

INTRODUCTION AUX DIRECTIVES

DÉFINITION D'UNE DIRECTIVE

Les **directives** sont des éléments du langage **Angular** permettant d'étendre les capacités du **HTML** en ajoutant du comportement dynamique au sein des **templates**.

TYPES DE DIRECTIVES : ATTRIBUT, ÉLÉMENT, COMPOSANT

Il existe trois types de directives en Angular:

Type de directive	Description
Directives d'attribut	Modifient l'apparence ou le comportement d'un élément existant.
Directives élément	Créent des éléments HTML personnalisés.
Composants	Directives avec templates (les plus courantes).

DIRECTIVE ngIf

SYNTAXE

*NGIF="CONDITION"

```
<p *ngIf="condition">Cet élément s'affiche si la condition est true</p>
```

<NG-TEMPLATE> AVEC *NGIF

```
<ng-template [ngIf]="condition">
  <p>Cet élément s'affiche si la condition est true</p>
</ng-template>
```

EXEMPLES D'UTILISATION

AFFICHER UN ÉLÉMENT EN FONCTION D'UNE CONDITION

```
<p *ngIf="isAdmin">Seul l'administrateur voit ce texte</p>
```

MASQUER UN ÉLÉMENT EN FONCTION D'UNE CONDITION

```
<p *ngIf="!isAdmin">Si ce n'est pas un administrateur, cet élément est visible</p>
```

PRÉSENTATION DES DIRECTIVES

INTRODUCTION AUX DIRECTIVES

Les **directives** sont des fonctionnalités d'**Angular** qui permettent de manipuler et de modifier le **DOM**.

TYPES DE DIRECTIVES

Type	Description
Attribut	Manipulation des éléments du DOM
Élément	Création et destruction d'éléments du DOM
Composant	Combinaison d'un élément et d'une directive

DIRECTIVE ngIf

DIRECTIVE ngIf - SYNTAXE

- Pour utiliser `ngIf`, ajoutez `*ngIf="condition"` sur l'élément que vous souhaitez **afficher** ou **masquer**.
- Vous pouvez également utiliser `<ng-template>` avec `*ngIf` pour une **plus grande flexibilité**.

```
<!-- Exemple avec un élément div -->
<div *ngIf="condition">
    Contenu affiché si la condition est vraie
</div>

<!-- Exemple avec ng-template -->
<ng-template [ngIf]="condition">
    <div>
        Contenu affiché si la condition est vraie
    </div>
</ng-template>
```

DIRECTIVE ngIf - EXEMPLES D'UTILISATION

AFFICHER UN ÉLÉMENT EN FONCTION D'UNE CONDITION

```
<p *ngIf="isOk">Tout va bien!</p>
```

MASQUER UN ÉLÉMENT EN FONCTION D'UNE CONDITION

```
<p *ngIf="!isOk">Il y a un problème!</p>
```

UTILISATION AVEC ngElse ET ngIf-then

SYNTAXE

```
<p *ngIf="condition; else elseBlock">...</p>
<ng-template #elseBlock>...</ng-template>
```

EXEMPLES D'UTILISATION

```
<p *ngIf="isOk; else notOk">Tout va bien!</p>
<ng-template #notOk><p>Il y a un problème!</p></ng-template>
```

PRÉSENTATION DES DIRECTIVES : DIRECTIVE NGFOR

DIRECTIVE NGFOR

La directive `**ngFor**` permet de créer des éléments de liste ou de tableau de manière répétée en fonction des éléments d'un tableau ou d'un objet.

Exemple d'utilisation :

```
<ul>
  <li *ngFor="let item of items">{{item}}</li>
</ul>
```

SYNTAXE

Pour utiliser `**ngFor**`, on utilise la syntaxe suivante :

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

*NGFOR AVEC INDEX, FIRST, LAST, EVEN, ODD

ngFor fournit des **variables supplémentaires** pour mieux contrôler le rendu des éléments :

Variable	Description
index	L'index de l'élément actuel
first	Vrai si c'est le premier élément
last	Vrai si c'est le dernier élément
even	Vrai si l'index est pair
odd	Vrai si l'index est impair

EXEMPLES D'UTILISATION

CRÉER UNE LISTE DYNAMIQUE

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

CRÉER UN TABLEAU DYNAMIQUE

```
<table>
  <tr *ngFor="let item of items">
    <td>{{ item.id }}</td>
    <td>{{ item.name }}</td>
  </tr>
</table>
```

Colonne 1 Colonne 2

ID	Nom
----	-----



DIRECTIVE ngSwitch

DIRECTIVE ngSwitch

La directive `**ngSwitch**` permet de créer une **structure de contrôle** pour afficher du contenu en fonction d'une **expression** ou d'une **condition**.

Exemple :

```
<div [ngSwitch]="condition">
  <div *ngSwitchCase="'cas1'">Contenu pour le cas 1</div>
  <div *ngSwitchCase="'cas2'">Contenu pour le cas 2</div>
  <div *ngSwitchDefault>Contenu par défaut si aucun cas ne correspond</div>
</div>
```

SYNTAXE

Pour utiliser ngSwitch, il faut déclarer l'**expression** à évaluer et les différents **cas de rendu**.

```
<div [ngSwitch]="expression">
  <div *ngSwitchCase="'case1'">Contenu pour case1</div>
  <div *ngSwitchCase="'case2'">Contenu pour case2</div>
  <div *ngSwitchDefault>Contenu par défaut</div>
</div>
```

*NGSWITCHCASE

*ngSwitchCase permet de définir un cas dans la structure **ngSwitch**. Si l'expression évaluée correspond à la valeur du cas, le contenu associé sera affiché.

```
<div [ngSwitch]="expression">
  <div *ngSwitchCase="'valeur1'">Contenu pour valeur1</div>
  <div *ngSwitchCase="'valeur2'">Contenu pour valeur2</div>
  <div *ngSwitchDefault>Contenu par défaut</div>
</div>
```

*NGSWITCHDEFAULT

`*ngSwitchDefault` est utilisé pour définir un contenu par **défaut**, qui s'affiche si aucun des cas spécifiés ne correspond à l'expression évaluée.

EXEMPLES D'UTILISATION

```
<!-- Exemple avec une variable "color" -->
<div [ngSwitch]="color">
  <div *ngSwitchCase="'red'">La couleur est rouge</div>
  <div *ngSwitchCase="'blue'">La couleur est bleue</div>
  <div *ngSwitchDefault>La couleur n'est ni rouge ni bleue</div>
</div>

<!-- Exemple avec une fonction "getStatus()" -->
<div [ngSwitch]="getStatus()">
  <div *ngSwitchCase="'online'">L'utilisateur est en ligne</div>
  <div *ngSwitchCase="'offline'">L'utilisateur est hors ligne</div>
  <div *ngSwitchDefault>Statut inconnu</div>
</div>
```

DIRECTIVES PERSONNALISÉES

CRÉATION D'UNE DIRECTIVE PERSONNALISÉE

Une directive personnalisée est une **fonctionnalité** créée pour étendre ou modifier le **comportement** d'un élément du DOM.

SYNTAXE

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

@Directive({
  selector: '[appDirective]'
})
export class AppDirective {
  constructor() { }
}
```

Note : Cette slide présente la syntaxe d'une **Directive** Angular. Les Directives permettent de manipuler le DOM et ajouter des comportements personnalisés aux éléments HTML.

EXEMPLES D'UTILISATION

1. Appliquer la **directive** à un élément HTML :

```
<div appDirective></div>
```

2. Utiliser la **directive** avec un paramètre :

```
<div appDirective="param"></div>
```

INTERACTION AVEC LE DOM

Dans une **directive personnalisée**, on utilise **Renderer2** et **ElementRef** pour interagir avec le **DOM**.

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

@Directive({
  selector: '[appCustomDirective]'
})
export class CustomDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {
    this.renderer.setStyle(this.el.nativeElement, 'color', 'blue');
  }
}
```

RENDERER2

Renderer2 est une classe **abstraite** d'Angular permettant de **manipuler** les éléments du **DOM**.

ELEMENTREF

ElementRef est une classe d'Angular permettant d'accéder aux éléments du **DOM**.

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

@Directive({
  selector: '[appDirective]'
})
export class AppDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {
    // Exemple d'utilisation
    this.renderer.setStyle(this.el.nativeElement, 'color', 'blue');
  }
}
```

MANIPULATION DES CLASSES ET DES STYLES

On peut manipuler les **classes** et les **styles** des éléments du DOM à l'aide de ****HostBinding**** et ****HostListener****.

Directive	Description
@HostBinding	Permet de lier une propriété du composant à un attribut HTML
@HostListener	Permet d'écouter un événement depuis l'élément hôte

HOSTBINDING

HostBinding est une **directive** permettant de lier une propriété d'un élément **DOM** avec une propriété de la directive.

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

@Directive({
  selector: '[appDirective]'
})
export class AppDirective {
  @HostBinding('class.active') isActive = false;
}
```

HOSTLISTENER

HostListener est une **directive** permettant d'écouter les **événements du DOM** et d'exécuter du code en fonction de ces événements.

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

@Directive({
  selector: '[appDirective]'
})
export class AppDirective {
  @HostListener('click') onClick() {
    // Exemple d'utilisation
    console.log('Clicked');
  }
}
```

DIRECTIVES STRUCTURELLES AVANCÉES

ngContainer

Le `ngContainer` est une **directive spéciale** servant à contenir des **directives structurelles** sans créer de nœuds DOM supplémentaires.

SYNTAXE ngContainer

```
<ng-container *ngIf="condition">
  <p>Contenu conditionnel</p>
</ng-container>
```

EXEMPLES D'UTILISATION ngContainer

```
<ng-container *ngFor="let item of items">
  <p>{{ item.title }}</p>
  <span>{{ item.description }}</span>
</ng-container>
```

ngTemplateOutlet

Le ****ngTemplateOutlet**** permet d'insérer un modèle (`<ng-template>`) à un endroit spécifié dans le **DOM**.

```
<div *ngFor="let item of listeItems">
  <ng-container [ngTemplateOutlet]="modele" [ngTemplateOutletContext]="{data: item}"></ng-container>
</div>

<ng-template #modele let-data="data">
  {{data.nom}} - {{data.prix}}
</ng-template>
```

SYNTAXE ngTemplateOutlet

```
<ng-container [ngTemplateOutlet]="templateRef"></ng-container>

<ng-template #templateRef>
  <p>Contenu du modèle</p>
</ng-template>
```

EXEMPLES D'UTILISATION ngTemplateOutlet

```
<ng-container *ngFor="let item of items" [ngTemplateOutlet]="itemTemplate" [ngTemplateOutletContext]="$implicit: item">
</ng-container>

<ng-template #itemTemplate let-item="item">
  <p>{{ item.title }}</p>
  <span>{{ item.description }}</span>
</ng-template>
```

ngTemplateOutletContext

Le **ngTemplateOutletContext** permet de passer des données au modèle inséré à l'aide de **ngTemplateOutlet**.

```
<ng-container *ngTemplateOutlet="template; context: data"></ng-container>
<ng-template #template let-item="itemData">
  <p>{{ item.title }}</p>
  <p>{{ item.description }}</p>
</ng-template>
```

SYNTAXE ngTemplateOutletContext

```
<ng-container [ngTemplateOutlet]="templateRef" [ngTemplateOutletContext]="{$implicit: data}">  
</ng-container>
```

EXEMPLES D'UTILISATION ngTemplateOutletContext

```
<ng-container *ngFor="let item of items" [ngTemplateOutlet]="itemTemplate" [ngTemplateOutletContext]="$implicit: item, index: i">
</ng-container>

<ng-template #itemTemplate let-item="item" let-index="index">
  <p>{{ index + 1 }}. {{ item.title }}</p>
  <span>{{ item.description }}</span>
</ng-template>
```

BONNES PRATIQUES

ARCHITECTURE ET ORGANISATION DU CODE

- Structure de dossier **modulaire** :
 - Un dossier par module
 - Un fichier par composant/directive/service
- Utiliser des **modules** pour :
 - Regrouper les fonctionnalités associées
 - Isoler les responsabilités
 - Organiser et maintenir le code

PERFORMANCES ET OPTIMISATION

- Utiliser la détection de changements appropriée :
 - `**ChangeDetectionStrategy.OnPush**` pour les composants avec des entrées immuables
- Utiliser le suivi personnalisé avec `*ngFor` :
 - `*ngFor="let item of items; trackBy: trackById"`
 - Améliore les performances en évitant la destruction et la recréation d'éléments inutiles
- Utiliser l'opérateur `**async**` pour les données asynchrones :
 - Évite les souscriptions manuelles et les fuites de mémoire
 - Permet de gérer l'état de chargement et d'erreur

BONNES PRATIQUES (SUITE)

- Utiliser les **modèles de conception appropriés** :
 - **Observer/Subject** pour la communication entre composants
 - **Injection de dépendances** pour les services
- Éviter d'accéder directement au **DOM** :
 - Utiliser **Renderer2** pour les manipulations DOM
 - Utiliser **ElementRef** avec prudence
- Garder les **directives** simples et maintenables :
 - Isoler ou déléguer la logique complexe dans les services
 - Favoriser la **composition** des directives sur l'héritage

