



Manipulation de composants dans



United Nations
Educational, Scientific and
Cultural Organization



UNESCO Chair
"Project-based learning"
ESPRIT School of engineering, Tunisia



EUR-ACE®

Délivrée par la
Commission
des Titres
d'Ingénieur



CONFÉRENCE DES
GRANDES
ÉCOLES



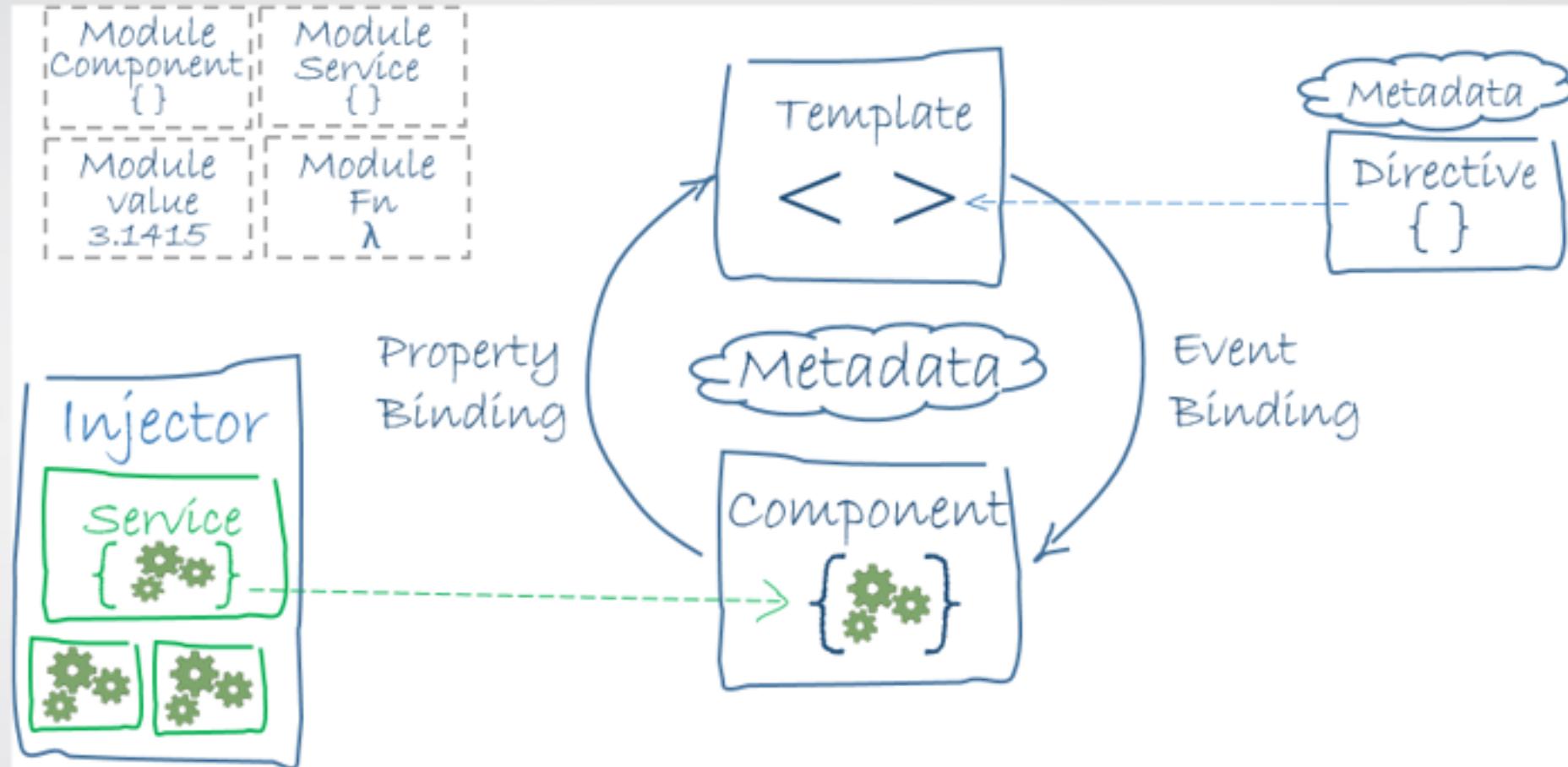
- ▶ Architecture d'un projet Angular classique (non-standalone)
- ▶ Système de modularité (NgModules)
- ▶ Les composants dans Angular
- ▶ Data-Binding
- ▶ Les directives
- ▶ Les pipes
- ▶ Cycle de vie d'un composant



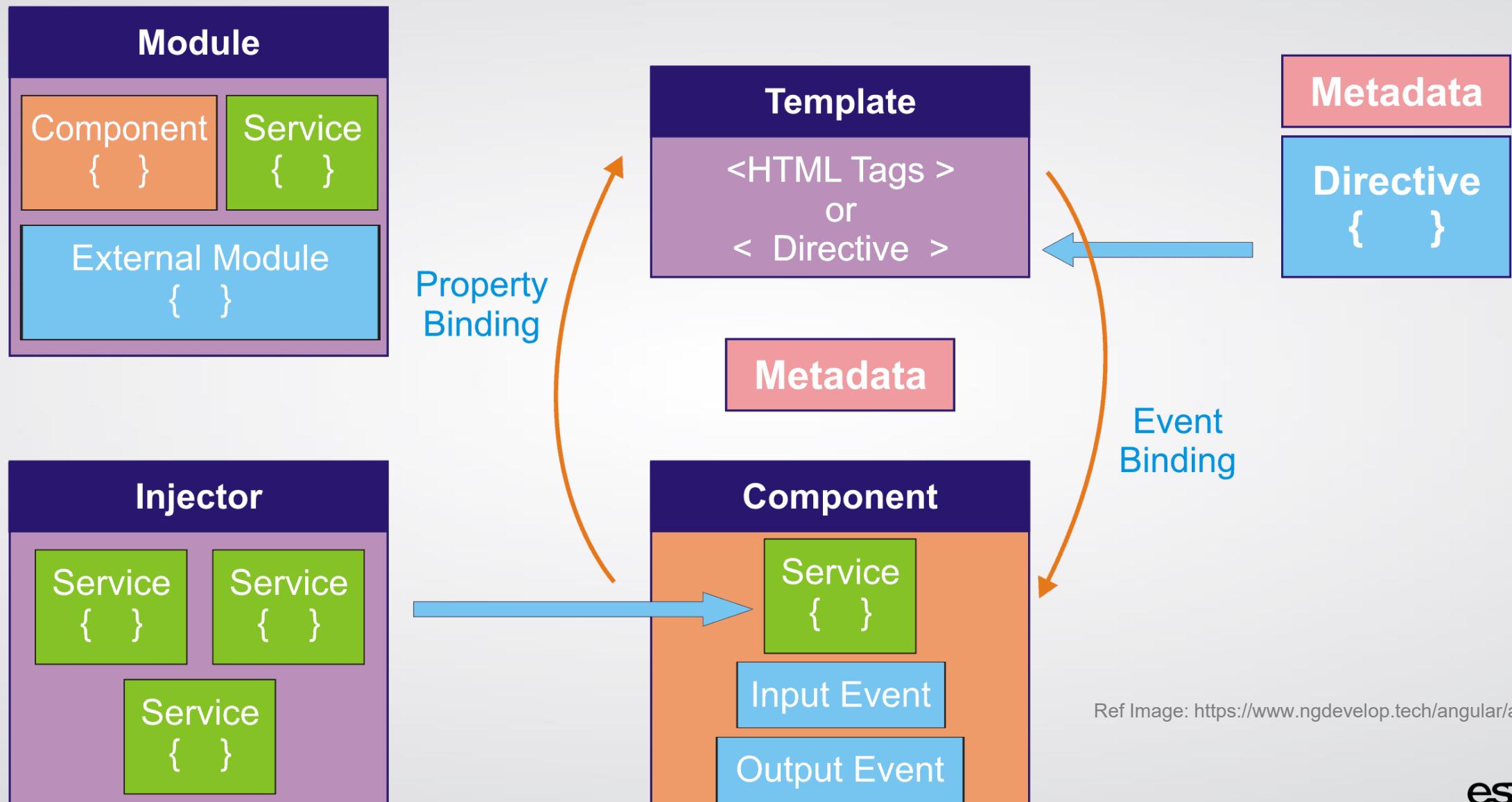


Architecture d'un projet Angular classique (non-standalone)

► Architecture Angular



► Architecture Angular



Ref Image: <https://www.ngdevelop.tech/angular/architecture/>

Système de modularité NgModules



► Angular est modulaire



- Une application Angular est modulaire.
- Elle possède au moins un module appelé « module racine » ou « root module »
- Elle peut contenir d'autres modules à part le module racine.
- Par convention, le module racine est appelé « **AppModule** » et se trouve dans un fichier appelé « **app.module.ts** »

The image shows a file explorer interface with two panes. The left pane displays the root directory 'FIRSTAPP' with subfolders '.vscode', 'node_modules', and 'src'. The 'src' folder is expanded, revealing 'app', 'assets', 'favicon.ico', 'index.html', 'main.ts', 'styles.css', '.editorconfig', '.gitignore', 'angular.json', 'package-lock.json', 'package.json', 'README.md', 'tsconfig.app.json', 'tsconfig.json', and 'tsconfig.spec.json'. A red arrow points from the 'app' folder in the main tree to the right pane, which shows a detailed view of the 'app' folder's contents. The right pane lists 'app-routing.ts', 'app.component.ts', 'app.component.ts', 'app.module.ts', 'app.module.ts', 'app.module.ts', 'app.module.ts', and 'app.module.ts'.

File	Type	Last Modified
app-routing.ts	TS	2023-09-12 14:45:23
# app.compone...	#	2023-09-12 14:45:23
<> app.compone...	<>	2023-09-12 14:45:23
TS app.compone...	TS	2023-09-12 14:45:23
TS app.compone...	TS	2023-09-12 14:45:23
TS app.compone...	TS	2023-09-12 14:45:23
TS app.module.ts	TS	2023-09-12 14:45:23



Système de modularité : NgModules



- Le système de modularité dans Angular est appelé **NgModules**.
- Un module peut être exporté sous forme de classe.
- La classe qui décrit le module Angular est une classe décorée par **@NgModule**.

Exemple: `FormsModule`, `HttpClientModule`, `RouterModule`

Les **décorateurs** sont des fonctions qui modifient les classes TypeScript. Ils sont essentiellement utilisés pour attacher des métadonnées à des classes afin que le système connaisse la configuration de ces classes et leur fonctionnement.

```
src > app > ts app.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6
7  @NgModule({
8    declarations: [
9      AppComponent
10    ],
11    imports: [
12      BrowserModule,
13      AppRoutingModule
14    ],
15    providers: [],
16    bootstrap: [AppComponent]
17  })
18  export class AppModule { }
```



NgModule - Métadonnées



- **Declarations:** les composants – directives – pipes utilisés par ce module
- **Imports:** les modules internes ou externes utilisés dans ce module
- **Providers:** Les services utilisés
- **Bootstrap:** indique quel composant racine Angular doit initialiser et afficher en premier. C'est le **point d'entrée visuel** de l'application.

RQ: Le tableau **bootstrap** est utilisé uniquement dans le module racine (souvent AppModule). Les autres modules (fonctionnels ou partagés) n'ont pas besoin de bootstrap.

```
▽ @NgModule({
  ▽ declarations: [
    AppComponent
  ],
  ▽ imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```



En résumé



- Une application Angular possède au moins un module, le root module, nommé par convention **AppModule**.
- Un module Angular est défini simplement avec une classe (*généralement vide*) et le décorateur `NgModule`.
- Un module Angular est un mécanisme permettant de :
 - regrouper des composants (mais aussi des services, directives, pipes etc...),
 - définir leurs dépendances
 - définir leur visibilité.

Les composants dans Angular

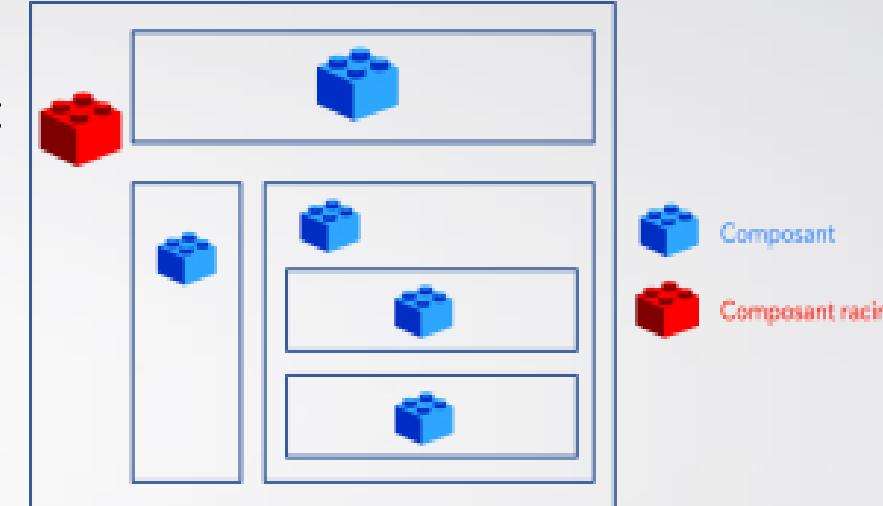




Composant Angular - Définition



- Un composant web est un élément personnalisé et réutilisable.
- Un composant peut correspondre à une page entière ou un élément de l'interface (menu, header, modal, datepicker, etc...).
- Une application a au moins un composant racine: Root Component.
- Une classe décorée par le décorateur `@Component` (`angular/core`) qui la définit comme étant un composant et spécifie sa métadonnée qui indique comment il sera utilisé.
- Angular prend en charge deux manières de rendre un composant accessible à d'autres composants : en tant que **composant standalone** ou au sein d'un **NgModule**.



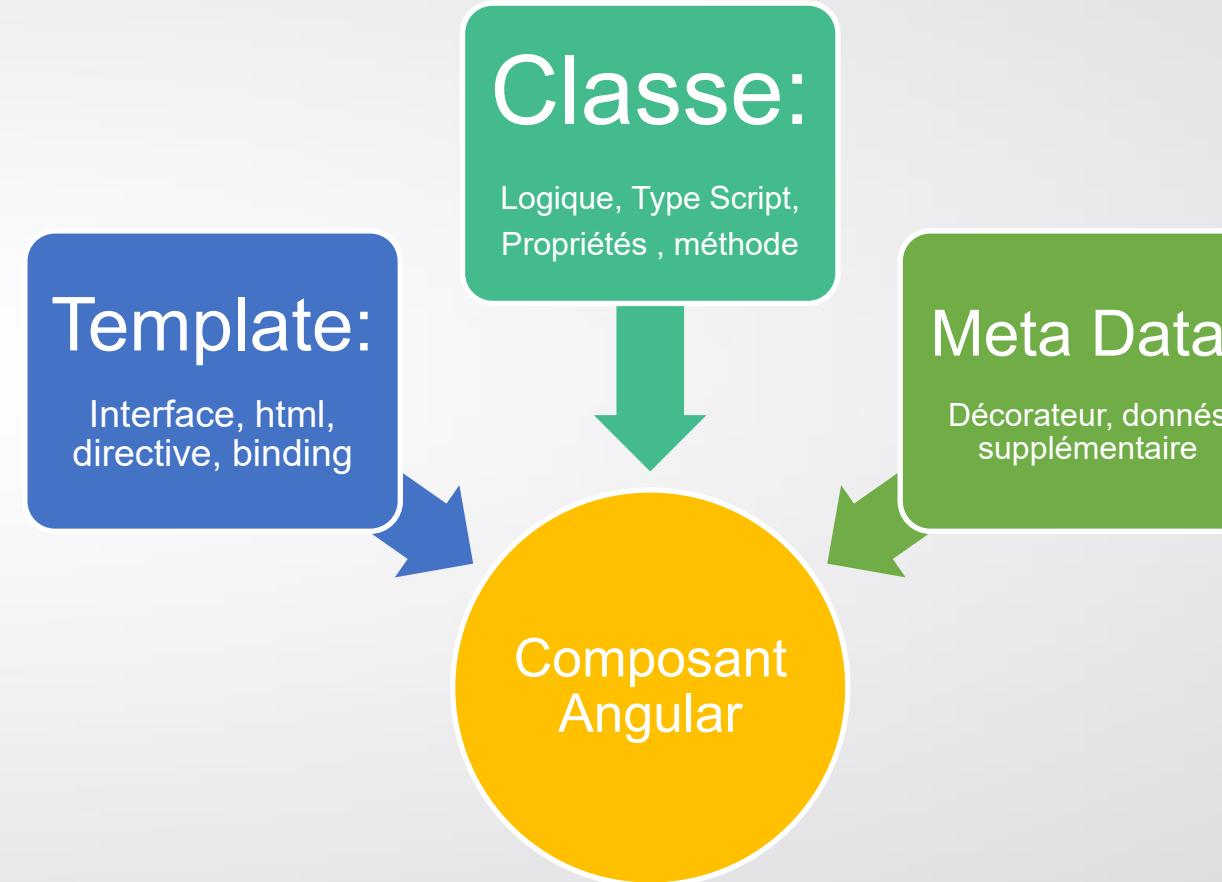


Composant Angular - Composition



Un composant Angular contient:

- Un **Template** contenant l'interface utilisateur en HTML. Nous utiliserons le databinding afin de rendre la vue dynamique,
- **Classe** (class) contenant le code associé à la vue , des propriétés et méthodes logiques qui seront utilisées dans la vue,
- Des **Metadata** nous permettant de définir la classe comme étant un composant Angular (component).





Composant non-standalone : Création



- Générer un nouveau composant dans votre Application angular:

ng generate component header ou ng g c header

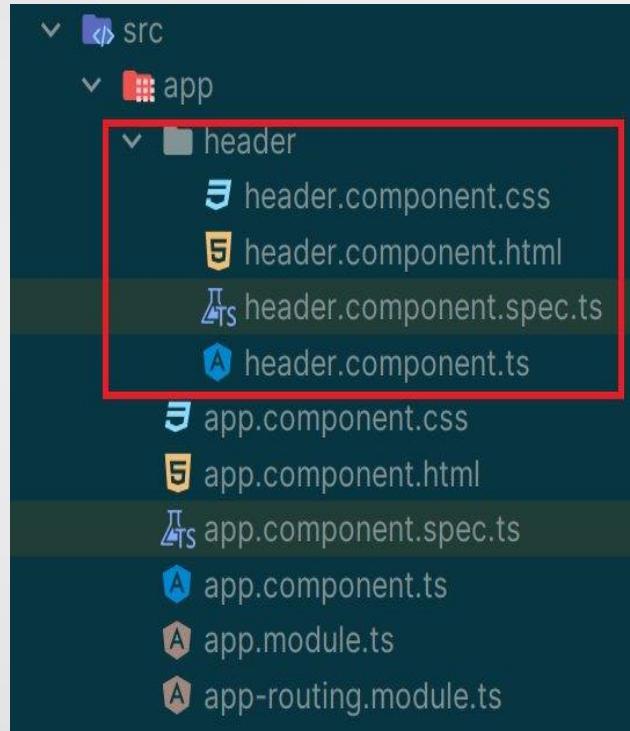
```
PS C:\Users\PC\Desktop\cours\projet> ng g c header
CREATE src/app/header/header.component.html (21 bytes)
CREATE src/app/header/header.component.spec.ts (559 bytes)
CREATE src/app/header/header.component.ts (202 bytes)
CREATE src/app/header/header.component.css (0 bytes)
UPDATE src/app/app.module.ts (475 bytes)
```



Composant non-standalone : Création



- Le composant header est dans votre projet:



```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';

1+ usages  ↗ WiemAouididi
@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent (highlighted with a red box)
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```



Composant non-standalone : Déclaration



Le décorateur `@Component (angular/core)` contient par défaut (génération par défaut de CLI) 3 tags:

- « **selector** » : Le sélecteur indique la déclaration qui permet d'insérer le composant dans le document HTML.
- « **templateUrl** »: Permet d'associer un fichier externe HTML contenant la structure de la vue du composant
- « **styleUrls** »: Spécifie les feuilles de styles CSS associées à ce composant.

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

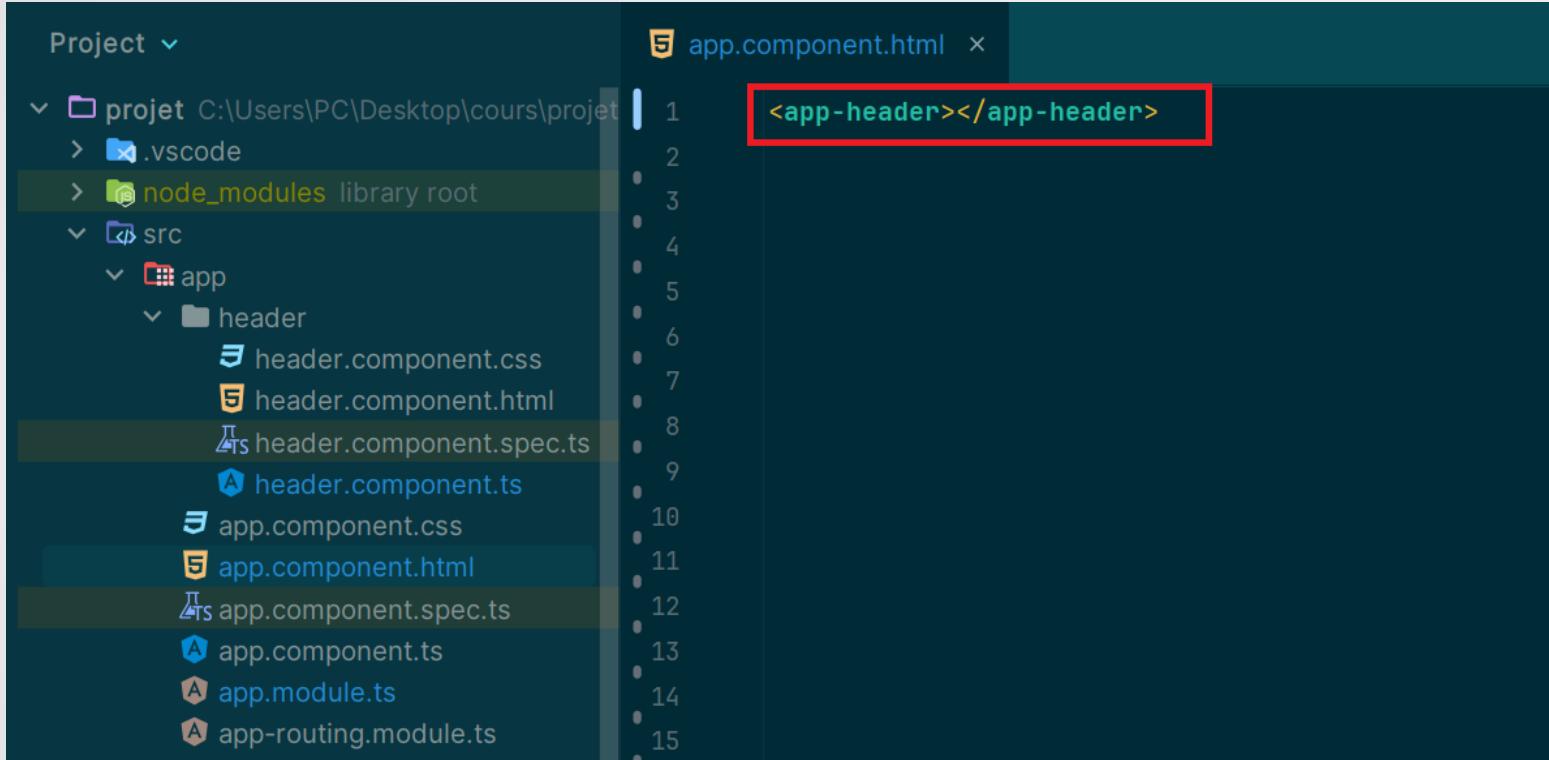
1+ usages  ↗ WiemAoudidi
@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent { }
```



Composant non-standalone : Affichage



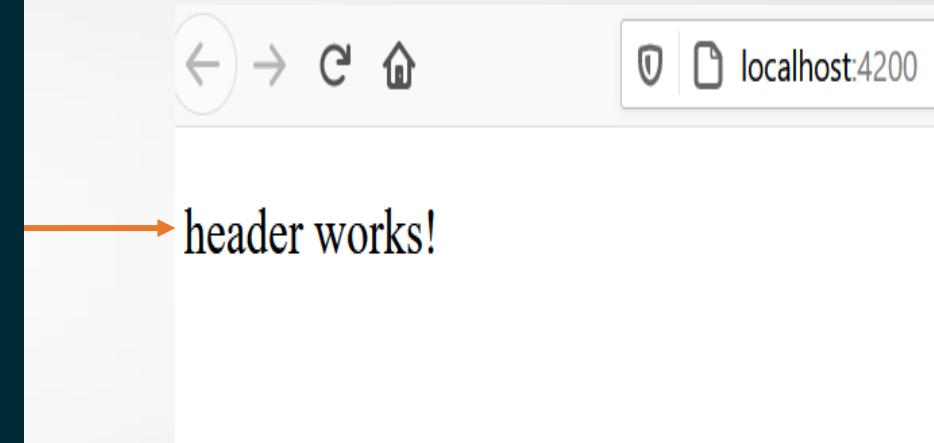
- Le sélecteur permet de faire appel à votre composant:



The screenshot shows the VS Code interface with the project structure on the left and the `app.component.html` file open on the right. The file contains the following code:

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

The selector `<app-header></app-header>` is highlighted with a red box.





Composant standalone : Création



- Générer un nouveau composant dans votre Application angular:

ng generate component header --standalone ou ng g c header --standalone

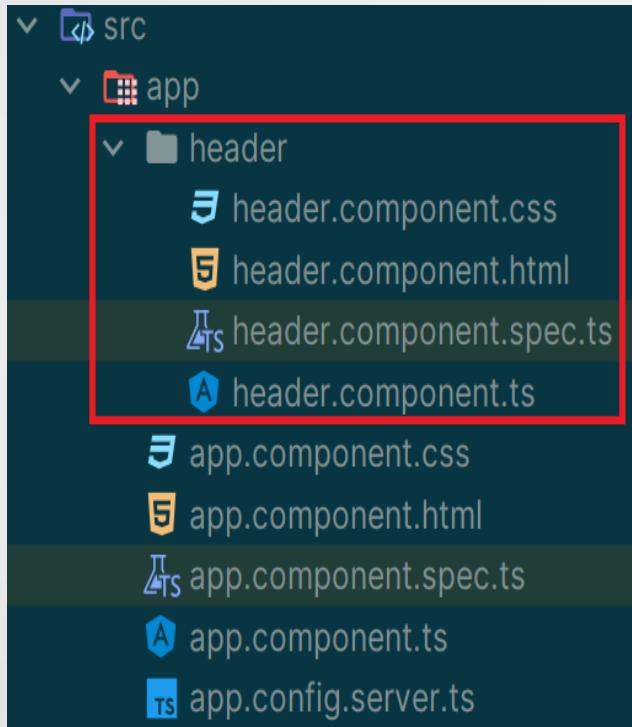
```
PS C:\Users\PC\Desktop\angular\projet> ng g c header --standalone
CREATE src/app/header/header.component.html (22 bytes)
CREATE src/app/header/header.component.spec.ts (615 bytes)
CREATE src/app/header/header.component.ts (246 bytes)
CREATE src/app/header/header.component.css (0 bytes)
```



Composant standalone : Création



- Le composant header est dans votre projet:



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

1+ usages   ↗ WiemAouididi
@Component({
  selector: 'app-header',
  standalone: true,
  imports: [],
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent { }
```



Composant standalone : Déclaration



Le décorateur @Component (angular/core) contient:

- **Le sélecteur « selector »:** Il indique la déclaration qui permet d'insérer le composant dans le document HTML.
- **Standalone:** permet de définir le composant comme **autonome**, c'est-à-dire qu'il **ne dépend pas d'un NgModule** pour fonctionner.
- **Imports:** liste les **autres composants, directives ou modules** nécessaires au fonctionnement du composant (ex : CommonModule, FormsModule, etc..).
- **TemplateUrl:** permet d'associer un fichier externe HTML contenant la structure de la vue du composant.
- **StyleUrls:** spécifier les feuilles de styles CSS associées à ce composant.

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

1+ usages   ↗ WiemAouididi
@Component({
  selector: 'app-header',
  standalone: true,
  imports: [],
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent { }
```



Composant standalone : Affichage



- Angular permet d'utiliser des composants standalone sans passer par un NgModule. Dans ce cas, l'intégration se fait directement via le tableau **imports** dans le décorateur **@Component**.
- Exemple:

.. Fichier **app.component.ts**

```
app.component.ts ×

1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3 import { HeaderComponent } from './header/header.component';
4
5 1+ usages  ↗ WiemAouididi
6 @Component({
7   selector: 'app-root',
8   standalone: true,
9   imports: [RouterOutlet, HeaderComponent],
10  templateUrl: './app.component.html',
11  styleUrls: ['./app.component.css']
12})
13 export class AppComponent {
14   title: string = 'cours';
15 }
```

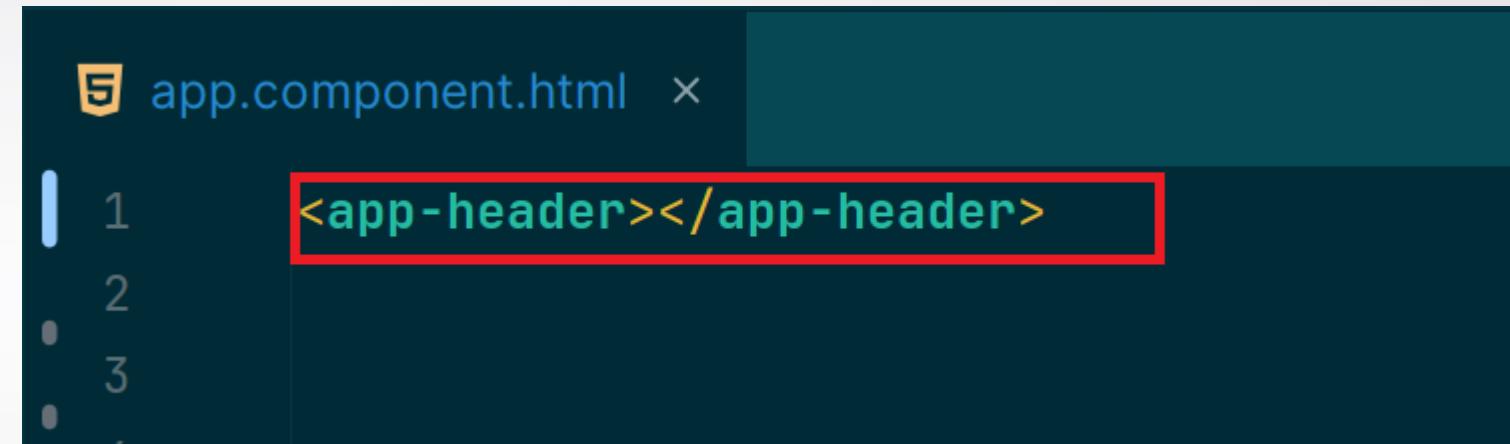


Composant standalone : Affichage



- Le sélecteur permet de faire appel à votre composant:

2. Fichier **app.component.html**



```
5 app.component.html x
1 <app-header></app-header>
2
3
```

3. **ng serve**





Comparaison des composants: v13- vs v14+



Élément	Angular 13 (et avant)	Angular ≥ 14 (standalone)
Module requis	Oui (NgModule obligatoire)	Non (composants standalone disponibles)
Fichier app.module.ts	Indispensable pour déclarer, importer et démarrer l'app	Facultatif, remplacé par bootstrapApplication()
Déclaration des composants	Via declarations dans NgModule	Pas nécessaire, composants autonomes déclarés via: @Component({ standalone: true })
Imports	Via imports dans NgModule	Via imports directement dans le composant standalone
Réutilisabilité	Liée à un module	Autonome et réutilisable partout



Data-Binding

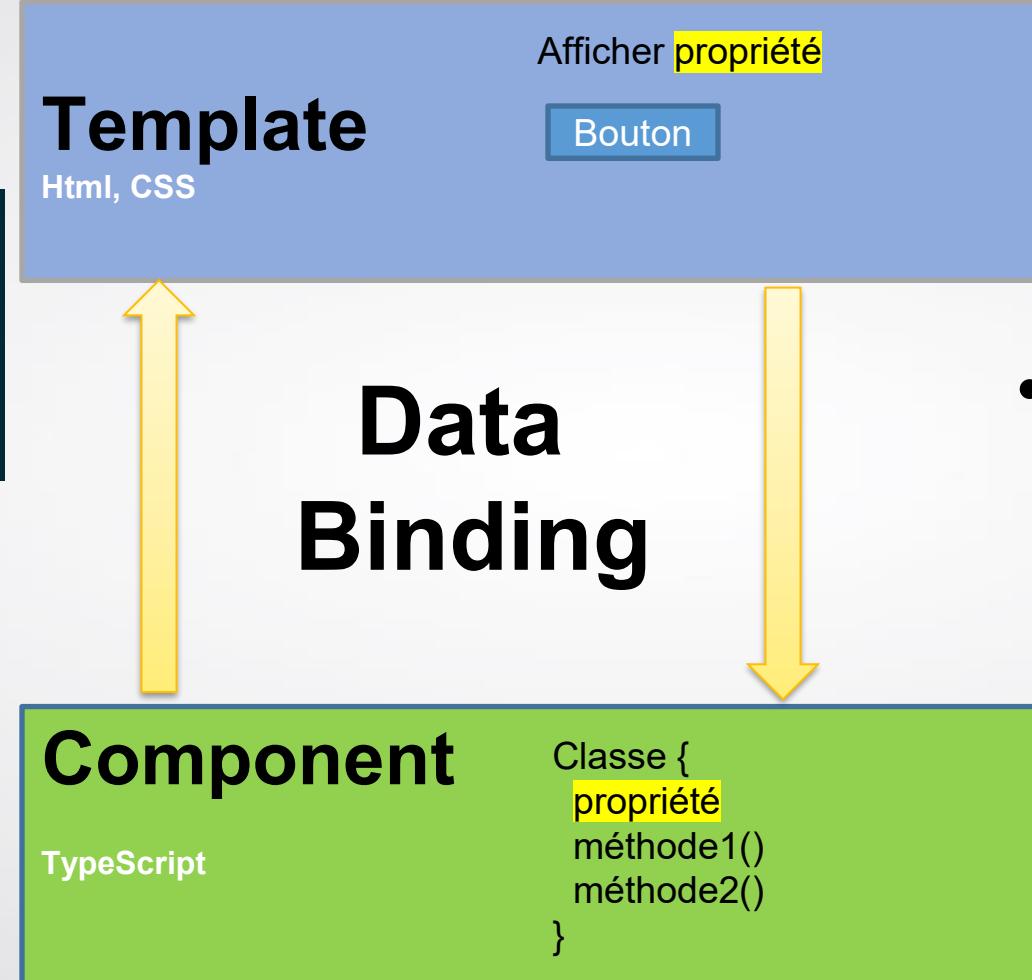


Data-binding : Définition



MetaData

```
Component({  
  selector: 'app-header',  
  templateUrl: './header.component.html',  
  styleUrls: ['./header.component.css']  
})  
  
Component({  
  selector: 'app-header',  
  standalone: true,  
  imports: [],  
  templateUrl: './header.component.html',  
  styleUrls: ['./header.component.css']  
})
```

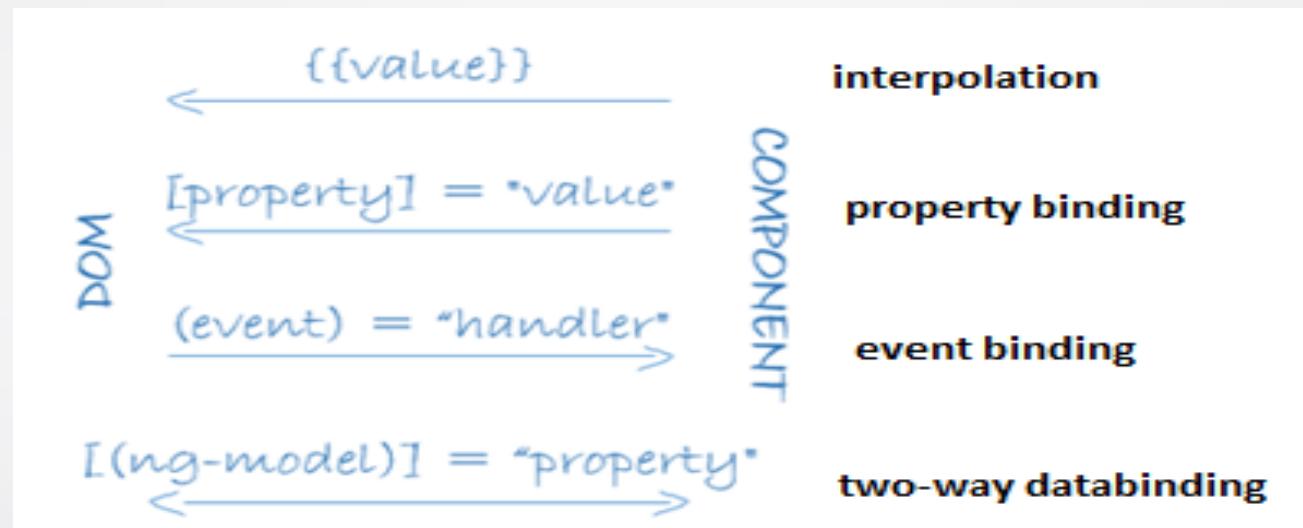


- Le databinding est un mécanisme de coordination entre le composant et son template dans un seul sens ou dans les deux sens.

Data-binding : Formes



- Il existe 4 formes de databinding





Data-binding : Interpolation



- La syntaxe d'interpolation permet d'accéder directement aux propriétés du composant associé. (TS -> HTML)
- L'interpolation permet d'évaluer une expression
`{{ expression_template }}`
- Angular évalue l'expression ensuite convertit le résultat en chaîne de caractères.
- Exemples :
 - ✓ `{{ property1 }}` : property 1 est une propriété du composant
 - ✓ `{{ 1 + 2 }}` : le résultat affiché est 3
 - ✓ `{{ 1 + methode A() }}` : le résultat affiché est la somme de 1 avec le résultat de la méthode getval()

Component A
Export class Component A{
property1="Bonjour";
methodeA(){ return 2 }
}

Template A
<h1> {{property1}} </h1>
<p>{{1+2}}</p>
<p>{{ 1 + methodeA() }}</p>

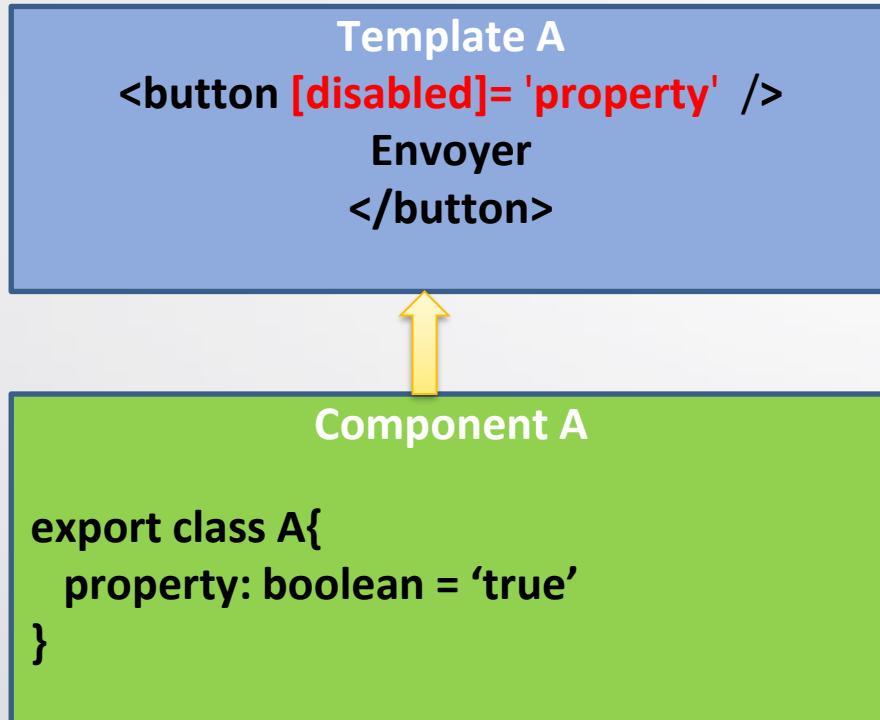
Bonjour
3
3



Data-binding : Property binding



Property binding : permet de modifier la valeur d'une propriété d'un élément du DOM par la valeur d'une propriété du composant.
L'information passe du TS vers HTML

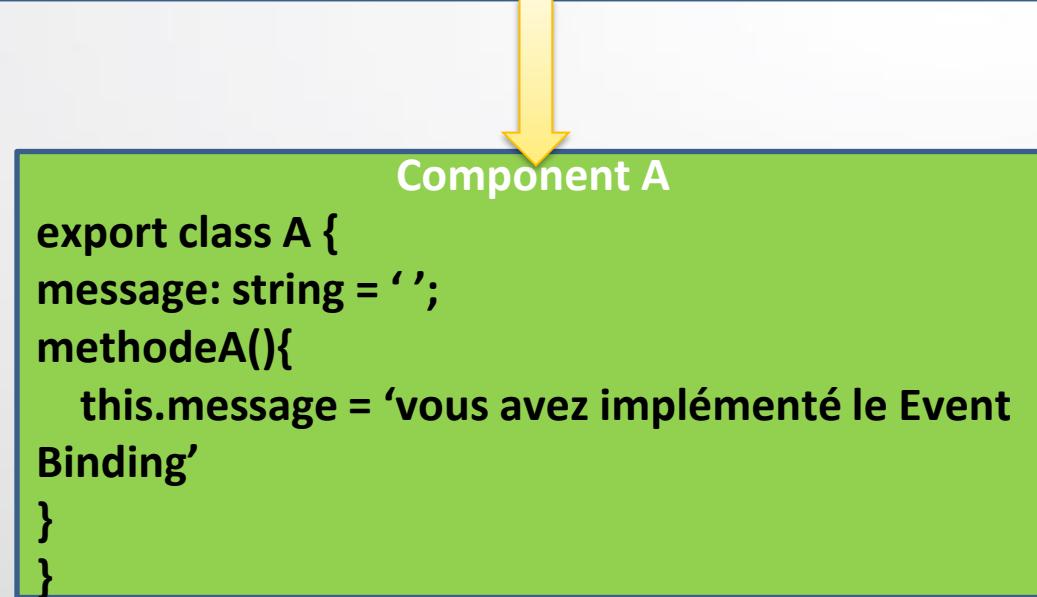
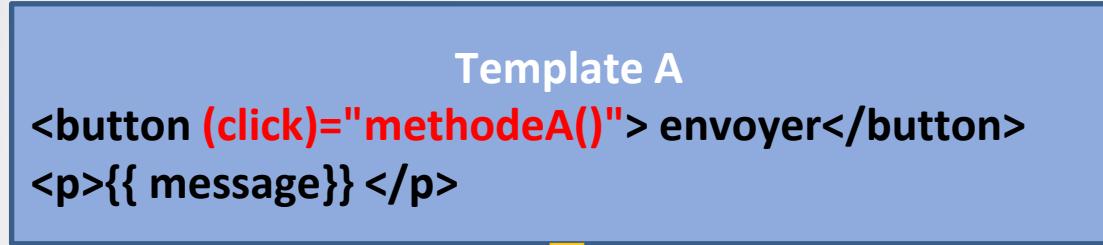




Data – bindig : EventBinding



Event binding : permet d'appeler une méthode du composant suite à une action faite par l'utilisateur au niveau du template.

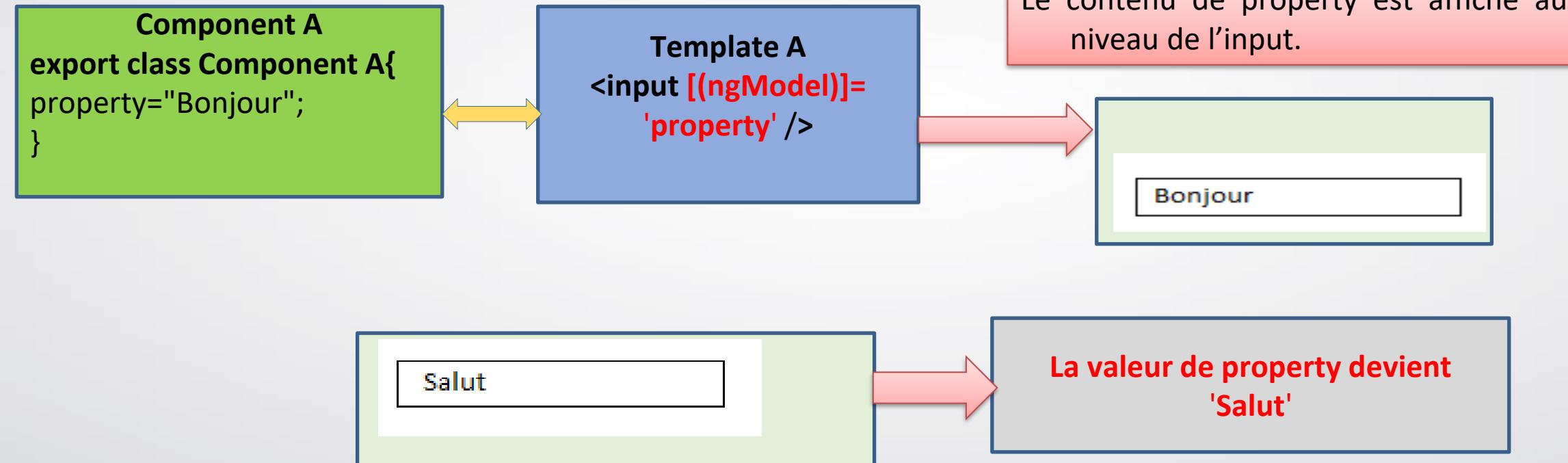




Data – binding : Two-way databinding



Two-way data binding : permet de récupérer une valeur à partir du template et l'envoyer vers une propriété du composant et vice versa. Ceci se fait grâce à la directive NgModel.



FormsModule – importation



La directive **NgModel** est définie dans le module **FormsModule** du package **@angular/forms**.

Pour l'utiliser il faut importer **FormsModule** dans l'emplacement adéquat.

Composant standalone

```
app.component.ts ×

1+ usages  ↗ WiemAouididi *
2  import { RouterOutlet } from '@angular/router';
3  import { HeaderComponent } from './header/header.component';
4  import { FormsModule } from '@angular/forms';

5
6
7  @Component({
8    selector: 'app-root',
9    standalone: true,
10   imports: [RouterOutlet, HeaderComponent, FormsModule],
11   templateUrl: './app.component.html',
12   styleUrls: ['./app.component.css']
13 })
14 export class AppComponent {
15 }
```

Module

```
app.module.ts ×

1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { AppComponent } from './app.component';
4  import { HeaderComponent } from './header/header.component';
5  import { FormsModule } from '@angular/forms';

6
7
8
9  @NgModule({
10   declarations: [
11     AppComponent,
12     HeaderComponent
13   ],
14   imports: [
15     BrowserModule,
16     FormsModule
17   ],
18   providers: [],
19   bootstrap: [AppComponent]
20 })
21 export class AppModule {}
```



Les directives



Les directives - Définition



- **Une directive** est une **fonctionnalité** qui permet de modifier le **comportement ou l'apparence d'un élément du DOM**.
- C'est une instruction que tu ajoutes à une **balise HTML** pour que Angular modifie le DOM ou le comportement de cet élément.
- Il existe trois types de directives :
 1. Directives **structurelles**: Modifie la **structure du DOM**
 2. Directives **d'attributs**: Modifie l'**apparence ou le style** d'un élément existant
 3. Directives **personnalisées**: Crée ton propre comportement réutilisable



Directives : Les directives structurelles



- Une **directive structurelle** modifie la structure du DOM (ajoute, enlève, ou déplace des éléments)

Directives structurelles	Détails
NgIf	Permet d' afficher ou de masquer un élément du DOM en fonction d'une condition booléenne.
NgFor	Permet de générer dynamiquement une liste d'éléments en répétant un bloc HTML pour chaque élément d'une collection (tableau, liste, etc.).
NgSwitch	Permet d' afficher un seul bloc HTML parmi plusieurs possibles, en fonction de la valeur d'une expression .



Directive structurelle : Nglf



- Lorsque Nglf est évalué à false, Angular **supprime l'élément et tous ses descendants du DOM**.
- Il **détruit également les composants associés**, ce qui permet de libérer de la mémoire et des ressources.

Élément	Syntaxe classique (*ngIf)	Nouvelle syntaxe (@if)
Condition simple	<div *ngIf="cond">...</div>	@if (cond) { ... }
Condition avec else	*ngIf="cond; else tpl" + <ng-template>	@if (...) { ... } @else { ... }
Exemple	<!-- Affiche si l'utilisateur est connecté --> <div *ngIf="isLoggedIn; else notConnected"> Bonjour, vous êtes connecté. </div> <ng-template #notConnected> <p>Veuillez vous connecter.</p> </ng-template>	@if (isLoggedIn) { <div>Bonjour, vous êtes connecté.</div> } @else { <p>Veuillez vous connecter.</p> }

Directive structurelle : NgIf

Exemple dans un composant standalone



1. Importer NgIf dans le composant

```
A app.component.ts ×

1  import { Component } from '@angular/core';
2  import { RouterOutlet } from '@angular/router';
3  import { HeaderComponent } from './header/header.component';
4  import {NgIf} from '@angular/common';

5

1+ usages  ↗ WiemAouididi *
6
7  ↗ @Component({
8    ↗ selector: 'app-root',
9    ↗ standalone: true,
10   ↗ imports: [RouterOutlet, HeaderComponent, NgIf],
11   ↗ templateUrl: './app.component.html',
12   ↗ styleUrls: ['./app.component.css']
13 })
14 export class AppComponent {
15   Condition : boolean = true;
}
```

2. Utilisation

```
<div *ngIf="Condition">
  <h3>Contenu à afficher</h3>
</div>
```

Directive structurelle : NgIf .. Else

Exemple dans un composant standalone



1. Importer NgIf dans le composant

```
app.component.ts x

1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3 import { HeaderComponent } from './header/header.component';
4 import { NgIf } from '@angular/common';
5
6 1+ usages  ↗ WiemAouididi *
7
8 @Component({
9   selector: 'app-root',
10  standalone: true,
11  imports: [RouterOutlet, HeaderComponent, NgIf],
12  templateUrl: './app.component.html',
13  styleUrls: ['./app.component.css']
14})
15 export class AppComponent {
  Condition : boolean = true;
}
```

2. Utilisation

```
<div *ngIf = "Condition ; else other_content">
  contenu ici
</div>
new *
<ng-template #other_content >
  autre contenu ici ...
</ng-template>
```



Directive structurelle : NgFor



- Permet de répéter un élément plusieurs fois dans le DOM. Elle prend en paramètre les entités à reproduire.
- Fournit certaines valeurs :
 - **index** : position de l'élément courant
 - **first** : vrai si premier élément
 - **last** : vrai si dernier élément
 - **even** : vrai si l'indice est pair
 - **odd** : vrai si l'indice est impair



Directive structurelle : NgFor



Directive	Syntaxe classique	Nouvelle syntaxe Angular 18
ngFor	*ngFor="let x of list"	@for (x of list) { ... }
Exemple	<pre><li *ngFor="let item of items; index as i; last as isLast"> {{ i }} - {{ item }} *ngIf="isLast">(dernier) </pre>	<pre>@for (item of items; let i = \$index; let isLast = \$last) { {{ i }} - {{ item }} @if (isLast) { (dernier) } }</pre>



Directive structurelle : NgFor

Exemple dans un composant standalone



1. Importer NgFor dans le composant

```
A app.component.ts ×

1  import { Component } from '@angular/core';
2  import { RouterOutlet } from '@angular/router';
3  import { HeaderComponent} from './header/header.component';
4  import {NgFor} from '@angular/common';

5
6  1+ usages  ↗ WiemAouididi *
7  @Component({
8    selector: 'app-root',
9    standalone: true,
10   imports: [RouterOutlet, HeaderComponent, NgFor],
11   templateUrl: './app.component.html',
12   styleUrls: ['./app.component.css'
13 })
14 export class AppComponent {
15   items :(...{})[] = [
16     { name: 'Article 1' },
17     { name: 'Article 2' },
18     { name: 'Article 3' }
19   ];
}
```

2. Utilisation

```
5 app.component.html ×

1 <div *ngFor="let item of items">
2   {{ item.name }}
3 </div>
```



Directive structurelle : NgSwitch



- Comme l'instruction switch en JavaScript, NgSwitch affiche un seul élément parmi plusieurs éléments possibles, en fonction d'une condition.
- Angular insère uniquement **l'élément sélectionné** dans le DOM.
- NgSwitch est un ensemble de **trois directives** :
 1. **NgSwitch** : directive principale qui évalue l'expression (la condition de sélection).
 2. **NgSwitchCase** : directive utilisée pour chaque **valeur possible** à comparer avec ngSwitch.
 3. **NgSwitchDefault** : directive utilisée lorsque **aucune valeur** ne correspond.



Directive structurelle : NgSwitch



Élément	Syntaxe classique	Nouvelle syntaxe Angular 18
Déclaration	[ngSwitch]="val"	@switch (val)
Cas	*ngSwitchCase="val"	@case ('val') { ... }
Cas par défaut	*ngSwitchDefault	@default { ... }
Exemple	<pre><div [ngSwitch]="color"> <p *ngSwitchCase="red">Rouge</p> <p *ngSwitchCase="blue">Bleu</p> <p *ngSwitchCase="green">Vert</p> <p *ngSwitchDefault>Couleur inconnue</p> </div></pre>	<pre>@switch (color) { @case ('red') { <p>Rouge</p> } @case ('blue') { <p>Bleu</p> } @case ('green') { <p>Vert</p> } @default { <p>Couleur inconnue</p> } }</pre>

Directive structurelle : NgSwitch

Exemple dans un composant standalone



1. Importer NgSwitch dans le composant

```
app.component.ts x

1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3 import { HeaderComponent } from './header/header.component';
4 import { NgSwitch, NgSwitchCase, NgSwitchDefault } from '@angular/common';
5
6 1+ usages  ↗ WiemAouididi *
7
8 @Component({
9   selector: 'app-root',
10  standalone: true,
11  imports: [RouterOutlet, HeaderComponent, NgSwitch, NgSwitchCase, NgSwitchDefault],
12  templateUrl: './app.component.html',
13  styleUrls: ['./app.component.css']
14})
15 export class AppComponent {
16   role : string = 'admin';
17 }
```

2. Utilisation

```
app.component.html x

1 <div [ngSwitch]="role">
2   <p *ngSwitchCase="'admin'">Bienvenue, administrateur !</p>
3   <p *ngSwitchCase="'user'">Bienvenue, utilisateur.</p>
4   <p *ngSwitchDefault>Accès invité</p>
5 </div>
6
```



Les directives – Directives attributs



- Les Directives d'**attribut** changent l'apparence et l'attitude d'un élément DOM, composant ou autre directive tels que : NgStyle, NgClass, NgModel, etc

```
<element [ngClass]="string|array|object">  
</element>
```

```
<p [ngClass]=""first second"">... </p>  
<p [ngClass]=["first", 'second']>... </p>  
<p [ngClass]="{{'first': true, 'second': true}}>... </p>
```

```
<element [ngStyle]="objectExpression">  
</element>
```

```
<p [ngStyle]="{{'font-style': 'italic'}}>... </p>
```



Les directives – Directives personnalisées



Les directives personnalisées sont créées par les développeurs pour répondre aux besoins spécifiques de leur application.

Les directives personnalisées peuvent être utilisées pour :

- Ajouter des fonctionnalités spécifiques à des éléments HTML, comme par exemple, afficher une infobulle lorsqu'un élément est cliqué.
- Modifier le style d'un élément HTML, comme par exemple, changer la couleur du texte ou ajouter une bordure.
- Ajouter des événements à des éléments HTML, comme par exemple, un événement de clic.
- Gérer des données, comme par exemple, afficher des données en temps réel.



Les directives – Directives personnalisées



Pour créer une directive personnalisée, il faut passer par les étapes suivantes:

1. **Définir la directive** : Définissez la directive en utilisant le décorateur @Directive et en spécifiant le sélecteur CSS qui sera utilisé pour sélectionner les éléments HTML à qui la directive sera appliquée. Vous pouvez utiliser la commande CLI :
`ng generate directive Nom_de_la_directive`
2. **Définir la logique** : Définissez la logique de la directive en utilisant des méthodes comme ngAfterViewInit ou ngAfterContentInit pour interagir avec les éléments HTML.
3. **Définir le module** : Définissez le module Angular dans lequel vous souhaitez créer la directive. La directive sera ajoutée dans la section "declarations"
4. **Utiliser la directive** : Utilisez la directive dans votre application en la plaçant sur un élément HTML.



Les directives – Directives personnalisées



Exemple:

//Définir la directive et la logique de la directive

```
import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) { }

  ngAfterViewInit() {
    this.renderer.addClass(this.el.nativeElement, 'highlight');
  }
}
```

```
//styles.css
.highlight {
  background-color: yellow;
  padding: 10px;
  border: 1px solid black;
}
```

<!----utiliser la directive --- >

<p appHighlight> Appliquer la directive HilightDirective sur cet élément</p>

//déclarer la directive dans le module correspondant

```
.....
@NgModule({
  declarations: [
    AppComponent,
    HighlightDirective
  ],
  .....
```

Résultat :

Appliquer la directive HilightDirective sur cet élément

...



Les pipes



Les pipes



- Un **pipe** est un **outil de transformation de données** dans les templates Angular.
- Il permet d'afficher une donnée dans un format différent, sans modifier sa valeur réelle dans le code TypeScript.
- Les pipes permettent de définir une fonction de transformation une seule fois, puis de la réutiliser dans plusieurs templates.
- En Angular, il existe trois types de pipes : les **pipes intégrés** (built-in), les **pipes personnalisés** (custom), et les **pipes purs ou impurs** selon leur comportement de mise à jour.



Pipes intégré - exemples



| Pipe | Description | Exemple | Résultat attendu |
|------------------|---------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|-----------------------|
| uppercase | Convertit tout le texte en majuscules . | <code>{{ 'bonjour' uppercase }}</code> | BONJOUR |
| lowercase | Convertit tout le texte en minuscules . | <code>{{ 'ANGULAR' lowercase }}</code> | angular |
| titlecase | Met en majuscules la première lettre de chaque mot. | <code>{{ 'bonjour tout le monde' titlecase }}</code> | Bonjour Tout Le Monde |
| date | Formate une date selon un format spécifique (short, longDate, etc.) | <code>{{ today date:'longDate' }}</code> | 29 juillet 2025 |
| number | Formate les nombres décimaux selon une configuration (1.2-2 = min 1 chiffre entier, 2 chiffres après la virgule). | <code>{{ 3.14159 number:'1.2-2' }}</code> | 3,14 |

Pipes intégré - exemples



| Pipe | Description | Exemple | Résultat attendu |
|-----------------|-------------------------------------------------------------------------------------------------|------------------------------------------|------------------------------|
| currency | Affiche un montant monétaire formaté selon la devise (USD, EUR, etc.) et la locale. | <code>{{ 1200 currency:'EUR' }}</code> | 1 200,00 € |
| percent | Affiche une valeur sous forme de pourcentage (0.75 → 75%). | <code>{{ 0.875 percent }}</code> | 87.5% |
| slice | Coupe une chaîne de caractères ou un tableau (slice:1:3 → éléments 1 à 2). | <code>{{ 'Angular' slice:0:3 }}</code> | Ang |
| json | Transforme un objet JavaScript en texte JSON lisible (utile pour le debug). | <code>{{ user json }}</code> | { "name": "Ali", "age": 25 } |
| async | Affiche automatiquement la valeur d'un Observable ou Promise dès qu'elle est disponible. | <code>{{ userData\$ async }}</code> | Résultat de l'observable |

Les pipes : Utilisation des pipes intégrés

Exemple dans un composant standalone



1. Importer le pipe dans le composant

```
A app.component.ts ×

1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3 import { HeaderComponent } from './header/header.component';
4 import { DatePipe } from '@angular/common';
5
6
7 1+ usages   WiemAouididi *
8
9 @Component({
10   selector: 'app-root',
11   standalone: true,
12   imports: [RouterOutlet, HeaderComponent, DatePipe],
13   templateUrl: './app.component.html',
14   styleUrls: ['./app.component.css']
15 })
16 export class AppComponent {
17   currentDate : string = '2024-07-08';
18 }
```

2. Utilisation

```
B app.component.html ×

1 <div>
2   <h1>Current date is {{ currentDate | date }}</h1>
3 </div>
4
```



Les pipes intégrés - exemples



```
<p>La date est : {{ myDate | date:'yyyy-MM-dd' }}</p>
```

myDate= 20/06/2024

La date est: "2024-06-20"

```
<p>En majuscules : {{ myString | uppercase }}</p>
```

myString= "hello"

En majuscules : HELLO

Vous pouvez enchaîner les pipes pour appliquer plusieurs transformations à une valeur

```
<div>{{ user.birthDate | date | uppercase }}</div>
```



Les pipes personnalisés



Les pipes personnalisés sont créés selon votre choix pour répondre à des besoins spécifiques.

Pour créer un pipe personnalisé, vous devez:

1. Créer une classe qui implémente l'interface PipeTransform et la décorer avec le décorateur @Pipe. Cette étape peut être réalisée automatiquement via la commande ng generate pipe Nom_Pipe
2. Implémenter la méthode transform qui prend la donnée en entrée et la retourne transformée.
3. Importer et déclarer le pipe dans le module où vous voulez l'utiliser (par exemple, dans app.module.ts) :
4. Utiliser le pipe dans le code HTML

► Les pipes personnalisés



Exemple:

```
//créer la classe et implémenter la méthode transform
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'euro'
})
export class EuroPipe implements PipeTransform {

  transform(value: number, ...args: unknown[]): string {
    return '€' + value.toFixed(2);
  }
}
```

<!--utiliser le pipe dans le HTML -->
<p>Le prix est : {{ 12.34 | euro }}</p>

//déclarer le pipe dans le module correspondant
.....
@NgModule({
 declarations: [
 AppComponent,
 EuroPipe
],

Résultat :

```
<p>Le prix est : {{ 12.34 | euro }}</p>
```

Cycle de vie d'un composant

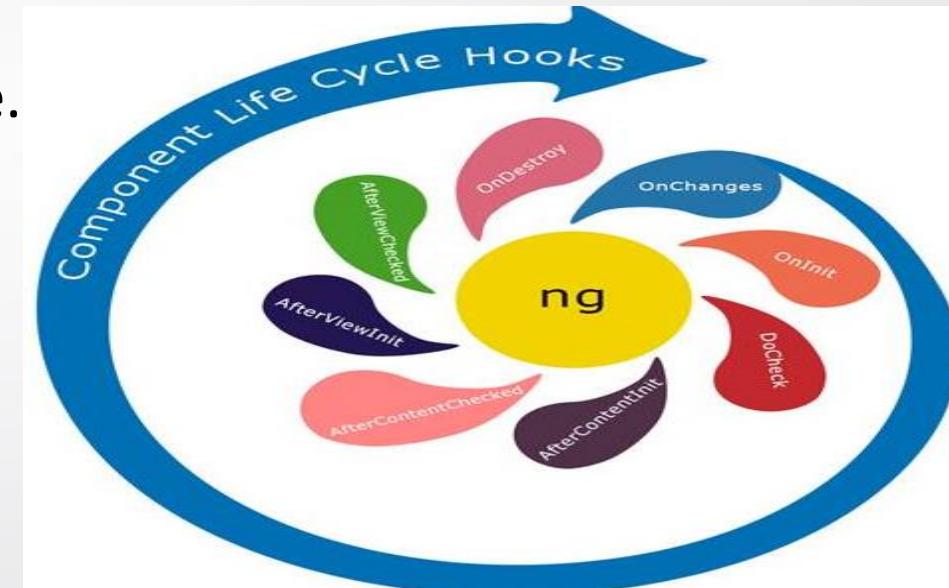




Cycle de vie d'un composant



- Un composant passe par plusieurs phases depuis sa création jusqu'à sa destruction : cycle de vie
- Angular maintient et suit ces différentes phases en utilisant des méthodes appelées « hooks ».
- A chaque phase on peut implémenter une logique.
- Ces méthodes se trouvent dans des interfaces dans la librairie « @angular/core »





Cycle de vie d'un composant



- Le constructeur d'un composant n'est pas un hook mais il fait partie du cycle de vie d'un composant : sa création
- Il est logiquement appelé en premier, et c'est à ce moment que les dépendances (services) sont injectées dans le composant par Angular.



Cycle de vie d'un composant



Méthode/hook	
ngOnChanges	Appelé lorsqu'une propriété input est définie ou modifiée de l'extérieur. L'état des modifications sur ces propriétés est fourni en paramètre
ngOnInit	Appelé une seule fois après le 1 ^{er} appel du hook ngOnChanges(). Permet de réaliser l'initialisation du composant, qu'elle soit lourde ou asynchrone (on ne touche pas au constructeur pour ça)
ngDoCheck	Appelé après chaque détection de changements
ngAfterContentInit	Appelé une fois que le contenu externe est projeté dans le composant (transclusion)



Cycle de vie d'un composant



Méthode	Rôle
ngAfterContentChecked	Appelé chaque fois qu'une vérification du contenu externe (transclusion) est faite
ngAfterViewInit	Appelé dès lors que la vue du composant ainsi que celle de ses enfants sont initialisés
ngAfterViewChecked	Appelé après chaque vérification des vues du composant et des vues des composants enfants.
ngOnDestroy	Appelé juste avant que le composant soit détruit par Angular. Il permet alors de réaliser le nettoyage adéquat de son composant. C'est ici qu'on veut se désabonner des Observables ainsi que des events handlers sur lesquels le composant s'est abonné.



Cycle de vie d'un composant



Les méthodes **ngAfterContentInit**, **ngAfterContentChecked**, **ngAfterViewInit** et **ngAfterViewChecked** sont exclusives aux composants, tandis que toutes les autres le sont aussi pour les directives.



Références



- <https://angular.io/>



► Merci de votre attention