Tutorial : Gestion des components

Partie : 4

Composition des composants dans Angular

La **composition des composants** dans Angular consiste à organiser plusieurs composants pour créer une application modulaire, réutilisable et bien structurée. Chaque composant joue un rôle spécifique, et ils interagissent les uns avec les autres.

---

1. Création de composants imbriqués

Les composants Angular peuvent être imbriqués : un composant parent peut inclure un ou plusieurs composants enfants.

Exemple :

Créons une application avec un **composant parent** et un **composant enfant**.

1. **Générer les composants** :

   ```
   bash

   ng generate component parent
   ng generate component child
   ```

2. **Composant enfant (`child.component.ts`)** :

   ```
   typescript

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

   @Component({
     selector: 'app-child',
     template: `
       <p>Je suis un composant enfant !</p>
     `,
     styles: [`
       p {
         color: blue;
         font-style: italic;
       }
     `]
   })
   export class ChildComponent {}
   ```

3. **Composant parent (`parent.component.html`)** : Ajoutez le sélecteur du composant enfant (`<app-child>`) dans le template du parent :

```
html

<h1>Composant Parent</h1>
<app-child></app-child> <!-- Appel du composant enfant -->
```

4. **Résultat** : Le navigateur affiche le contenu du composant parent et du composant enfant :

```
Composant Parent
Je suis un composant enfant !
```

---

2. Passage de données du parent vers l'enfant

Le **decorator @Input** permet de transmettre des données d'un composant parent à un composant enfant.

Exemple :

1. **Dans le composant enfant (`child.component.ts`)** : Ajoutez une propriété décorée avec `@Input` pour recevoir des données :

```
typescript

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

@Component({
  selector: 'app-child',
  template: `
    <p>Message du parent : {{ parentMessage }}</p>
  `
})
export class ChildComponent {
  @Input() parentMessage!: string; // Propriété venant du parent
}
```

2. **Dans le composant parent (`parent.component.html`)** : Passez une valeur à l'enfant via un **binding de propriété** :

```
html

<h1>Composant Parent</h1>
<app-child [parentMessage]="'Bonjour, enfant !'"></app-child>
```

3. **Résultat** :

```
bash

Composant Parent
Message du parent : Bonjour, enfant !
```

---

3. Émission d'événements de l'enfant vers le parent

Le **decorator @Output** permet à un composant enfant d'envoyer des données ou des événements au composant parent.

Exemple :

1. **Dans le composant enfant (`child.component.ts`)** : Utilisez `@Output` et un **EventEmitter** pour envoyer un événement au parent :

   ```
   typescript

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

   @Component({
     selector: 'app-child',
     template: `
       <button (click)="notifyParent()">Notifier le parent</button>
     `
   })
   export class ChildComponent {
     @Output() notify = new EventEmitter<string>();

     notifyParent() {
       this.notify.emit('Message de l'enfant !');
     }
   }
   ```

2. **Dans le composant parent (`parent.component.html`)** : Capturez l'événement émis par l'enfant avec `(notify)` :

   ```
   html

   <h1>Composant Parent</h1>
   <app-child (notify)="handleNotification($event)"></app-child>
   ```

3. **Dans le composant parent (`parent.component.ts`)** : Implémentez la méthode appelée lorsqu'un événement est capturé :

   ```
   typescript

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

   @Component({
     selector: 'app-parent',
     templateUrl: './parent.component.html'
   })
   export class ParentComponent {
     handleNotification(message: string) {
       console.log('Notification reçue :', message);
     }
   }
   ```

4. **Résultat** : Lorsque vous cliquez sur le bouton, le message "Message de l'enfant !" s'affiche dans la console.

---

4. Composition hiérarchique

Angular vous permet de créer une structure hiérarchique :

- **AppComponent** : Composant racine.
- **Sous-composants** : Divisez les sections de votre application (en-tête, pied de page, navigation, contenu, etc.) en plusieurs composants.

Exemple d'organisation hiérarchique :

```
markdown

AppComponent
  ├── HeaderComponent
  ├── FooterComponent
  └── ContentComponent
        ├── SidebarComponent
        └── MainContentComponent
```

Avantages de la composition des composants

1. **Réutilisabilité** : Les composants peuvent être réutilisés dans plusieurs parties de l’application.
2. **Lisibilité** : Division du code en parties logiques et cohérentes.
3. **Modularité** : Les composants sont indépendants et peuvent être testés séparément.

Partie : 5

Contrôle de flux dans les composants Angular : *ngIf

En Angular, la directive structurelle `*ngIf` est utilisée pour afficher ou masquer des éléments HTML dynamiquement en fonction d'une condition.

---

1. Syntaxe de base de `*ngIf`

La directive `*ngIf` évalue une expression booléenne. Si l'expression est `true`, l'élément est affiché ; sinon, il est supprimé du DOM.

Exemple :

```
html

<div *ngIf="condition">
  Ce texte est affiché si la condition est vraie.
</div>
```

---

2. Utiliser `*ngIf` dans un composant

1. **Composant TypeScript :** Déclarez une propriété pour la condition.

   ```
   typescript

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

   @Component({
     selector: 'app-mon-composant',
     templateUrl: './mon-composant.component.html',
   })
   export class MonComposant {
     afficherTexte: boolean = true; // Variable conditionnelle
   }
   ```

2. **Template HTML :** Ajoutez `*ngIf` pour conditionner l'affichage d'un élément.

   ```
   html

   <button (click)="afficherTexte = !afficherTexte">Afficher /
   Masquer</button>

   <p *ngIf="afficherTexte">Ce texte peut être affiché ou masqué.</p>
   ```

---

3. Alternative avec `else`

Angular permet d'utiliser une alternative avec la syntaxe `else`. Cela affiche un autre élément lorsque la condition est fausse.

Exemple :

```
html

<div *ngIf="afficherTexte; else texteAlternatif">
  Ce texte est affiché si afficherTexte est vrai.
</div>

<ng-template #texteAlternatif>
  <p>Ce texte est affiché si afficherTexte est faux.</p>
</ng-template>
```

---

4. Combinaison avec `then` et `else`

Pour gérer des flux complexes, vous pouvez utiliser `*ngIf` avec `then` et `else`.

Exemple :

```
html

<div *ngIf="condition; then blocVrai; else blocFaux"></div>

<ng-template #blocVrai>
  <p>Condition vraie !</p>
</ng-template>

<ng-template #blocFaux>
  <p>Condition fausse.</p>
</ng-template>
```

---

5. Différence avec les classes CSS

Contrairement à l'utilisation de classes CSS pour masquer un élément (`display: none`), `*ngIf` supprime complètement l'élément du DOM lorsqu'il n'est pas affiché. Cela améliore les performances en réduisant les calculs inutiles.

---

6. Exemple complet

1. **Composant TypeScript :**

   ```
   typescript

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

   @Component({
     selector: 'app-exemple',
     templateUrl: './exemple.component.html',
   ```

```
})
export class ExempleComponent {
  utilisateurConnecte: boolean = false;

  seConnecter() {
    this.utilisateurConnecte = true;
  }

  seDeconnecter() {
    this.utilisateurConnecte = false;
  }
}
```

2. **Template HTML :**

```
html

<div *ngIf="utilisateurConnecte; else nonConnecte">
  <p>Bienvenue, utilisateur !</p>
  <button (click)="seDeconnecter()">Se déconnecter</button>
</div>

<ng-template #nonConnecte>
  <p>Vous n'êtes pas connecté.</p>
  <button (click)="seConnecter()">Se connecter</button>
</ng-template>
```

Résultat :

- Si `utilisateurConnecte` est `true` : Affiche "Bienvenue, utilisateur !".
- Si `utilisateurConnecte` est `false` : Affiche "Vous n'êtes pas connecté."

---

7. Astuce : Opérateur logique

Vous pouvez utiliser des opérateurs logiques pour combiner plusieurs conditions.

Exemple :

```
html

<div *ngIf="utilisateurConnecte && roleAdmin">
  <p>Bienvenue, administrateur !</p>
</div>
```

`*ngIf` est une directive puissante pour le contrôle de flux dans Angular. Elle permet de gérer dynamiquement l'affichage des éléments de l'interface en fonction des données de l'application.

Partie : 6

Contrôle de flux dans Angular : *ngFor

La directive structurelle `*ngFor` en Angular est utilisée pour parcourir des collections (comme des tableaux) et afficher des éléments répétés dans le DOM pour chaque élément de la collection.

---

1. Syntaxe de base de `*ngFor`

La syntaxe typique de `*ngFor` est :

```
html

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

- **`item`** : La variable locale qui représente chaque élément de la collection.
- **`items`** : La collection (tableau, liste, etc.) à parcourir.

---

2. Utiliser `*ngFor` dans un composant

1. **Composant TypeScript :** Déclarez une collection dans le composant.

   ```
   typescript

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

   @Component({
     selector: 'app-mon-composant',
     templateUrl: './mon-composant.component.html',
   })
   export class MonComposant {
     fruits: string[] = ['Pomme', 'Banane', 'Orange', 'Mangue'];
   }
   ```

2. **Template HTML :** Utilisez `*ngFor` pour parcourir le tableau :

   ```
   html

   <ul>
     <li *ngFor="let fruit of fruits">
       {{ fruit }}
     </li>
   </ul>
   ```

Résultat :

```
diff

- Pomme
- Banane
- Orange
- Mangue
```

---

3. Accéder à l'index des éléments

Vous pouvez accéder à l'index de chaque élément à l'aide de `let i = index`.

Exemple :

```
html

<ul>
  <li *ngFor="let fruit of fruits; let i = index">
    {{ i + 1 }}. {{ fruit }}
  </li>
</ul>
```

Résultat :

```
markdown

1. Pomme
2. Banane
3. Orange
4. Mangue
```

---

4. Utilisation des propriétés supplémentaires

Angular fournit plusieurs variables locales utiles dans `*ngFor` :

- **`index`** : L'index de l'élément courant.
- **`first`** : `true` si c'est le premier élément.
- **`last`** : `true` si c'est le dernier élément.
- **`even`** : `true` si l'index est pair.
- **`odd`** : `true` si l'index est impair.

Exemple :

```
html

<ul>
  <li *ngFor="let fruit of fruits; let i = index; let isEven = even">
    <span *ngIf="isEven">(Pair) </span>{{ i + 1 }}. {{ fruit }}
  </li>
</ul>
```

---

5. Parcours d'objets complexes

Pour des objets avec des propriétés, utilisez le binding pour accéder aux valeurs.

Exemple :

1. **Composant TypeScript :**

```
typescript

personnes = [
  { nom: 'Alice', age: 25 },
  { nom: 'Bob', age: 30 },
  { nom: 'Charlie', age: 35 },
];
```

2. **Template HTML :**

```
html

<ul>
  <li *ngFor="let personne of personnes">
    {{ personne.nom }} - {{ personne.age }} ans
  </li>
</ul>
```

Résultat :

```
diff

- Alice - 25 ans
- Bob - 30 ans
- Charlie - 35 ans
```

---

6. Suivi des éléments avec `trackBy`

Par défaut, Angular recrée chaque élément du DOM lorsque la collection change, même si certains éléments n'ont pas changé. Pour améliorer les performances, utilisez une fonction `trackBy` pour identifier les éléments de manière unique.

Exemple :

1. **Composant TypeScript :**

```
typescript

personnes = [
  { id: 1, nom: 'Alice' },
  { id: 2, nom: 'Bob' },
  { id: 3, nom: 'Charlie' },
];

trackById(index: number, personne: any): number {
  return personne.id;
}
```

2. **Template HTML :**

```
html

<ul>
  <li *ngFor="let personne of personnes; trackBy: trackById">
    {{ personne.nom }}
  </li>
</ul>
```

---

7. Combinaison avec `*ngIf`

Vous pouvez combiner `*ngFor` et `*ngIf` pour afficher des éléments conditionnellement.

Exemple :

```
html

<ul>
  <li *ngFor="let fruit of fruits" *ngIf="fruit !== 'Orange'">
    {{ fruit }}
  </li>
</ul>
```

Astuce : Pour éviter une combinaison confuse, utilisez une **ng-container** :

```
html

<ng-container *ngFor="let fruit of fruits">
  <li *ngIf="fruit !== 'Orange'">{{ fruit }}</li>
</ng-container>
```

Avantages de `*ngFor`

- Simplifie la création d'éléments répétitifs.
- Gère efficacement les collections dynamiques.
- Fournit des fonctionnalités avancées comme `trackBy` pour améliorer les performances.

`*ngFor` est une directive essentielle pour gérer des données dynamiques et construire des interfaces réactives dans Angular.