation complete. [3.189 seconds] - 2026-08-31T19:48:41.623Z

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
21:48:41 [vite] (client) Re-optimizing dependencies because vite config has changed
  ➜  Local:   http://localhost:4200/
  ➜  press h + enter to show help
```


Navigation
localhost:4200
Supprimer
Sara
Java
Supprimer
Jean
Python
Supprimer
Charles
Parent : Charles
Mes produits
iPhone 16 - 999 €
Samsung Galaxy S25 - 899 €
MacBook Air - 1299 €
Dell XPS - 1199 €
iPad Air - 699 €
Galaxy Tab - 599 €
17°C
Eclaircies
Rechercher