esprit
Se former autrement
ONORIS UNITED UNIVERSITIES

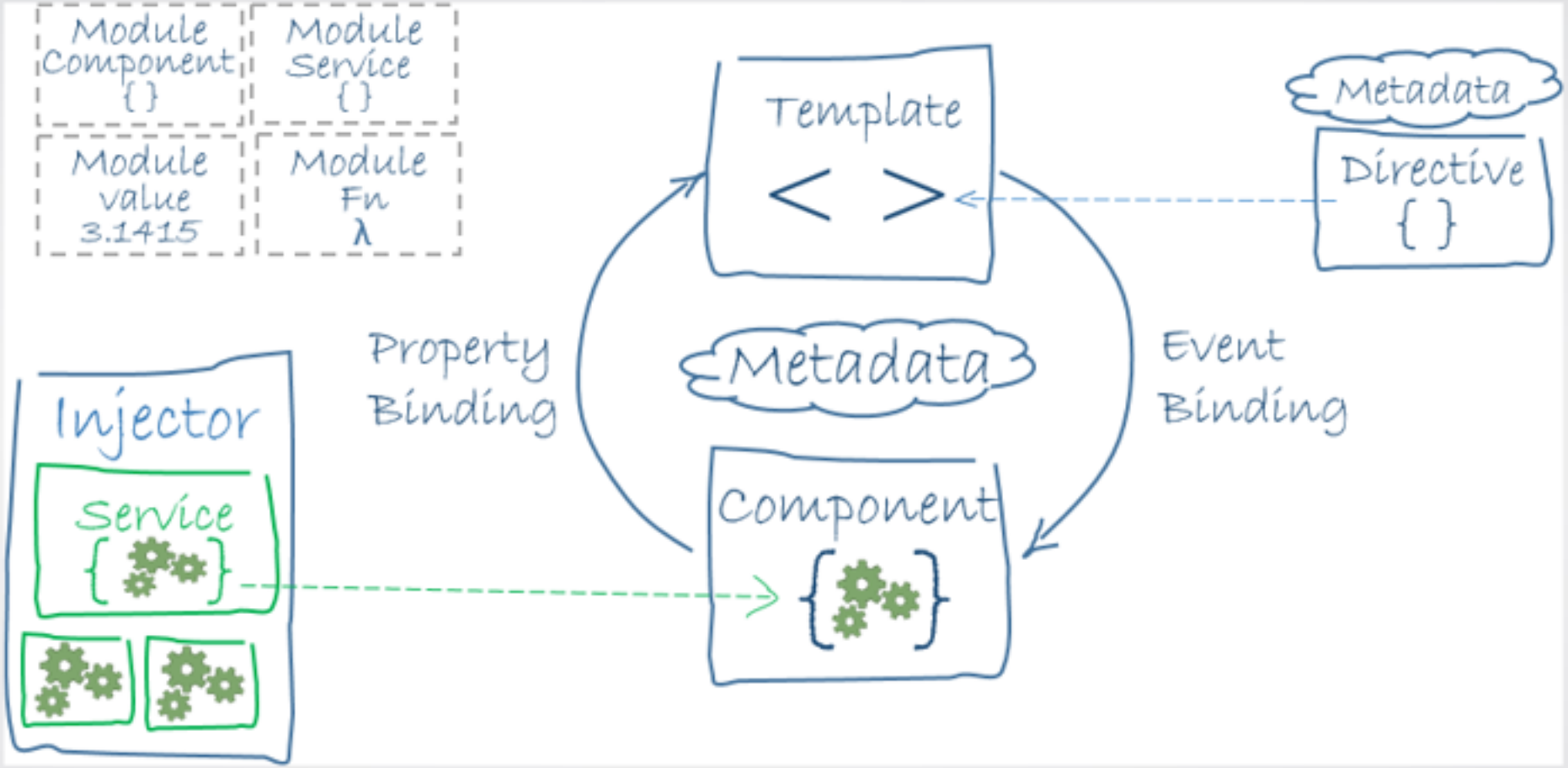
Manipulation de composants dans

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
CONFERENCE DES GRANDES ÉCOLES
cdio
CONCEIVE DESIGN IMPLEMENT OPERATE

- 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

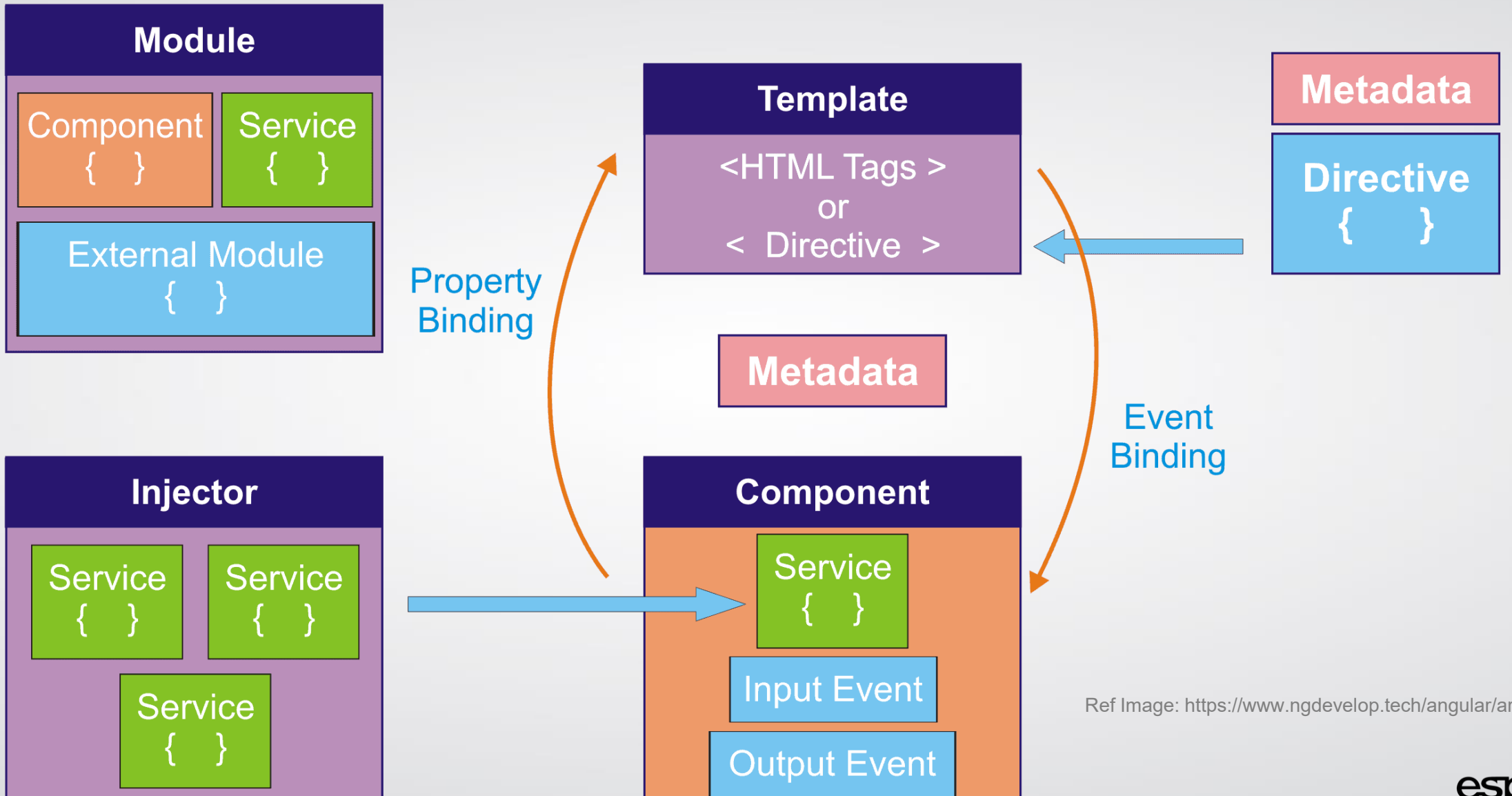
Architecture d’un projet Angular classique (non-standalone)

Architecture Angular



Ref image: https://angular.io/guide/architecture

Architecture Angular

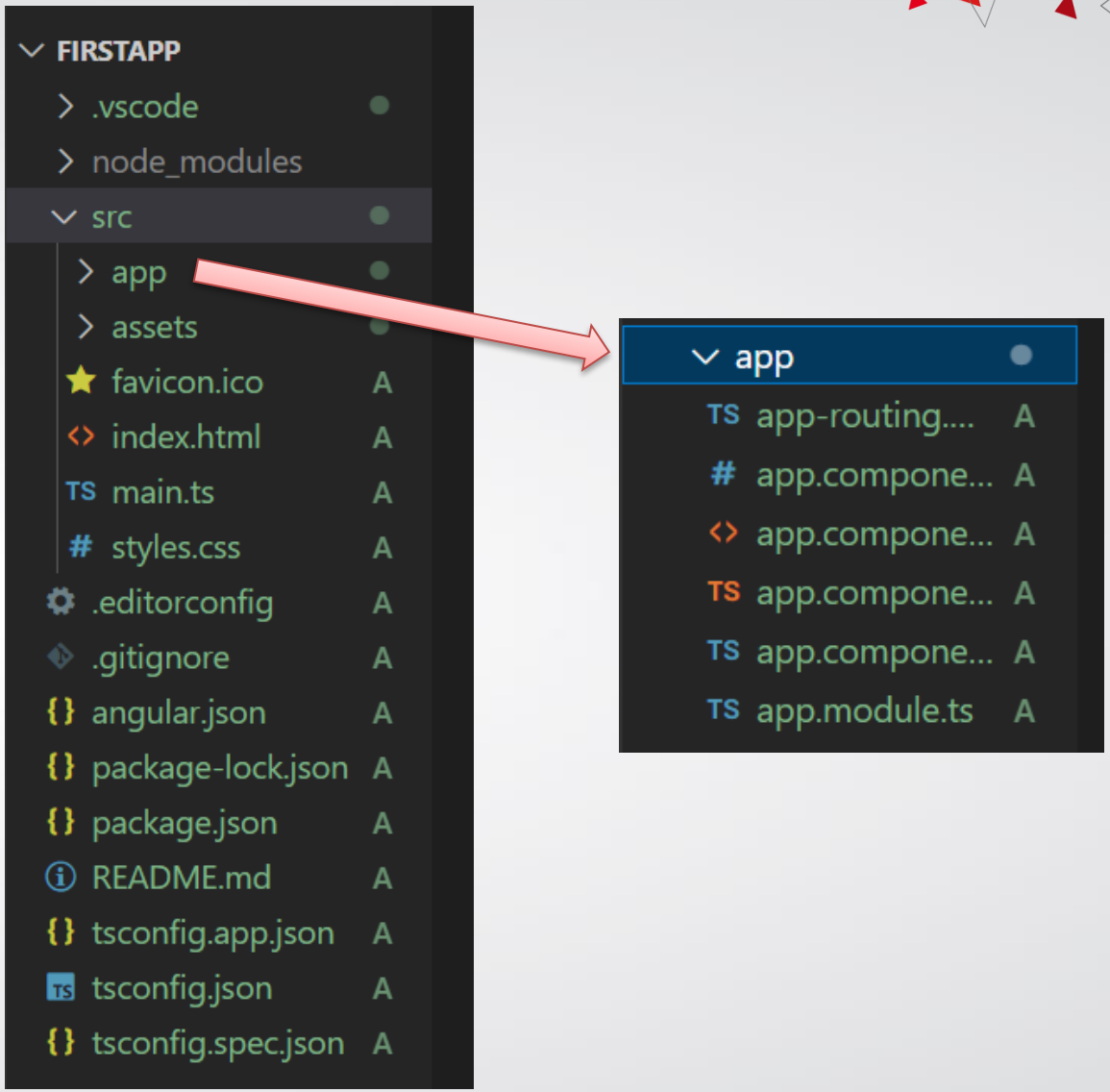


Ref Image: https://www.ngdevelop.tech/angular/architecture/

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** »



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 > ...

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

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

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

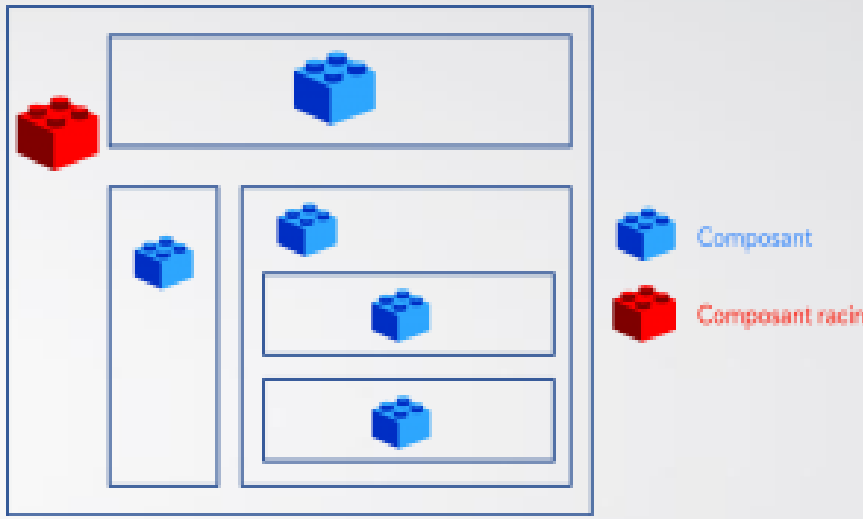
- Un composant web est un élément personnalisé et réutilisable.

- 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 le définit comme étant un composant et spécifie sa métadata 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

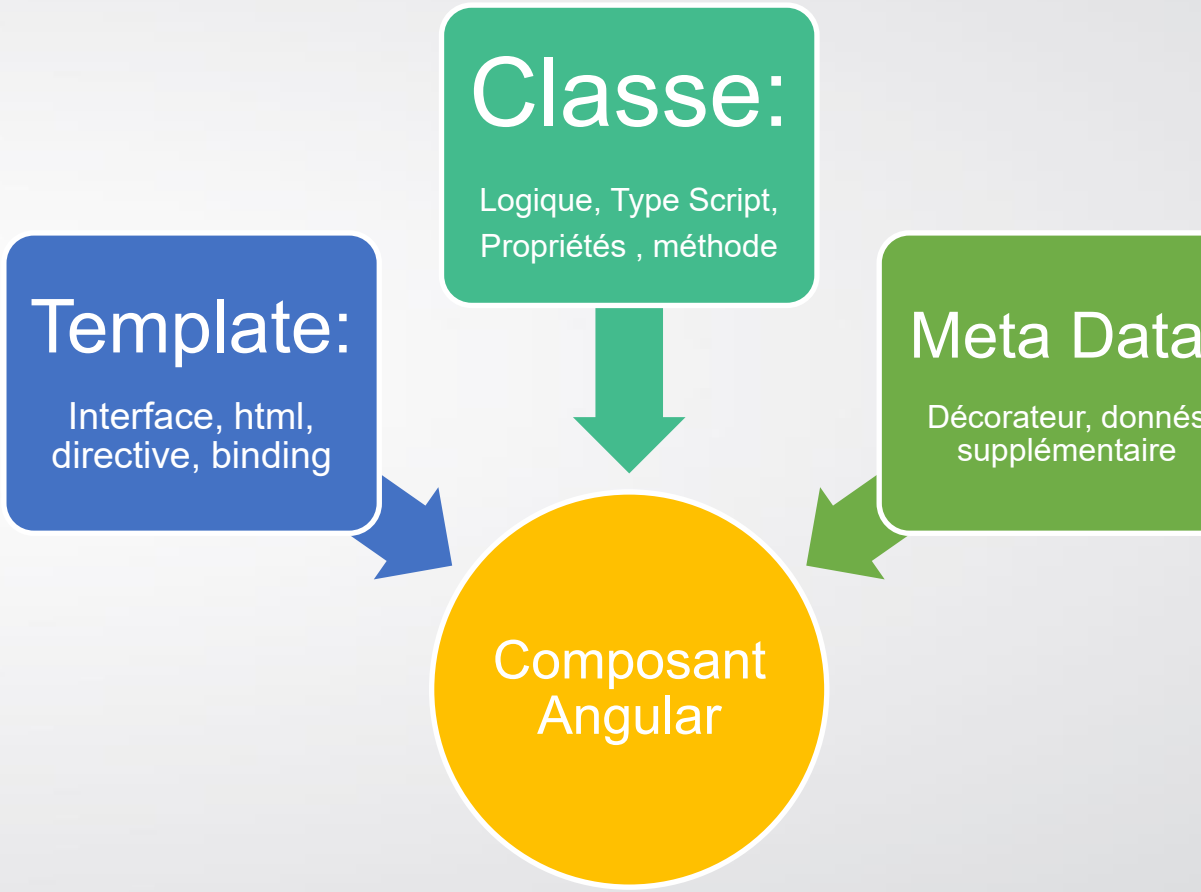
Un composant Angular contient:

.

- 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

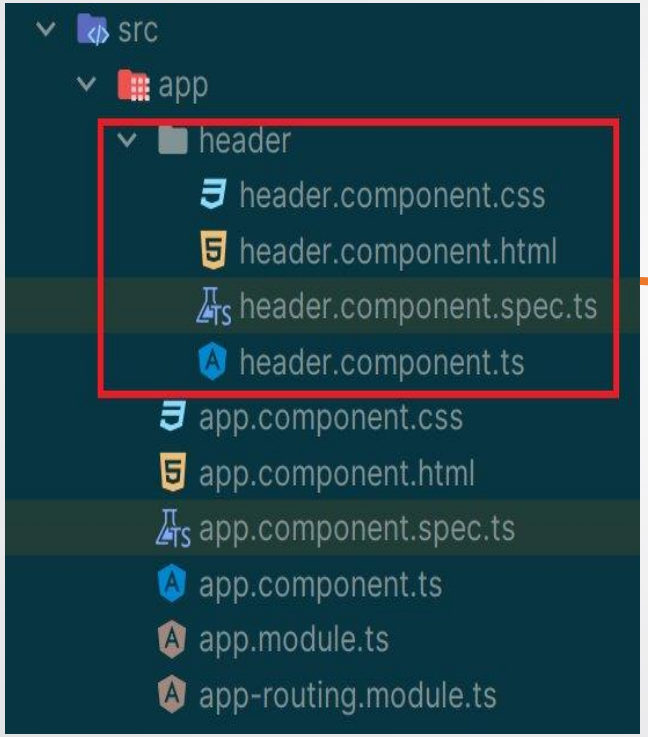
- Générer un nouveau composant dans votre Application angular:

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:




```ts
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
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Composant non-standalone : Déclaration

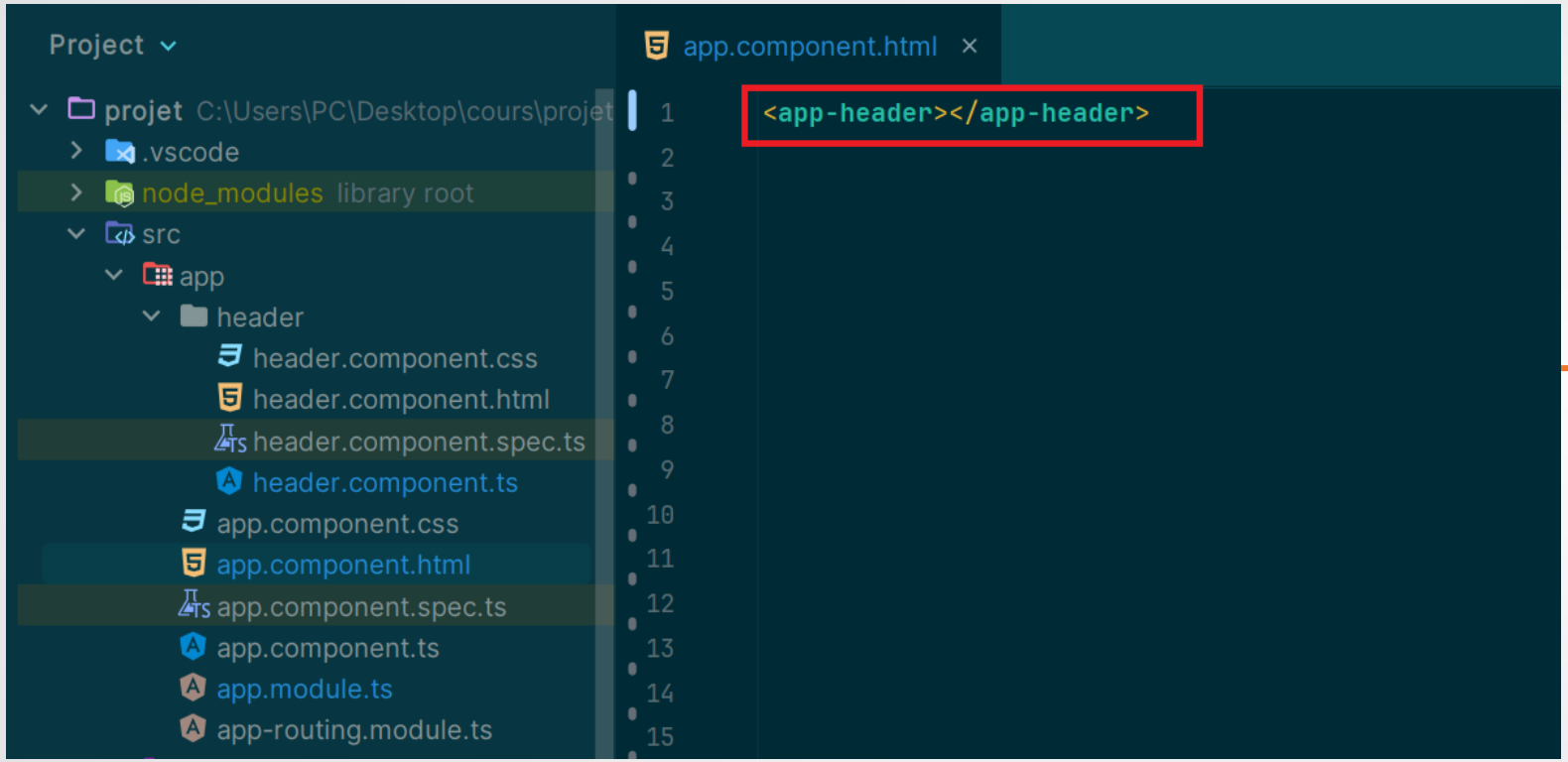
Le décorateur @Component (angular/core) contient par défaut (génération par défaut de CLI) 3 tags:

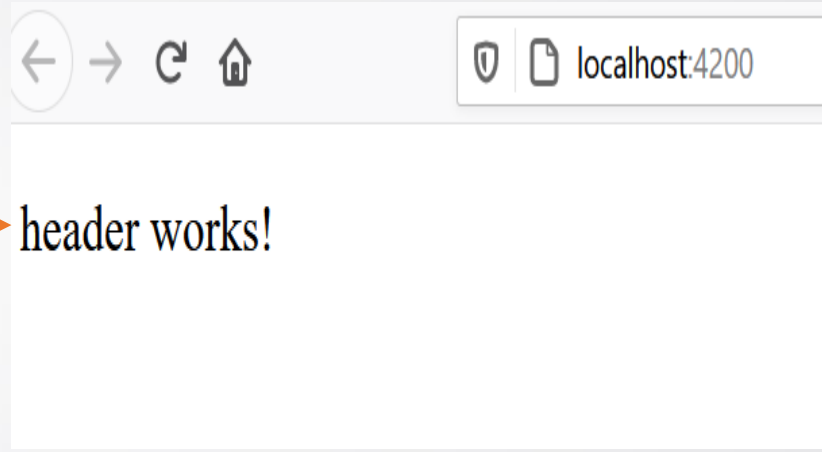
- **« selector » :** Le sélecteur indique la déclaration qui permet d'insérer le composant dans le document HTML.
- **« templateUrl »**: Permet d'associer un fichier externe HTML contenant la structure de la vue du composant
- **« styleUrls »**: Spécifie les feuilles de styles CSS associées à ce composant.

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

1+ usages   WiemAouididi
@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:





Composant standalone : Création

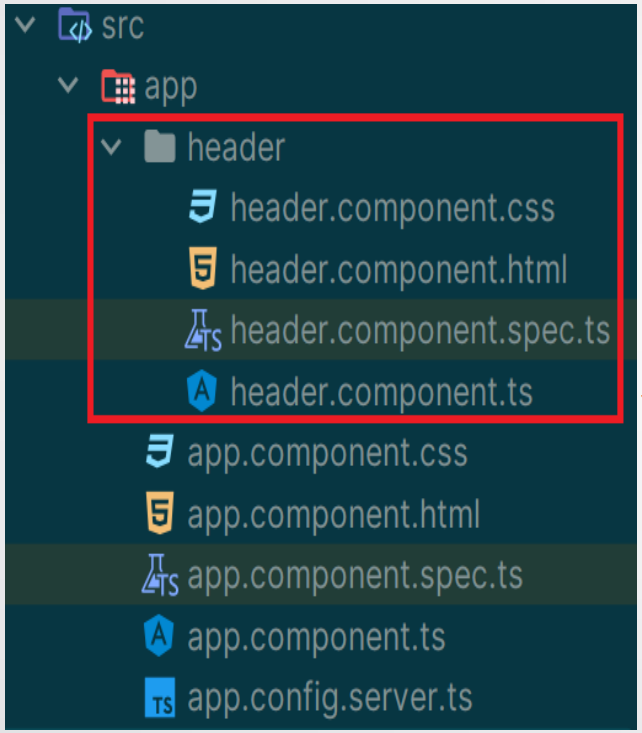
- Générer un nouveau composant dans votre Application angular:

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:




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

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

}
```

Composant standalone : Affichage

- Angular permet d'utiliser des composants standalones 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**

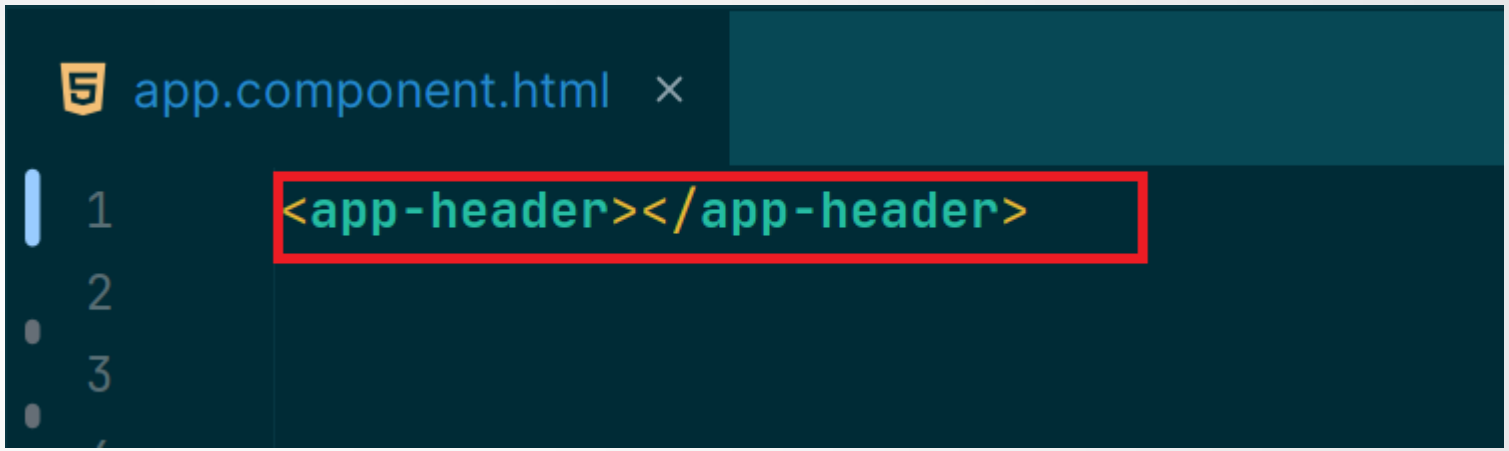
app.component.ts

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

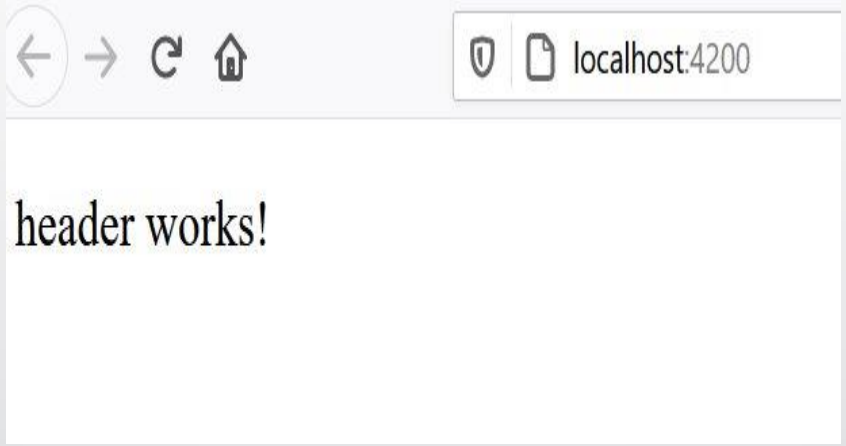
Composant standalone : Affichage

- Le sélecteur permet de faire appel à votre composant:

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



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

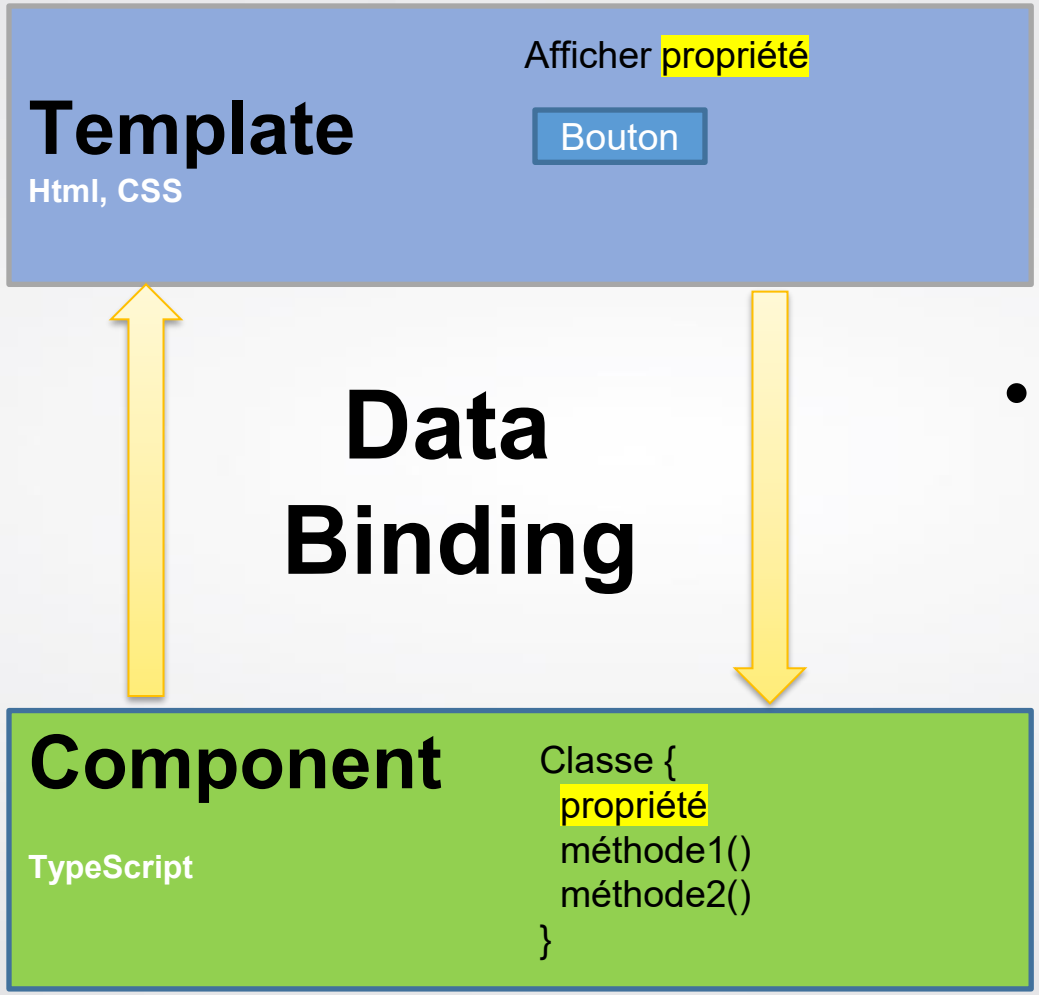
Data-Binding

Data-binding : Définition

MetaData

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

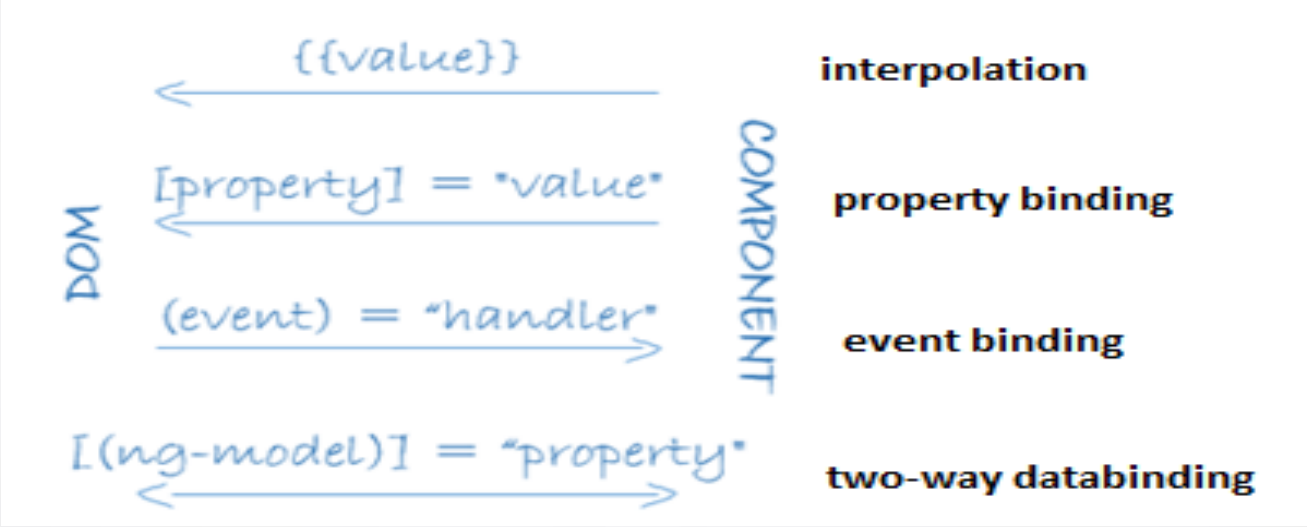
```
Component({
 selector: 'app-header',
 standalone: true,
 imports: [],
 templateUrl: './header.component.html',
 styleUrl: './header.component.css'
)
```



- 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

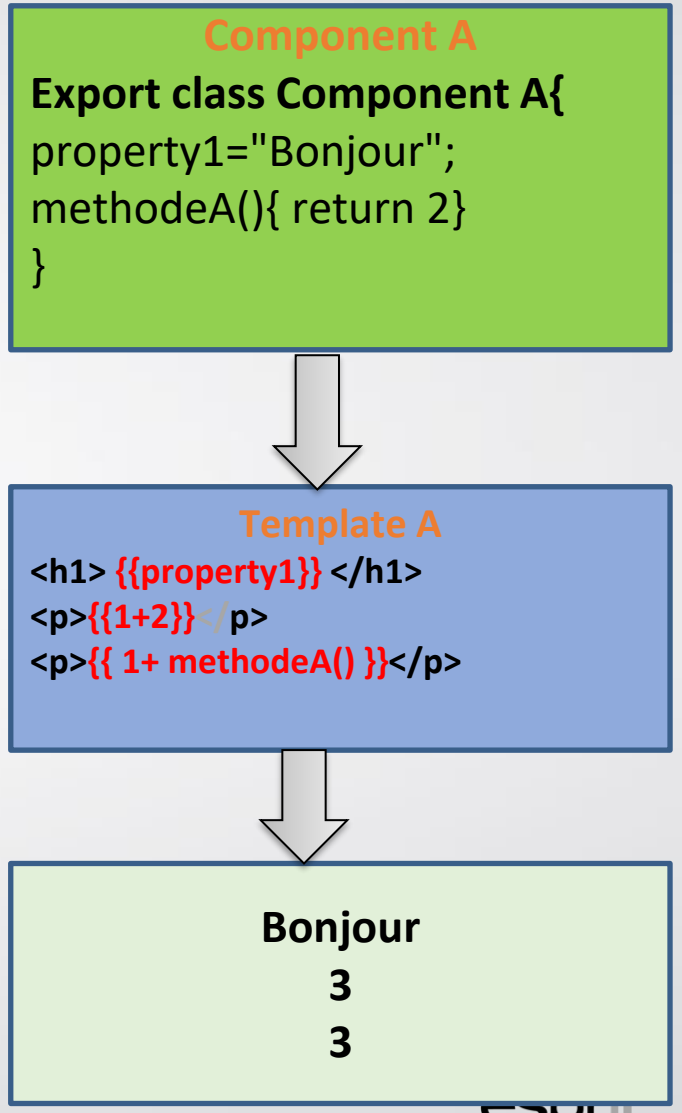


Data-binding : Interpolation

- 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 chaine 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 e résultat de la méthode getval()



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

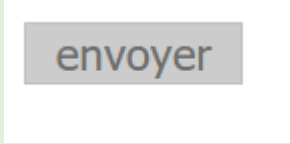
Template A

```
<button [disabled]= 'property'  />
Envoyer
</button>
```

Component A

```
export class A{
   property: boolean = 'true'
}
```

Rendu Final



Data – bindig : EventBinding

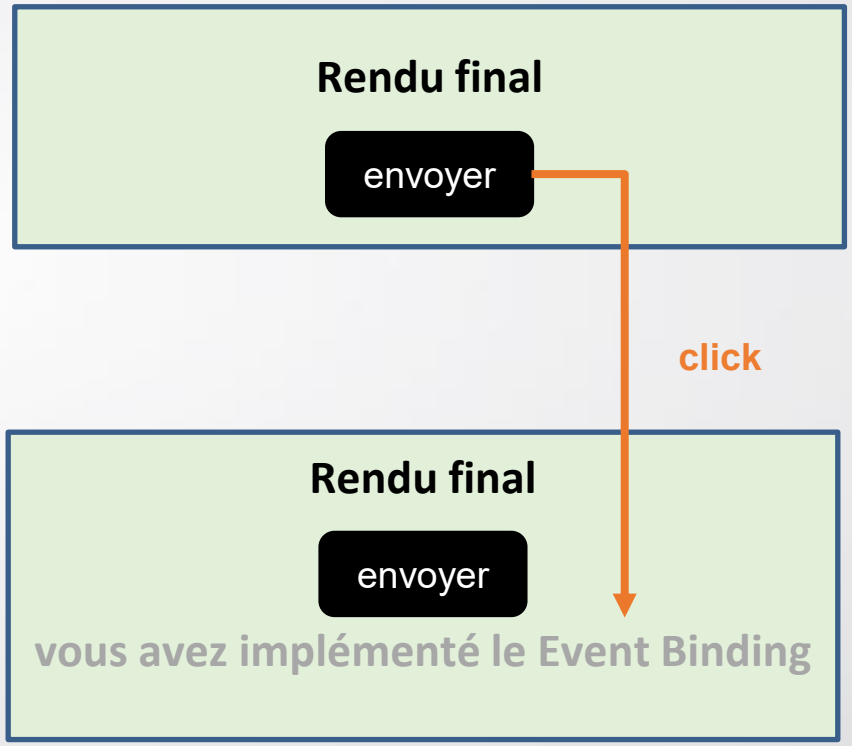
Event binding : permet d'appeler une méthode du composant suite à une action faite par l'utilisateur au niveau du template.

Template A

```
<button (click)="methodeA()"> envoyer</button>
<p>{{ message}} </p>
```

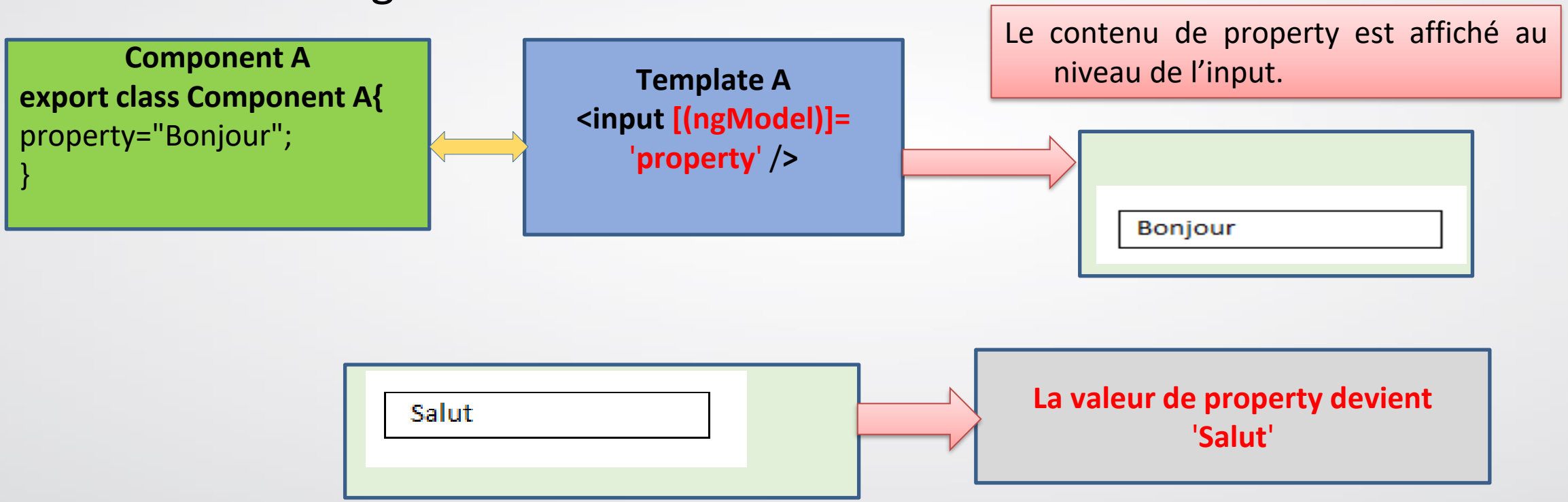
Component A

```
export class A {
message: string = ' ';
methodeA(){
   this.message = 'vous avez implémenté le Event Binding'
}
}
```



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 définit dans le module **FormsModule** du package **@angular/forms.**

Pour l'utiliser il faut importer FormsModule dans l'emlacement adéquat.

Composant standalone

app.component.ts

```
import { RouterOutlet } from '@angular/router';
import {HeaderComponent} from './header/header.component';
import {FormsModule} from '@angular/forms';


1+ usages  WiemAouididi *
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, HeaderComponent, FormsModule],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
}
```

Module

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import {FormsModule} from '@angular/forms';


@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
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 : NgIf

- Lorsque NgIf 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">...	@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.
	
	<ng-template #notConnected>
	  Veuillez vous connecter.
	</ng-template>
		@if (isLoggedIn) {
		  Bonjour, vous êtes connecté.
		} @else {
		  Veuillez vous connecter.
		}

Directive structurelle : NgIf

Exemple dans un composant standalone

1. Importer NgIf dans le composant

app.component.ts

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import {HeaderComponent} from './header/header.component';
import {NgIf} from '@angular/common';

1+ usages  WiemAouididi *
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, HeaderComponent, NgIf],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
Condition : boolean  = true;
}
```

2. Utilisation

```html
<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

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import {HeaderComponent} from './header/header.component';
import {NgIf} from '@angular/common';

1+ usages  WiemAouididi *
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, HeaderComponent, NgIf],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
Condition : boolean  = true;
}
```

2. Utilisation

```html
<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	<li *ngFor="let item of items; index as i; last as isLast">
	{{ i }} - {{ item }} <span *ngIf="isLast">(dernier)
	
		@for (item of items; let i = $index; let isLast = $last) {
		
		{{ i }} - {{ item }}
		@if (isLast) {
		(dernier)
		}
		
		}

Directive structurelle : NgFor

Exemple dans un composant standalone

1. Importer NgFor dans le composant

app.component.ts

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import {HeaderComponent} from './header/header.component';
import {NgFor} from '@angular/common';

1+ usages  WiemAouididi *
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, HeaderComponent, NgFor],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  items : ({...})[]  = [
    { name: 'Article 1' },
    { name: 'Article 2' },
    { name: 'Article 3' }
  ];
}
```

2. Utilisation

app.component.html

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

Directive structurelle : NgSwich

- 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 : NgSwich

Élément	Syntaxe classique	Nouvelle syntaxe Angular 18
Déclaration	[ngSwitch]="val"	@switch (val)
Cas	*ngSwitchCase="'val'"	@case ('val') { ... }
Cas par défaut	*ngSwitchDefault	@default { ... }

Exemple	<div [ngSwitch]="color">
	<p *ngSwitchCase="'red'">Rouge
	<p *ngSwitchCase="'blue'">Bleu
	<p *ngSwitchCase="'green'">Vert
	<p *ngSwitchDefault>Couleur inconnue
	
		@switch (color) {
		@case ('red') {
		 Rouge
		}
		@case ('blue') {
		 Bleu
		}
		@case ('green') {
		 Vert
		}
		@default {
		 Couleur inconnue
		}
		}

Directive structurelle : NgSwich

Exemple dans un composant standalone

1. Importer NgSwitch dans le composant

app.component.ts

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import {HeaderComponent} from './header/header.component';
import {NgSwitch, NgSwitchCase, NgSwitchDefault} from '@angular/common';

1+ usages  WiemAouididi *
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, HeaderComponent, NgSwitch, NgSwitchCase, NgSwitchDefault],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  role : string  = 'admin';
}

```

2. Utilisation

app.component.html

```html
<div [ngSwitch]="role">
  <p *ngSwitchCase="'admin'">Bienvenue, administrateur !</p>
  <p *ngSwitchCase="'user'">Bienvenue, utilisateur.</p>
  <p *ngSwitchDefault>Accès invité</p>
</div>

```

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>
```

[ngClass] VS [class]

- **[class]** est utile pour appliquer **une ou deux classes** spécifiques conditionnellement. C'est simple et lisible, mais ça devient plus compliqué si tu as besoin de plus de classes ou de conditions plus complexes.
- **[ngClass]** est plus flexible et puissant. Il te permet d’appliquer **plusieurs classes conditionnellement** avec des objets, des tableaux ou des expressions complexes, tout en gardant une syntaxe claire et lisible.

[ngClass] VS [class]

Exemple de la manipulation de deux classe CSS avec [class] :

```
<div [class.active]="isActive" [class.inactive]="!isActive" class="box">
 <p>Ce box est {{ isActive ? 'actif' : 'inactif' }}</p>
</div>
```

- [class] est utilisé pour appliquer ou supprimer une **seule classe CSS** en fonction d'une condition (ou de plusieurs conditions, mais une par une).
- [class] ne reçoit pas un objet mais chaque classe doit être manipulée individuellement avec une condition.

ngClass VS [class]

Exemple de la manipulation de deux classe CSS avec [ngClass] :

```
<div [ngClass]="{ 'active': isActive, 'inactive': !isActive }" class="box">
 <p>Ce box est {{ isActive ? 'actif' : 'inactif' }}</p>
</div>
```

[ngClass] prend un objet où chaque clé est le nom de la classe CSS et la valeur associée est la condition (vraie ou fausse). Si la condition est vraie, la classe est appliquée, sinon elle est retirée..

[ngStyle] VS [style]

- **[style]** est utilisé pour lier un style CSS spécifique à une propriété d'un élément HTML. Cette méthode permet de **modifier une seule propriété de style CSS** à la fois.

- **ngStyle** est plus flexible que [style], car il te permet d'appliquer plusieurs styles CSS en même temps via un objet JavaScript. Chaque propriété de l'objet représente une propriété CSS, et la valeur associée est la valeur du style.

[ngStyle] VS [style]

- **Exemple avec [style] :**

```
<div [style.background-color]="backgroundColor"Exemple avec [style] :

 [style.font-size]="fontSize">Contenu</div>
```

- **Exemple avec [ngStyle] :**

```
<div [ngStyle]="{ 'background-color': backgroundColor, 'font-size': fontSize
}">Contenu</div>
```

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**.	{{ 'bonjour' | uppercase }}	BONJOUR
lowercase	Convertit tout le texte en **minuscules**.	{{ 'ANGULAR' | lowercase }}	angular
titlecase	Met en **majuscules la première lettre** de chaque mot.	{{ 'bonjour tout le monde' | titlecase }}	Bonjour Tout Le Monde
date	Formate une **date** selon un format spécifique (short, longDate, etc.)	{{ today | date:'longDate' }}	29 juillet 2025
number	Formate les **nombres décimaux** selon une configuration (1.2-2 = min 1 chiffre entier, 2 chiffres après la virgule).	{{ 3.14159 | number:'1.2-2' }}	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.	{{ 1200 | currency:'EUR' }}	1 200,00 €
percent	Affiche une **valeur sous forme de pourcentage** (0.75 → 75%).	{{ 0.875 | percent }}	87.5%
slice	Coupe une **chaîne de caractères ou un tableau** (slice:1:3 → éléments 1 à 2).	{{ 'Angular' | slice:0:3 }}	Ang
json	Transforme un **objet JavaScript en texte JSON lisible** (utile pour le debug).	{{ user | json }}	{ "name": "Ali", "age": 25 }
async	Affiche **automatiquement la valeur d'un Observable ou Promise** dès qu'elle est disponible.	{{ userData$ | async }}	Résultat de l'observable

Les pipes : Utilisation des pipes intégrés

Exemple dans un composant standalone

1. Importer le pipe dans le composant

app.component.ts

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import {HeaderComponent} from './header/header.component';
import {DatePipe} from '@angular/common';


1+ usages  WiemAouididi *
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, HeaderComponent, DatePipe],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  currentDate : string = '2024-07-08';
}
```

2. Utilisation

app.component.html

```html
<div>
  <h1>Current date is {{ currentDate | date }}</h1>
</div>

```

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 intégrés

DatePipe : {{ value | date: format:timezone }}

Format	Résultat attendu
{{ today | date: 'short' }}	<!-- Exemple : 4/15/21, 2:30 PM -->
{{ today | date: 'medium' }}	<!-- Exemple : Apr 15, 2021, 2:30:00 PM -->
{{ today | date: 'long' }}	<!-- Exemple : April 15, 2021 at 2:30:00 PM UTC+2 -->
{{ today | date: 'fullDate' }}	<!-- Exemple : Thursday, April 15, 2021 -->
{{ today | date: 'dd/MM/yyyy' }}	<!-- Exemple : 15/04/2021 -->
{{ today | date: 'HH:mm:ss' }}	<!-- Exemple : 14:30:45 -->
{{ today | date: 'yyyy-MM-dd HH:mm' }}	<!-- Exemple : 2021-04-15 14:30 -->

Les pipes personnalisés

Les pipes personnalisés sont crés 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 :

Le prix est : {{ 12.34 | euro }}

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 1er 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