INTRODUCTION AUX COMPOSANTS

DÉFINITION D'UN COMPOSANT

Un **composant** est une unité modulaire de l'**interface utilisateur**, composée de **HTML**, **CSS** et **logique métier**. Ils forment la base d'une application Angular.

RÔLE ET UTILITÉ DES COMPOSANTS

- Construire des **interfaces utilisateur complexes**
- **Réutilisation de code**
- **Encapsulation** de la logique métier
- Faciliter la **maintenance**
- Améliorer la **lisibilité**

STRUCTURE D'UN COMPOSANT

Un composant **Angular** est composé de :

Fichiers	Description
Fichier TypeScript	Contient la logique métier
Fichier HTML	Contient la structure de l'interface
Fichier CSS	Contient le style de l'interface
Fichier de test (optionnel)	Pour les tests unitaires

CRÉATION D'UN COMPOSANT

UTILISATION DU CLI ANGULAR

COMMANDE DE GÉNÉRATION

Pour générer un **composant** en utilisant le CLI **Angular**, utilisez la commande suivante :

```
ng generate component nom-du-composant
```

ou

```
ng g c nom-du-composant
```

OPTIONS DE LA COMMANDE

Option	Description
`--skipTests`	Ne génère pas de fichier de test
`--inlineTemplate`	Utilise un template inline plutôt qu'un fichier HTML externe
`--inlineStyle`	Utilise un style inline plutôt qu'un fichier CSS externe

CRÉATION MANUELLE

FICHIER TYPESCRIPT

Créez un fichier `nom-du-composant.component.ts` avec le contenu suivant :

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

@Component({
  selector: 'app-nom-du-composant',
  templateUrl: './nom-du-composant.component.html',
  styleUrls: ['./nom-du-composant.component.css']
})
export class NomDuComposantComponent {
  //... **propriétés** et **méthodes**
}
```

FICHIER HTML

Créez un fichier `nom-du-composant.component.html` et ajoutez le code HTML pour le **template** du composant.

FICHIER CSS

Créez un fichier `nom-du-composant.component.css` et ajoutez les styles **CSS** pour le composant.

ANATOMIE D'UN COMPOSANT

IMPORTS ET DÉCORATEURS

DÉCORATEUR @COMPONENT

Le décorateur `@Component` permet de définir une classe en tant que **composant Angular**.

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

@Component({
  selector: 'app-mon-composant',
  templateUrl: './mon-composant.component.html',
  styleUrls: ['./mon-composant.component.css']
})
export class MonComposantComponent {}
```

IMPORT DES CLASSES ANGULAR

Il est nécessaire d'importer les **classes Angular** nécessaires pour chaque composant. Exemple :

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

CLASSE DU COMPOSANT

DÉCLARATION DES PROPRIÉTÉS

Les **propriétés** d'un composant sont déclarées et initialisées dans la **classe**.

```
export class MonComposantComponent {
  maPropriete: string = 'Hello World';
}
```

MÉTHODES ET ÉVÉNEMENTS

Les **méthodes** et **événements** du composant sont déclarés dans la classe.

```
export class MonComposantComponent {
  onClick() {
    alert('Bouton cliqué!');
  }
}
```

TEMPLATE

LIAISON DE DONNÉES

La liaison de données permet d'**interagir** entre le modèle (composant) et la vue (template).

```
<p>{{ maPropriete }}</p>
<button (click)="onClick()">Cliquez-moi!</button>
```

INTERPOLATION ET EXPRESSIONS

L'interpolation permet d'afficher des données ou d'exécuter des **expressions JavaScript**.

```
<p>{{ maPropriete.toUpperCase() }}</p>
```

DIRECTIVES ET PIPES

Les **directives** et les **pipes** permettent de modifier le comportement des éléments HTML ou de filtrer des données.

```
<div *ngIf="maPropriete">
  {{ maPropriete | uppercase }}
</div>
```

INTERACTION ENTRE COMPOSANTS

COMMUNICATION PARENT-ENFANT

Les **composants Angular** peuvent interagir entre eux de différentes manières.

- Utilisation de l'**Input**
- Utilisation des **Evénements**
- Utilisation de **ViewChild**

Méthode	Description	Exemple
Input	Passer la donnée du parent vers l'enfant	`<enfant [inputVariable]="parentVariable"></enfant>`
Evénements	Écouter les événements émis par l'enfant dans le parent	`<enfant (outputEvent)="parentEventHandler($event)"></enfant>`
ViewChild	Parent sélectionne les éléments (=enfant(s)) grâce à une référence locale	`@ViewChild('child') childComponent: ChildComponent;`

UTILISATION DE `@Input`

Les propriétés `@Input` permettent de passer des **données** du **composant parent** au **composant enfant**.

```ts
// enfant.component.ts
@Input() nom: string;
```

```html
<!-- parent.component.html -->
<app-enfant [nom]="nomParent"></app-enfant>
```

UTILISATION DE `@Output`

Les événements `@Output` permettent au **composant enfant** de communiquer avec son **parent**.

```ts
// enfant.component.ts
@Output() action = new EventEmitter<void>();

onAction() {
  this.action.emit();
}
```

```html
<!-- parent.component.html -->
<app-enfant (action)="onActionEnfant()"></app-enfant>
```

UTILISATION DE VIEWCHILD/CONTENTCHILD

`ViewChild` et `ContentChild` permettent d'accéder aux instances des composants enfants.

```
// parent.component.ts
@ViewChild(EnfantComponent) enfant: EnfantComponent;
```

COMMUNICATION ENFANT-PARENT

PASSER DES DONNÉES VIA LES ÉVÉNEMENTS

Il est possible de passer des données du **composant enfant** vers le **composant parent** en utilisant les événements `@Output`.

```ts
// enfant.component.ts
@Output() action = new EventEmitter<number>();

onAction() {
  this.action.emit(42);
}
```

```html
<!-- parent.component.html -->
<app-enfant (action)="onActionEnfant($event)"></app-enfant>
```

COMMUNICATION ENTRE COMPOSANTS NON LIÉS

UTILISATION D'UN SERVICE COMMUN

Un service peut être utilisé pour partager des **données** et des **méthodes** entre des composants qui ne sont pas directement liés.

```
// shared.service.ts
@Injectable({ providedIn: 'root' })
export class SharedService {
  private data: string;

  setData(data: string) {
    this.data = data;
  }

  getData() {
    return this.data;
  }
}
```

```
// composantA.component.ts
@Component(...)
export class ComposantAComponent {
  constructor(private sharedService: SharedService) {}

  updateData() {
    this.sharedService.setData('Nouvelle donnée');
  }
}

// composantB.component.ts
@Component(...)
export class ComposantBComponent {
  constructor(private sharedService: SharedService) {}
```

CYCLE DE VIE D'UN COMPOSANT

CYCLE DE VIE D'UN COMPOSANT

Chaque composant **Angular** passe par plusieurs étapes au cours de son cycle de vie, depuis sa création jusqu'à sa destruction.

- Création du composant
- Mise à jour des données entrantes
- Détection des changements
- Destruction du composant

Méthode du cycle de vie	Description
ngOnChanges	Appelé lorsque des données entrantes changent
ngOnInit	Appelé une fois après la création du composant
ngDoCheck	Appelé lors de chaque détection des changements
ngAfterContentInit	Appelé après que le contenu du composant a été dans l'arbre DOM
ngAfterViewInit	Appelé après que les vues enfants ont été mises à jour

HOOKS DU CYCLE DE VIE

Angular fournit des **hooks du cycle de vie** pour réagir et exécuter du code lors de ces étapes :

Hook	Description
`ngOnInit`	Lors de la création du composant
`ngOnChanges`	Lorsqu'une propriété liée par un input subit un changement
`ngDoCheck`	Après la détection des changements pour les inputs
`ngOnDestroy`	Avant la destruction du composant
`ngAfterViewInit`	Après l'initialisation de la vue
`ngAfterViewChecked`	Après la détection des changements pour la vue
`ngAfterContentInit`	Après l'insertion du contenu par projection
`ngAfterContentChecked`	Après la détection des changements pour le contenu projeté

NGONINIT

Le hook **ngOnInit** est exécuté une fois que le **composant** est initialisé et après que ses propriétés **@Input** ont été définies.

```
ngOnInit(): void {
  // Code à exécuter lors de l'initialisation
}
```

NGONCHANGES

Le hook `ngOnChanges` est exécuté lorsque l'une des propriétés **@Input** change. Il reçoit un objet contenant les changements.

```
ngOnChanges(changes: SimpleChanges): void {
  // Code à exécuter lors des changements
}
```

Propriété	Description
previousValue	La valeur précédente de la propriété avant le changement
currentValue	La valeur actuelle de la propriété après le changement
firstChange	Un booléen indiquant si c'est le premier changement de la propriété

NGDOCHECK

Le hook `**ngDoCheck**` est exécuté lors de chaque cycle de **détection de changement**.

```
ngDoCheck(): void {
  // Code à exécuter lors de la détection de changement
}
```

NGONDESTROY

Le hook `ngOnDestroy` est exécuté juste avant la **destruction** du composant.

```
ngOnDestroy(): void {
  // Code à exécuter lors de la destruction
}
```

NGAFTERVIEWINIT

Le hook `ngAfterViewInit` est exécuté après que les **vues enfants** (et leurs vues) ont été **initialisées**.

```
ngAfterViewInit(): void {
  // Code à exécuter après l'initialisation des vues enfants
}
```

NGAFTERVIEWCHECKED

Le hook `ngAfterViewChecked` est exécuté après chaque vérification des **vues enfants** (et leurs vues).

```
ngAfterViewChecked(): void {
  // Code à exécuter après la vérification des vues enfants
}
```

NGAFTERCONTENTINIT

Le hook `ngAfterContentInit` est exécuté après que le contenu projeté dans le composant a été initialisé.

```
ngAfterContentInit(): void {
  // Code à exécuter après l'initialisation du contenu projeté
}
```

NGAFTERCONTENTCHECKED

Le hook `ngAfterContentChecked` est exécuté après chaque vérification du **contenu projeté** dans le composant.

```
ngAfterContentChecked(): void {
  // Code à exécuter après la vérification du contenu projeté
}
```

UTILISATION ET CAS D'USAGE DES HOOKS

Les **hooks** du cycle de vie vous permettent de prendre en charge des cas d'usage spécifiques lorsque les **propriétés** changent, le **contenu** est projeté, etc. Utilisez-les en fonction de vos besoins pour gérer le **cycle de vie** de vos composants.

Hooks du cycle de vie	Description
OnInit	Appelé lors de l'initialisation du composant
OnChanges	Appelé lorsque les propriétés liées sont modifiées
OnDestroy	Appelé avant que le composant ne soit détruit

COMPOSANTS DYNAMIQUES

COMPOSANTS DYNAMIQUES

Les **composants dynamiques** sont créés et détruits à la volée pendant l'exécution de l'application en fonction des besoins.

CRÉATION DE COMPOSANTS À LA VOLÉE

Pour créer des composants **dynamiques**, utilisez `**ComponentFactoryResolver**` et `**ViewContainerRef**`.

```
@ViewChild('container', { read: ViewContainerRef }) container: ViewContainerRef;

constructor(private componentFactoryResolver: ComponentFactoryResolver) {}

createComponent(component: Type<any>) {
  const factory = this.componentFactoryResolver.resolveComponentFactory(component);
  this.container.createComponent(factory);
}
```

UTILISATION DE ComponentFactoryResolver

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

constructor(private componentFactoryResolver: ComponentFactoryResolver) {}

createDynamicComponent() {
  const factory = this.componentFactoryResolver.resolveComponentFactory(MyDynamicComponent);
  // ...
}
```

UTILISATION DE `ViewContainerRef`

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

constructor(private viewContainerRef: ViewContainerRef) {}

createDynamicComponent() {
  // ...
  const componentRef = this.viewContainerRef.createComponent(factory);
}
```

DESTRUCTION DE COMPOSANTS DYNAMIQUES

Pour détruire un composant dynamique, utilisez la méthode `destroy` de `**ComponentRef**`.

```
componentRef.destroy();
```

Voici un exemple complet de création et de destruction d'un composant dynamique :

```
import { Component, ComponentFactoryResolver, ViewContainerRef } from '@angular/core';
import { MyDynamicComponent } from './my-dynamic.component';

@Component({
  selector: 'app-root',
  template: `
    <button (click)="createDynamicComponent()">Create</button>
    <button (click)="destroyDynamicComponent()">Destroy</button>
    <ng-container #container></ng-container>
  `,
})
export class AppComponent {
  private componentRef;

  constructor(
```

BONNES PRATIQUES

ORGANISATION ET MODULARISATION

- Regrouper les **composants** par fonctionnalités
- Utiliser des **modules** pour faciliter les imports
- Créer un **CoreModule** pour les services et composants globaux

PASSAGE DE DONNÉES ET ÉVÉNEMENTS

- Utiliser **@Input**, **@Output** et **EventEmitter**
- Minimiser la communication directe entre les composants

@Input	**@Output**
Reçoit des données d'un composant parent	Envoie des données à un composant parent
Permet aux composants enfants de communiquer avec les parents	Utilise l'objet EventEmitter pour émettre des événements

DÉTACHEMENT DE LA DÉTECTION DE CHANGEMENT

- Utiliser **ChangeDetectionStrategy.OnPush**
- Mettre à jour les références plutôt que de muter les **objets**
- Utiliser la détection de changements **manuelle** avec **ChangeDetectorRef**

PERFORMANCES ET OPTIMISATIONS

- Éviter d'utiliser **ngDoCheck** et **ngOnChanges** pour les traitements lourds
- Utiliser les **pipes pures** pour les transformations de données
- Séparer les composants en **composants intelligents** (containers) et composants purement **présentationnels** (dumb components)