Angular : concepts de base

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Angular

4 modes de liaison (binding) avec **Angular**

- One way binding
  - Property binding `[ ... ]`
  - Interpolation `{{ ... }}`
  - Event binding `( ... )`
- Two way binding `[( ... )]`

Angular

Le fichier `app.ts`

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  title = 'angular-standalone';
}
```

Le fichier `app.html`

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  ...
</div>
```

Angular

Le fichier `app.ts`

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  title = 'angular-standalone';
}
```

Le fichier `app.html`

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  ...
</div>
```


{{ Interpolation }}

`Welcome to {{ title }}!`:
`title` sera remplacé dans le template par la valeur de la variable `title` définie dans `app.ts`

Angular

Objectif : afficher les attributs et les valeurs sous forme d'une liste HTML

- Créer un répertoire `classes` dans `app` où toutes les classes seront stockées.
- Créer une classe `Stagiaire` avec les attributs `num`, `nom` et `prénom`.
- Créer un objet de cette classe dans `app.ts`
- Afficher les valeurs de cet objet sous forme de liste dans `app.html`

On peut aussi utiliser la commande

```
ng generate class classes/stagiaire
```

On peut aussi utiliser la commande

```
ng generate class classes/stagiaire
```

Pour générer une classe sans le fichier de test (`.spec.ts`)

```
ng generate class classes/stagiaire -skip-tests=true
```

On peut aussi utiliser la commande

```
ng generate class classes/stagiaire
```

Pour générer une classe sans le fichier de test (`.spec.ts`)

```
ng generate class classes/stagiaire -skip-tests=true
```

Ou

```
ng generate class classes/stagiaire -skip-tests
```

On peut aussi utiliser la commande

```
ng generate class classes/stagiaire
```

Pour générer une classe sans le fichier de test (`.spec.ts`)

```
ng generate class classes/stagiaire -skip-tests=true
```

Ou

```
ng generate class classes/stagiaire -skip-tests
```

Remarque

`-skip-tests` pourrait être utilisée avec `generate` quel que soit l'élément à générer.

Contenu de `stagiaire.ts`

```
export default class Stagiaire {
    constructor(private _num?: number, private _nom?: string, private
       _prenom?: string) { }

    get num(): number | undefined {
        return this._num;
    }
    set num(_num: number | undefined) {
        this._num = _num;
    }
    get nom(): string | undefined {
        return this._nom;
    }
    set nom(_nom: string | undefined) {
        this._nom = _nom;
    }
    get prenom(): string | undefined {
        return this._prenom;
    }
    set prenom(_prenom: string | undefined) {
        this._prenom = _prenom;
    }
}
```

Angular

Créons un objet de type `Stagiaire` dans `app.ts`

```
import { Component } from '@angular/core';
import Stagiaire from './classes/stagiaire';

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {

  title = 'angular-standalone';
  stagiaire: Stagiaire = new Stagiaire(100, 'Wick', 'John');

}
```

Angular

Affichons l'objet `stagiaire` **dans** `app.html`

```
<ul>
  <li> Nom : {{ stagiaire.nom }} </li>
  <li> Prénom : {{ stagiaire.prenom }} </li>
</ul>
```

Angular

L'écriture suivante

```
{{ stagiaire }}
```

Angular

L'écriture suivante

```
{{ stagiaire }}
```

affiche

```
[object Object]
```

Angular

On peut utiliser un pipe pour afficher un objet au format JSON

```
{{ stagiaire | json }}
```

Angular

On peut utiliser un pipe pour afficher un objet au format JSON

```
{{ stagiaire | json }}
```

Pour cela, il faut ajouter `CommonModule`

```
@Component({
  selector: 'app-root',
  imports: [CommonModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
```

Angular

On peut utiliser un pipe pour afficher un objet au format JSON

```
{{ stagiaire | json }}
```

Pour cela, il faut ajouter `CommonModule`

```
@Component({
  selector: 'app-root',
  imports: [CommonModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
```

Le résultat est

```
{ "_num": 100, "_nom": "Wick", "_prenom": "John" }
```

Angular

Modifions l'objet `stagiaire` **dans** `app.ts`

```
import { Component } from '@angular/core';
import Stagiaire from './classes/stagiaire';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-root',
  imports: [CommonModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {

  title = 'angular-standalone';
  stagiaire: Stagiaire = new Stagiaire(100, 'Wick');

}
```

Angular

Dans `app.html`, et depuis Angular 12, on peut utiliser La coalescence nulle

```
<ul>
  <li> Nom : {{ stagiaire.nom }} </li>
  <li> Prénom : {{ stagiaire.prenom ?? "prénom indéfini" }} </li>
</ul>
```

Angular

Créer un tableau d'entiers `tab` **dans** `app.ts`

```
@Component({
  selector: 'app-root',
  imports: [CommonModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {

  title = 'angular-standalone';
  stagiaire: Stagiaire = new Stagiaire(100, 'Wick');
  tab: number[] = [2, 3, 5, 8, 1];

}
```

Afficher le tableau `tab` dans `app.html`

```
<ul>
  <li> {{ tab[0] }} </li>
  <li> {{ tab['1'] }} </li>
  <li> {{ tab["2"] }} </li>
</ul>
```

Afficher le tableau `tab` dans `app.html`

```
<ul>
  <li> {{ tab[0] }} </li>
  <li> {{ tab['1'] }} </li>
  <li> {{ tab["2"] }} </li>
</ul>
```


Remarques

- Ce code est trop répétitif
- Et si on ne connaissait pas le nombre d'éléments
- Ou si on ne voulait pas afficher tous les éléments

Afficher le tableau `tab` dans `app.html`

```
<ul>
  <li> {{ tab[0] }} </li>
  <li> {{ tab['1'] }} </li>
  <li> {{ tab["2"] }} </li>
</ul>
```


Remarques

- Ce code est trop répétitif
- Et si on ne connaissait pas le nombre d'éléments
- Ou si on ne voulait pas afficher tous les éléments

Solution

utiliser les directives (section suivante)

Angular

Ajouter une méthode `direBonjour()` **dans** `app.ts`

```
import { Component } from '@angular/core';
import Stagiaire from './classes/stagiaire';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-root',
  imports: [CommonModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {

  title = 'angular-standalone';
  stagiaire: Stagiaire = new Stagiaire(100, 'Wick');
  tab: number[] = [2, 3, 5, 8, 1];

  direBonjour(): string {
    return 'Bonjour Angular';
  }

}
```

Angular

Appeler la méthode `direBonjour()` **dans** `app.html`

```
<p> {{ direBonjour() }} </p>
```

Angular

Pour afficher le contenu de l'attribut `title` dans le template, on peut utiliser l'interpolation

```
<p> {{ title }} </p>
```

Angular

Pour afficher le contenu de l'attribut `title` dans le template, on peut utiliser l'interpolation

```
<p> {{ title }} </p>
```

On peut aussi utiliser le Property binding sur la propriété JavaScript `textContent`

```
<p [textContent]=title></p>
```

Angular

Pour afficher le contenu de l'attribut `title` dans le template, on peut utiliser l'interpolation

```
<p> {{ title }} </p>
```

On peut aussi utiliser le Property binding sur la propriété JavaScript `textContent`

```
<p [textContent]=title></p>
```


[property binding]

- `[property]='value'` : permet de remplacer `value` par sa valeur dans `app.ts`
- `[property]='expression'` : permet d'évaluer l'expression dans la classe et affecte son résultat à la propriété correspondante.

Angular

Évènement (Event)

- Action appliquée par l'utilisateur ou simulée par le développeur sur un `component.html`.
- Évènement déclenché ⇒ Méthode, dans le `component.ts` associé, exécutée.

Angular

Évènement (Event)

- Action appliquée par l'utilisateur ou simulée par le développeur sur un `component.html`.
- Évènement déclenché ⇒ Méthode, dans le `component.ts` associé, exécutée.

Event binding (...)

- Objectif : associer un évènement à une méthode de la classe associée au template.
- Pas besoin de préfixer l'évènement par `'on'` comme en **JavaScript**.

Angular

Ajoutons la méthode `alertBonjour()` dans `app.ts`

```
import { Component } from '@angular/core';
import Stagiaire from './classes/stagiaire';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-root',
  imports: [CommonModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {

  title = 'angular-standalone';
  stagiaire: Stagiaire = new Stagiaire(100, 'Wick');
  tab: number[] = [2, 3, 5, 8, 1];

  direBonjour(): string {
    return 'Bonjour Angular';
  }
  alertBonjour(): void {
    alert('Bonjour Angular');
  }
```

Angular

Pour exécuter la méthode `direBonjour()` lorsqu'on clique sur un bouton

```
<button (click)="alertBonjour()">
    Dire Bonjour
</button>
```

Angular

Exercice

- Créer deux boutons **HTML** dont l'un est initialement désactivé (`disabled`).
- À chaque clic sur le bouton activé, celui-ci se désactive tandis que l'autre s'active.

Solution

```
<div>
  <button [disabled]="disabled" (click)="switchDisabled()">
    bouton 1
  </button>

  <button [disabled]="!disabled" (click)="switchDisabled()">
    bouton 2
  </button>
</div>
```

Solution

```
<div>
  <button [disabled]="disabled" (click)="switchDisabled()">
    bouton 1
  </button>

  <button [disabled]="!disabled" (click)="switchDisabled()">
    bouton 2
  </button>
</div>
```

Sans oublier de déclarer `disabled`

```
disabled = true
```

Solution

```
<div>
  <button [disabled]="disabled" (click)="switchDisabled()">
    bouton 1
  </button>

  <button [disabled]="!disabled" (click)="switchDisabled()">
    bouton 2
  </button>
</div>
```

Sans oublier de déclarer `disabled`

```
disabled = true
```

Et la méthode `switchDisabled()`

```
switchDisabled() {
  this.disabled = !this.disabled
}
```

Angular

Considérons la zone de saisie suivante dans laquelle nous ne voudrions accepter que des lettres en minuscule

```
<div>
  <label for="texte"> Merci de bien saisir un texte :
  </label>
  <input type="text" id="texte" (input)="showValue($event)">
</div>
```

Angular

Considérons la zone de saisie suivante dans laquelle nous ne voudrions accepter que des lettres en minuscule

```
<div>
  <label for="texte"> Merci de bien saisir un texte :
  </label>
  <input type="text" id="texte" (input)="showValue($event)">
</div>
```

Pour connaître le caractère saisi, utilisons l'objet `event` dans `showValue()`

```
showValue(event: Event) {
  console.log((event.target as HTMLInputElement).value);
  // affiche tous les caractères saisis

  console.log((event as InputEvent).data);
  // affiche le dernier caractère saisi
}
```

Angular

Structures de contrôle : deux approches

- **Control Flow Syntax** (depuis Angular 17)
- **CommonModule** (depuis Angular 2)

Angular

Deux approches des structures de contrôle

Ancienne approche	**Nouvelle approche**
CommonModule depuis Angular 2	**Control Flow Syntax** depuis Angular 17
Utilise `*ngIf`, `*ngFor`...	Utilise `@if`, `@for`...
Basée sur des directives	Syntaxe intégrée, plus lisible

Angular

Control Flow Syntax

- Introduit dans **Angular 17**
- Ayant comme objectif de remplacer les directives structurelles
  - `@for` pour remplacer `*ngFor`
  - `@if` pour remplacer `*ngIf`
  - `@switch` pour remplacer `*ngSwitch`

Angular

Pour essayer Control Flow Syntax dans un projet existant, il faut utiliser la migration suivante

```
ng generate @angular/core:control-flow
```

Angular

Afficher les éléments du tableau `tab` en utilisant `*ngFor`

```
<ul>
    @for (elt of tab; track elt) {
    <li> {{ elt }} </li>
    }
</ul>
```

Angular

Afficher les éléments du tableau `tab` en utilisant `*ngFor`

```
<ul>
    @for (elt of tab; track elt) {
    <li> {{ elt }} </li>
    }
</ul>
```

`track elt`

- permet de spécifier la clé utilisée pour associer les éléments du tableau aux vues dans le **DOM**.
- permet à **Angular** d'exécuter un ensemble minimal d'opérations sur le **DOM** lorsque des éléments sont ajoutés, supprimés ou déplacés dans une collection.

Angular

Et pour avoir l'indice de l'itération courante, on peut utiliser la variable contextuelle `$index`

```
<ul>
    @for (elt of tab; track elt) {
    <li>{{ $index }} {{ elt }} </li>
    }
</ul>
```

Angular

Et pour avoir l'indice de l'itération courante, on peut utiliser la variable contextuelle `$index`

```
<ul>
    @for (elt of tab; track elt) {
    <li>{{ $index }} {{ elt }} </li>
    }
</ul>
```

ou

```
<ul>
    @for (elt of tab; let i = $index; track i) {
    <li>{{ i }} {{ elt }} </li>
    }
</ul>
```

Angular

Autres variables contextuelles

- `$first` : retourne `true` si l'élément courant est le premier de la liste, `false` sinon.
- `$last` : retourne `true` si l'élément courant est le dernier de la liste, `false` sinon.
- `$even` : retourne `true` si l'indice de l'élément courant est pair, `false` sinon.
- `$odd` : retourne `true` si l'indice de l'élément courant est impair, `false` sinon.
- `$count` : retourne le nombre total d'éléments dans la collection.

Angular

Exemple

```
<ul>
    @for (item of numbers; let i = $index; track i) {
    <li>
        {{ i }} : {{ item }} : {{ $first }} : {{ $last }} : {{ $even }}
            : {{ $odd }}
    </li>
    }
</ul>
```

Angular

Le bloc optionnel `@empty` est exécuté lorsque la liste est vide

```
<ul>
    @for (elt of tab; track i) {
    <li>{{ elt }} </li>
    } @empty {
        La liste ne contient aucun élément.
    }
</ul>
```

Angular

Considérons la liste suivante (à déclarer dans `app.ts`)

```
stagiaires: Array<Stagiaire> = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven')
];
```

Angular

Considérons la liste suivante (à déclarer dans `app.ts`)

```
stagiaires: Array<Stagiaire> = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven')
];
```


Exercice

Écrire un script **Angular** qui permet d'afficher dans une liste **HTML** les nom et prénom de chaque stagiaire de la liste `stagiaires`.

Angular

Solution

```
<ul>
    @for (st of stagiaires; track st.num) {
    <li> {{ st.prenom }} {{ st.nom }} </li>
    }
</ul>
```

Angular

Remarques

- `@for` nous permet d'itérer sur les tableaux, mais pas sur les objets.
- Il est possible d'itérer sur les objets en utilisant le pipe `keyvalue`.

Angular

Exemple d'utilisation de `keyvalue` dans des boucles imbriquées

```
<ul>
    @for (st of stagiaires; track st.num) {
        @for (elt of st | keyvalue ; track elt) {
        <li> {{ elt.key }} {{ elt.value }}</li>
        }
    }
</ul>
```

Angular

Exemple avec `@if`

```
<ul>
  @if(tab[1] % 2 != 0) {
  <li>
    {{ tab[1] }} est impair
  </li>
  }
</ul>
```

Angular

Exemple avec `@if`

```
<ul>
  @if(tab[1] % 2 != 0) {
  <li>
    {{ tab[1] }} est impair
  </li>
  }
</ul>
```


Attention, avec ce code, si la condition est fausse, un `<li>` vide sera quand-même attaché au DOM

```
<ul>
  <li>
    @if(tab[1] % 2 != 0) {
    {{ tab[1] }} est impair
    }
  </li>
</ul>
```

Angular

Exemple avec `@else`

```
<ul>
  <li>
    @if(tab[0] % 2 != 0) {
    {{ tab[0] }} est impair
    } @else {
    {{ tab[0] }} est pair
    }
  </li>
</ul>
```

Angular

Exercice 1

Utiliser les Control Flow Syntax d'**Angular** pour afficher sous forme d'une liste **HTML** les éléments du tableau `tab` ainsi que leurs parités.

Angular

Exercice 2

Utiliser les Control Flow Syntax d'**Angular** pour afficher sous forme d'une liste **HTML** les éléments du tableau `moyennes: number[] = [18, 5, 11, 15]` avec le message suivant :

- Si $0 \leq$ valeur < 10 alors `échec`,
- Si $10 \leq$ valeur < 13 alors `moyen`,
- Si $13 \leq$ valeur < 16 alors `bien`,
- Sinon `très bien`.

Angular

Exemple avec `@switch`

```
<ul>
    @for(elt of tab; track $index) {
    <li>
        @switch(elt) {
            @case(1) {
                {{ elt }} = un
            }
            @case(2) {
                {{ elt }} = deux
            }
            @case(3) {
                {{ elt }} = trois
            }
            @default {
                {{ elt }} = autre
            }
        }
    </li>
    }
</ul>
```

Angular

Refaire cet exercice avec `@switch`

Afficher sous forme d'une liste les éléments du tableau `moyennes = [18, 5, 11, 15]` avec le message suivant :

- Si $0 \leq$ valeur < 10 alors `échec`
- Si $10 \leq$ valeur < 13 alors `moyen`
- Si $13 \leq$ valeur < 16 alors `bien`
- Sinon `très bien`

Angular

Plusieurs directives possibles

- `*ngFor`
- `*ngIf`
- `*ngSwitch`
- `ngStyle`
- `ngClass`

Angular

Plusieurs directives possibles

- `*ngFor`
- `*ngIf`
- `*ngSwitch`
- `ngStyle`
- `ngClass`

Ces directives s'utilisent conjointement avec les composants web suivant

- `ng-container`
- `ng-template`

Angular

*ngFor

- permet de répéter un traitement (affichage d'un élément **HTML**)
- s'utilise comme un attribut de balise et sa valeur est une instruction itérative **TypeScript**

Angular

Afficher les éléments du tableau `tab` en utilisant `*ngFor`

```
<ul>
  <li *ngFor="let elt of tab">
    {{ elt }}
  </li>
</ul>
```

Angular

Afficher les éléments du tableau `tab` en utilisant `*ngFor`

```
<ul>
  <li *ngFor="let elt of tab">
    {{ elt }}
  </li>
</ul>
```

Le code précédent est une forme simplifiée du code suivant

```
<ul>
  <ng-template ngFor let-elt [ngForOf]="tab">
    <li>{{ elt }} </li>
  </ng-template>
</ul>
```

Angular

Afficher les éléments du tableau `tab` en utilisant `*ngFor`

```
<ul>
  <li *ngFor="let elt of tab">
    {{ elt }}
  </li>
</ul>
```

Le code précédent est une forme simplifiée du code suivant

```
<ul>
  <ng-template ngFor let-elt [ngForOf]="tab">
    <li>{{ elt }} </li>
  </ng-template>
</ul>
```

`ng-template` n'apparaîtra pas dans le **DOM**.

Et pour avoir l'indice de l'itération courante

```
<ul>
  <li *ngFor="let elt of tab; let i = index">
    {{ i }} : {{ elt }}
  </li>
</ul>
```

Et pour avoir l'indice de l'itération courante

```
<ul>
  <li *ngFor="let elt of tab; let i = index">
    {{ i }} : {{ elt }}
  </li>
</ul>
```

ou

```
<ul>
  <li *ngFor="let elt of tab; index as i">
    {{ i }} : {{ elt }}
  </li>
</ul>
```

Et pour avoir l'indice de l'itération courante

```
<ul>
  <li *ngFor="let elt of tab; let i = index">
    {{ i }} : {{ elt }}
  </li>
</ul>
```

ou

```
<ul>
  <li *ngFor="let elt of tab; index as i">
    {{ i }} : {{ elt }}
  </li>
</ul>
```

Ou la version la plus longue

```
<ul>
  <ng-template ngFor let-i="index" let-elt [ngForOf]="tab">
    <li> {{ i }} : {{ elt }} </li>
  </ng-template>
</ul>
```

Angular

On peut aussi utiliser `first` pour savoir si l'élément courant est le premier de la liste

```
<ul>
  <li *ngFor="let elt of tab; let i = index; let isFirst = first">
    {{ i }} : {{ elt }} : {{ isFirst }}
  </li>
</ul>
```

Angular

On peut aussi utiliser `first` pour savoir si l'élément courant est le premier de la liste

```
<ul>
  <li *ngFor="let elt of tab; let i = index; let isFirst = first">
    {{ i }} : {{ elt }} : {{ isFirst }}
  </li>
</ul>
```


Autres paramètres possible

- `last` : retourne `true` si l'élément courant est le dernier de la liste, `false` sinon.
- `even` : retourne `true` si l'indice de l'élément courant est pair, `false` sinon.
- `odd` : retourne `true` si l'indice de l'élément courant est impair, `false` sinon.

Angular

Considérons la liste suivante (à déclarer dans `app.ts`)

```
stagiaires: Array<Stagiaire> = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven')
];
```

Angular

Considérons la liste suivante (à déclarer dans `app.ts`)

```
stagiaires: Array<Stagiaire> = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven')
];
```


Exercice

Écrire un script **Angular** qui permet d'afficher dans une liste **HTML** les nom et prénom de chaque stagiaire de la liste `stagiaires`.

Angular

Solution

```
<ul>
  <li *ngFor="let st of stagiaires">
    {{ st.prenom }} : {{ st.nom }}
  </li>
</ul>
```

Angular

Considérons la méthode suivante (à déclarer dans `app.ts`)

```
ajouterStagiaires() {
  this.stagiaires = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven'),
    new Stagiaire(104, 'Dalton', 'Jack'),
    new Stagiaire(105, 'Maggio', 'Carol')
  ];
}
```

Angular

Considérons la méthode suivante (à déclarer dans `app.ts`)

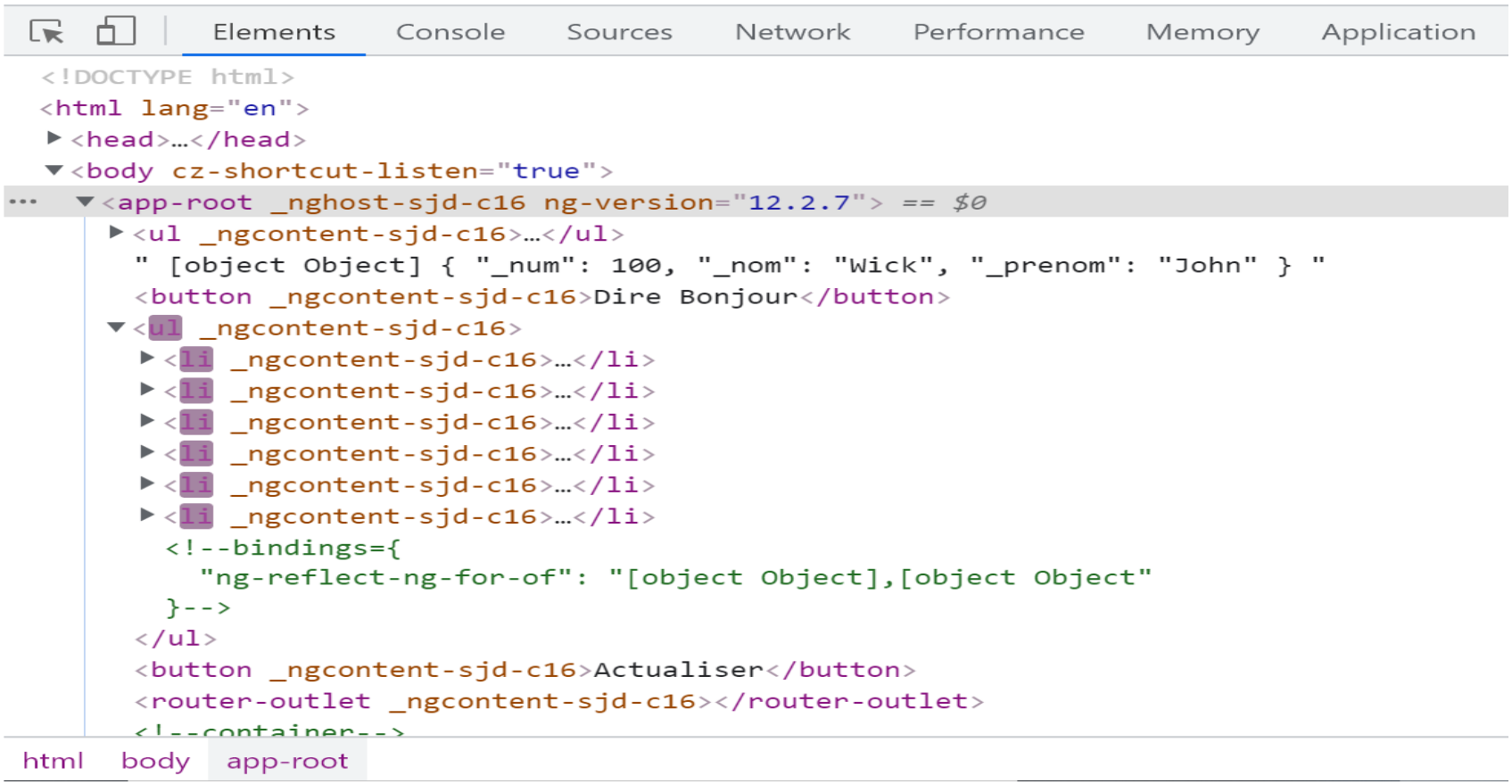
```
ajouterStagiaires() {
  this.stagiaires = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven'),
    new Stagiaire(104, 'Dalton', 'Jack'),
    new Stagiaire(105, 'Maggio', 'Carol')
  ];
}
```


Et le bouton suivant permettant de mettre à jour la liste de stagiaires

```
<ul>
  <li *ngFor="let st of stagiaires">
    {{ st.prenom }} : {{ st.nom }}
  </li>
</ul>
<button (click)="ajouterStagiaires()">Actualiser</button>
```

Angular

En cliquant sur le bouton, la liste (et la partie du DOM associée) sera entièrement reconstruite



Angular

Question

Comment faire pour ajouter seulement les éléments manquants sans reconstruire toute la liste ?

Angular

Question

Comment faire pour ajouter seulement les éléments manquants sans reconstruire toute la liste ?

Réponse

Utiliser `trackBy` dans `*ngFor` qui permet de suivre les éléments lorsqu'ils sont ajoutés ou supprimés du tableau pour des meilleures performances.

Angular

Nous devons définir une méthode `trackBy` qui identifie chaque élément de manière unique

```
trackByStagiaireNum(index: number, stagiaire: Stagiaire) {
  return stagiaire.num;
}
```

Angular

Nous devons définir une méthode `trackBy` qui identifie chaque élément de manière unique

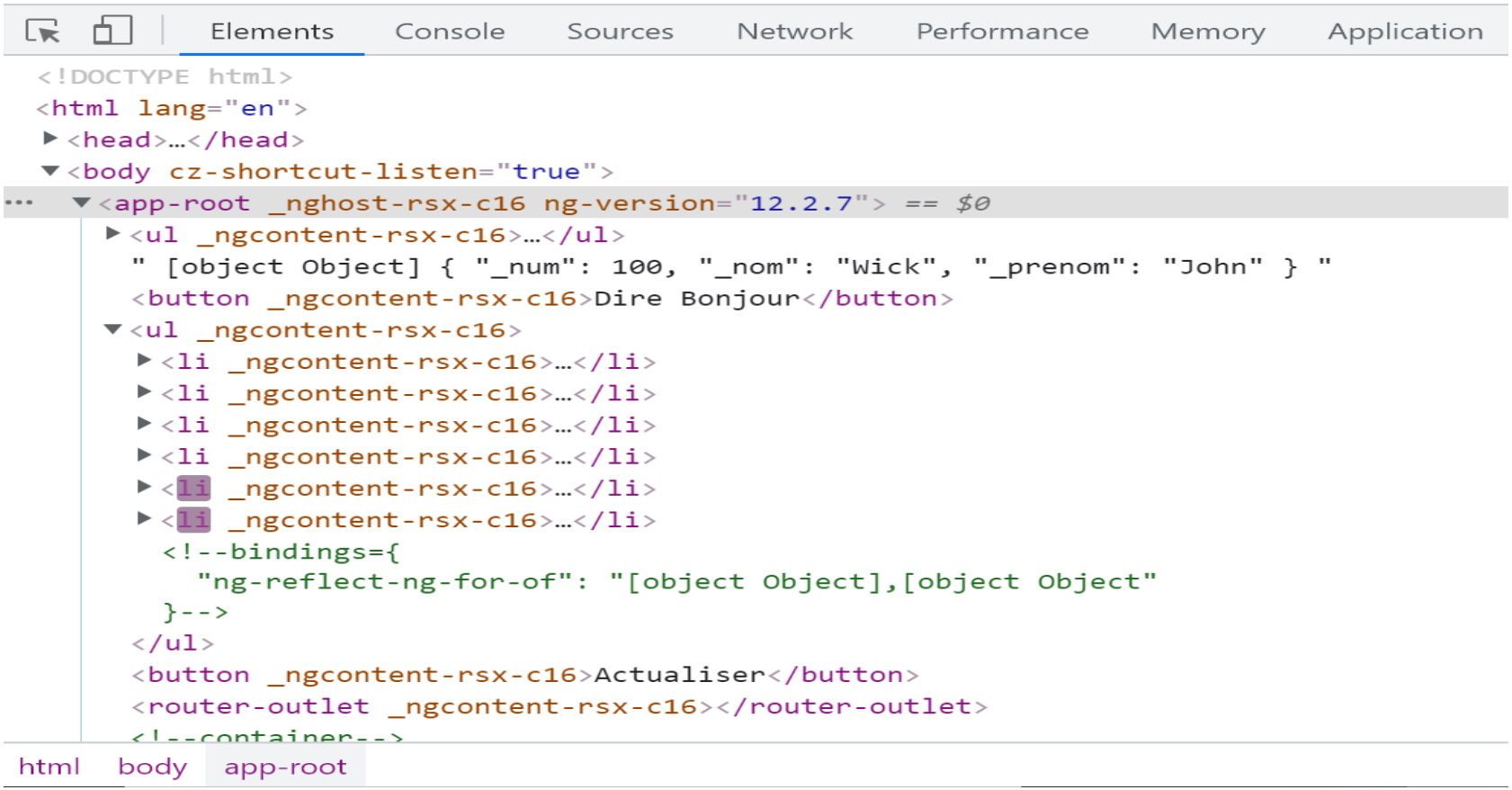
```
trackByStagiaireNum(index: number, stagiaire: Stagiaire) {
  return stagiaire.num;
}
```

Associons cette méthode à la directive `*ngFor`

```
<ul>
  <li *ngFor="let st of stagiaires; trackBy:trackByStagiaireNum">
    {{ st.prenom }} : {{ st.nom }}
  </li>
</ul>
<button (click)="ajouterStagiaires()">Actualiser</button>
```

Angular

En cliquant sur le bouton, les deux derniers éléments ont été ajoutés au DOM sans reconstruire intégralement la liste



Angular

Ou la version la plus longue

```
<ul>
  <ng-template ngFor let-st [ngForOf]="stagiaires" [ngForTrackBy]="
     trackByStagiaireNum">
    <li>
      {{ st.prenom }} : {{ st.nom }}
    </li>
  </ng-template>
</ul>
<button (click)="ajouterStagiaires()">Actualiser</button>
```

Angular

Remarques

- `*ngFor` nous permet d'itérer sur les tableaux, mais pas sur les objets.
- Il est possible d'itérer sur les objets en utilisant le pipe `keyvalue`.

Angular

Pour tester puis afficher si le deuxième élément du tableau est impair

```
<ul>
  <li *ngIf="tab[1] % 2 != 0">
    {{ tab[1] }} est impair
  </li>
</ul>
```

Angular

Pour tester puis afficher si le deuxième élément du tableau est impair

```
<ul>
  <li *ngIf="tab[1] % 2 != 0">
    {{ tab[1] }} est impair
  </li>
</ul>
```

Ou en plus long

```
<ul>
  <ng-template [ngIf]="tab[1] % 2 != 0">
    <li>
      {{ tab[1] }} est impair
    </li>
  </ng-template>
</ul>
```

Angular

Exercice

Utiliser les directives **Angular** pour afficher le premier élément du tableau (`tab`) ainsi que sa parité (pair ou impair).

Une première solution avec `*ngIf` et `else`

```
<ul>
  <li *ngIf="tab[0] % 2 != 0; else pair">
    {{ tab[0] }} est impair
  </li>
  <ng-template #pair>
     <li>
       {{ tab[0] }} est pair
     </li>
  </ng-template>
</ul>
```

Une première solution avec `*ngIf` et `else`

```
<ul>
  <li *ngIf="tab[0] % 2 != 0; else pair">
    {{ tab[0] }} est impair
  </li>
  <ng-template #pair>
     <li>
       {{ tab[0] }} est pair
     </li>
  </ng-template>
</ul>
```

Ou en plus long

```
<ul>
  <ng-template [ngIf]="tab[0] % 2 != 0" [ngIfElse]=pair>
    <li>
      {{ tab[0] }} est impair
    </li>
  </ng-template>
  <ng-template #pair>
    <li>
      {{ tab[0] }} est pair
    </li>
  </ng-template>
</ul>
```

Une deuxième solution avec `*ngIf`, `then` **et** `else`

```
<ul>
  <li *ngIf="tab[0] % 2 != 0; then impair else pair">
    Ce code ne sera jamais pris en compte
  </li>
  <ng-template #impair>
    <li>
      {{ tab[0] }} est impair
    </li>
  </ng-template>
  <ng-template #pair>
    <li>
      {{ tab[0] }} est pair
    </li>
  </ng-template>
</ul>
```

Une deuxième solution avec `*ngIf`**,** `then` **et** `else`

```
<ul>
  <li *ngIf="tab[0] % 2 != 0; then impair else pair">
    Ce code ne sera jamais pris en compte
  </li>
  <ng-template #impair>
    <li>
      {{ tab[0] }} est impair
    </li>
  </ng-template>
  <ng-template #pair>
    <li>
      {{ tab[0] }} est pair
    </li>
  </ng-template>
</ul>
```

Ou en plus long

```
<ul>
  <ng-template [ngIf]="tab[0] % 2 != 0" [ngIfThen]=impair [ngIfElse]=pair></ng-template>
  <ng-template #impair>
    <li>
      {{ tab[0] }} est impair
    </li>
  </ng-template>
  <ng-template #pair>
    <li>
      {{ tab[0] }} est pair
    </li>
  </ng-template>
</ul>
```

Angular

Remarque

`*ngIf` et `*ngFor` ne doivent pas être utilisées dans la même balise.

Angular

Exercice

Utiliser les directives **Angular** pour afficher sous forme d'une liste **HTML** tous les éléments du tableau précédent (`tab`) ainsi que leur parité.

Angular

Solution

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <li *ngIf="elt % 2 != 0; else sinon">
      {{ elt }} est impair
    </li>
    <ng-template #sinon>
      <li>
        {{ elt }} est pair
      </li>
    </ng-template>
  </ng-container>
</ul>
```

Angular

Solution

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <li *ngIf="elt % 2 != 0; else sinon">
      {{ elt }} est impair
    </li>
    <ng-template #sinon>
      <li>
        {{ elt }} est pair
      </li>
    </ng-template>
  </ng-container>
</ul>
```

`ng-container` n'apparaît pas dans le **DOM**.

Angular

Exercice

Utiliser les directives **Angular** pour afficher sous forme d'une liste **HTML** les éléments du tableau `moyennes: number[] = [18, 5, 11, 15]` avec le message suivant :

- Si $0 \leq$ valeur < 10 alors `échec`,
- Si $10 \leq$ valeur < 13 alors `moyen`,
- Si $13 \leq$ valeur < 16 alors `bien`,
- Sinon `très bien`.

Angular

Une solution possible

```
<ul>
  <ng-container *ngFor="let elt of moyennes">
    <li *ngIf="elt >= 0 && elt < 10; else moyen">
      {{ elt }} : échec
    </li>
    <ng-template #moyen>
      <li *ngIf="elt < 13; else bien">
        {{ elt }} : moyen
      </li>
    </ng-template>
    <ng-template #bien>
      <li *ngIf="elt < 16; else tbien">
        {{ elt }} : bien
      </li>
    </ng-template>
    <ng-template #tbien>
      <li *ngIf="elt >= 16; else autre">
        {{ elt }} : très bien
      </li>
    </ng-template>
    <ng-template #autre>
      <li>
        autre
      </li>
    </ng-template>
  </ng-container>
</ul>
```

Angular

*ngIf vs hidden

- *ngIf commentera le contenu si la condition est fausse. L'élément ne sera pas attaché au **DOM**.
- [hidden] attachera l'élément au **DOM** et le marquera avec l'attribut hidden.
- hidden ne supprime pas l'élément du **DOM** mais le rend invisible, tandis que *ngIf retire complètement l'élément du **DOM**.

Angular

ngSwitch

- Équivalent à `switch` en **JavaScript**.
- Acceptant des valeurs de type `number`, `string` et `boolean`.

Exemple avec ngSwitch

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <ng-container [ngSwitch]="elt">
      <li *ngSwitchCase="1">
        {{ elt }} = un
      </li>
      <li *ngSwitchCase="2">
        {{ elt }} = deux
      </li>
      <li *ngSwitchCase="3">
        {{ elt }} = trois
      </li>
      <li *ngSwitchDefault>
        {{ elt }} : autre
      </li>
    </ng-container>
  </ng-container>
</ul>
```

Exemple avec `ngSwitch`

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <ng-container [ngSwitch]="elt">
      <li *ngSwitchCase="1">
        {{ elt }} = un
      </li>
      <li *ngSwitchCase="2">
        {{ elt }} = deux
      </li>
      <li *ngSwitchCase="3">
        {{ elt }} = trois
      </li>
      <li *ngSwitchDefault>
        {{ elt }} : autre
      </li>
    </ng-container>
  </ng-container>
</ul>
```

Pour comparer avec une chaîne de caractère, il faut ajouter les simples quotes (par exemple : `*ngSwitchCase="'bonjour'"`.

Angular

Refaire cet exercice avec `ngSwitch`

Utiliser les directives **Angular** pour afficher sous forme d'une liste les éléments du tableau `moyennes = [18, 5, 11, 15]` avec le message suivant :

- Si $0 \leq$ valeur $<$ 10 alors `échec`
- Si $10 \leq$ valeur $<$ 13 alors `moyen`
- Si $13 \leq$ valeur $<$ 16 alors `bien`
- Sinon `très bien`

Angular

Solution

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <ng-container [ngSwitch]="true">
      <li *ngSwitchCase="elt >= 0 && elt < 10">
        {{ elt }} : échec
      </li>
      <li *ngSwitchCase="elt >= 10 && elt < 13">
        {{ elt }} : moyen
      </li>
      <li *ngSwitchCase="elt >= 13 && elt < 16">
        {{ elt }} : bien
      </li>
      <li *ngSwitchCase="elt >= 16">
        {{ elt }} : très bien
      </li>
      <li *ngSwitchDefault>
        autre
      </li>
    </ng-container>
  </ng-container>
</ul>
```

Contrairement au `*ngIf`, plusieurs `*ngSwitchCase` peuvent être exécutée pour une même valeur. Par conséquence, le code suivant ne réalise pas le travail demandé

```
<ul>
  <ng-container *ngFor="let elt of moyennes">
    <ng-container [ngSwitch]="true">
      <li *ngSwitchCase="elt >= 0 && elt < 10">
        {{ elt }} : échec
      </li>
      <li *ngSwitchCase="elt < 13">
        {{ elt }} : moyen
      </li>
      <li *ngSwitchCase="elt < 16">
        {{ elt }} : bien
      </li>
      <li *ngSwitchCase="elt >= 16">
        {{ elt }} : très bien
      </li>
      <li *ngSwitchDefault>
        autre
      </li>
    </ng-container>
  </ng-container>
</ul>
```

Angular

Remarques

- L'objectif de l'exercice précédent était une excellente occasion de se familiariser avec les directives **Angular** telles que `*ngSwitchCase` et `*ngIf`.
- Ceci est essentiel pour comprendre et maîtriser la création d'interfaces utilisateur dynamiques dans **Angular**.
- Quand c'est possible, on préfère déplacer la logique de calcul dans les fichiers **TypeScript** (`*.component.ts`).
- Cela réduit considérablement la complexité du **HTML** et rend le code plus lisible et plus facile à maintenir.

Angular

ngPlural

- Introduit dans **Angular** 10.
- Similaire à `ngSwitch` mais n'acceptant que des valeurs numériques.
- Pas de mot-clé pour le cas par défaut, mais on peut utiliser la constante `other`.

Angular

Exemple avec `ngPlural`

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <li>
      <ng-container [ngPlural]="elt">
        <ng-template ngPluralCase="1">
          {{ elt }} : un
        </ng-template>
        <ng-template ngPluralCase="2">
          {{ elt }} : deux
        </ng-template>
        <ng-template ngPluralCase="3">
          {{ elt }} : trois
        </ng-template>
        <ng-template ngPluralCase="other">
          {{ elt }} : autre
        </ng-template>
      </ng-container>
    </li>
  </ng-container>
</ul>
```

Angular

ngStyle

- permet de modifier dynamiquement le style d'un élément **HTML**.
- permet de récupérer des valeurs définies dans la partie `script` ou `style`.

Angular

ngStyle

- permet de modifier dynamiquement le style d'un élément **HTML**.
- permet de récupérer des valeurs définies dans la partie `script` ou `style`.

Remarque

N'utilisez jamais `ngStyle` dans une balise `<ng-template>` ou `<ng-container>` car ces dernières ne sont pas directement présentes dans le **DOM**.

Angular

Ajoutons les deux attributs suivants dans `app.ts`

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrls: ['./app.css']
})
export class App {

  // le contenu précédent

  couleur = 'white';
  couleurBg = 'red';
}
```

Angular

Pour attribuer une valeur à la propriété CSS d'une balise

```
<p [style.background-color]='couleurBg'>
  {{ title }}
</p>
```

Angular

Pour attribuer une valeur à la propriété CSS d'une balise

```
<p [style.background-color]='couleurBg'>
  {{ title }}
</p>
```

Pour deux propriétés

```
<p [style.background-color]='couleurBg' [style.color]='couleur'>
  {{ title }}
</p>
```

Angular

Pour attribuer une valeur à la propriété CSS d'une balise

```
<p [style.background-color]='couleurBg'>
  {{ title }}
</p>
```

Pour deux propriétés

```
<p [style.background-color]='couleurBg' [style.color]='couleur'>
  {{ title }}
</p>
```


Remarque

L'écriture précédente peut-être simplifiée en utilisant la directive `ngStyle`.

Angular

ngStyle

- permet de modifier le style d'un élément **HTML**.
- s'utilise conjointement avec le **property binding** pour récupérer des valeurs définies dans la classe.

Angular

Simplification avec `ngStyle`

```
<p [ngStyle]="{ color: couleur, backgroundColor: couleurBg }">
  {{ title }}
</p>
```

Angular

`ngStyle` **peut aussi prendre comme valeur un objet**

```
<p [ngStyle]="monStyle">
    {{ title }}
</p>
```

Angular

`ngStyle` **peut aussi prendre comme valeur un objet**

```
<p [ngStyle]="monStyle">
    {{ title }}
</p>
```

Contenu de `monStyle` **dans** `.component.ts`

```
monStyle = { color: 'white', backgroundColor: 'skyblue' }
```

Angular

Il est possible de définir des méthodes dans `app.ts` qui gèrent le style d'un élément HTML

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrls: ['./app.css']
})
export class App {
  // le contenu précédent

  getColor(): string {
    return 'white';
  }
  getBgColor(): string {
    return 'red';
  }
}
```

Angular

Pour afficher le contenu de l'attribut `nom` dans le template avec une couleur de fond rouge

```
<p [ngStyle]="{ color: getColor(), bgColor: getBgColor() }">
  {{ title }}
</p>
```

Angular

Exercice

Afficher sous forme d'une liste **HTML** le premier élément et le dernier du tableau `tab` en blanc sur un fond bleu, le reste en blanc sur fond rouge.

Angular

ngClass

- permet d'attribuer de nouvelles classes d'un élément **HTML**.
- s'utilise conjointement avec le **property binding** pour récupérer des valeurs définies dans la classe ou dans la feuille de style.

Angular

ngClass

- permet d'attribuer de nouvelles classes d'un élément **HTML**.
- s'utilise conjointement avec le **property binding** pour récupérer des valeurs définies dans la classe ou dans la feuille de style.

Remarque

N'utilisez jamais `ngClass` dans une balise `<ng-template>` ou `<ng-container>` car ces dernières ne sont pas directement présentes dans le **DOM**.

Considérons le contenu suivant de `app.ts`

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrls: ['./app.css']
})
export class App {
  // le contenu précédent
}
```

Considérons le contenu suivant de `app.ts`

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrls: ['./app.css']
})
export class App {
  // le contenu précédent
}
```

On définit deux classes rouge et bleu dans `app.css`

```
.rouge {
    color: red;
}
.bleu {
    color: blue;
}
.gras {
    font-weight: bold;
}
```

Angular

Pour associer la classe `rouge` à la balise `<p>`

```
<p [ngClass]="{ 'rouge': true }">
  {{ title }}
</p>
```

Angular

Pour associer la classe `rouge` à la balise `<p>`

```
<p [ngClass]="{ 'rouge': true }">
  {{ title }}
</p>
```

On peut aussi appeler une méthode dans la directive `ngClass`

```
<p [ngClass]="{ 'rouge': afficherEnRouge() }">
  {{ title }}
</p>
```

Angular

Pour associer la classe `rouge` à la balise `<p>`

```
<p [ngClass]="{ 'rouge': true }">
  {{ title }}
</p>
```

On peut aussi appeler une méthode dans la directive `ngClass`

```
<p [ngClass]="{ 'rouge': afficherEnRouge() }">
  {{ title }}
</p>
```

Définissons la méthode `afficherEnRouge` dans `app.ts`

```
afficherEnRouge(): boolean {
    return this.couleur === 'blue' ? true : false;
}
```

Angular

Ainsi, on peut faire aussi un affichage conditionnel

```
<p [ngClass]="{'rouge': nom == 'wick' }">
  {{ title }}
</p>
```

Angular

Ainsi, on peut faire aussi un affichage conditionnel

```
<p [ngClass]="{'rouge': nom == 'wick' }">
  {{ title }}
</p>
```

Ou encore (le paragraphe sera affiché en rouge si `nom` contient `wick`, en bleu sinon)

```
<p [ngClass]="{ 'rouge': title == 'ng', 'bleu': title != 'ng' }">
  {{ title }}
</p>
```

Angular

Ainsi, on peut faire aussi un affichage conditionnel

```
<p [ngClass]="{'rouge': nom == 'wick' }">
  {{ title }}
</p>
```

Ou encore (le paragraphe sera affiché en rouge si `nom` contient `wick`, en bleu sinon)

```
<p [ngClass]="{ 'rouge': title == 'ng', 'bleu': title != 'ng' }">
  {{ title }}
</p>
```


Plusieurs classes peuvent être associées à un même élément

```
<p [ngClass]="{ 'rouge': title == 'ng', 'bleu': title != 'ng', 'gras':
   title.length!= 2 }">
  {{ title }}
</p>
```

Angular

On peut aussi utiliser les expressions ternaires

```
<p [ngClass]="nom == 'wick' ? 'rouge' : 'bleu'">
  {{ title }}
</p>
```

Angular

Exercice

Utiliser `ngClass` pour afficher en bleu les éléments pairs du tableau précédent (`tab`) et en rouge les éléments impairs.

Angular

Première solution

```
<ul>
  @for (v of tab; track $index) {
  <li [ngClass]="v % 2 == 0 ? 'bleu' : 'rouge'">
    {{ v }}
  </li>
  }
</ul>
```

Angular

Première solution

```
<ul>
  @for (v of tab; track $index) {
  <li [ngClass]="v % 2 == 0 ? 'bleu' : 'rouge'">
    {{ v }}
  </li>
  }
</ul>
```

Deuxième solution

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <li [ngClass]="elt % 2 == 0 ? 'bleu' : 'rouge'">
      {{ elt }}
    </li>
  </ng-container>
</ul>
```

Angular

Troisième solution

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <li [ngClass]="{'rouge': !estPair(elt), 'bleu':
      estPair(elt)}">
      {{ elt }}
    </li>
  </ng-container>
</ul>
```

Angular

Troisième solution

```
<ul>
  <ng-container *ngFor="let elt of tab">
    <li [ngClass]="{'rouge': !estPair(elt), 'bleu':
      estPair(elt)}">
      {{ elt }}
    </li>
  </ng-container>
</ul>
```


Dans `app.ts`, on définit la méthode `estPair`

```
estPair(elt: number): boolean {
    return elt % 2 === 0;
}
```

Angular

Considérons la liste `stagiaires` précédente

```
stagiaires: Array<Stagiaire> = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven')
];
```

Angular

Considérons la liste `stagiaires` précédente

```
stagiaires: Array<Stagiaire> = [
    new Stagiaire(100, 'Wick', 'John'),
    new Stagiaire(101, 'Abruzzi', 'John'),
    new Stagiaire(102, 'Marley', 'Bob'),
    new Stagiaire(103, 'Segal', 'Steven')
];
```


Exercice

Écrire un script **Angular** qui permet d'afficher dans une liste **HTML** les éléments du tableau `stagiaires` (afficher uniquement nom et prénom). Les éléments d'indice pair seront affichés en rouge, les impairs en bleu.

Angular

Première solution

```
<ul>
  <ng-container *ngFor="let elt of stagiaires; let isEven =
     even; let isOdd = odd">
    <li [ngClass]="{'rouge': isEven,'bleu': isOdd}">
      {{ elt.nom }} {{ elt.prenom }}
    </li>
  </ng-container>
</ul>
```

Angular

Première solution

```
<ul>
  <ng-container *ngFor="let elt of stagiaires; let isEven =
     even; let isOdd = odd">
    <li [ngClass]="{'rouge': isEven,'bleu': isOdd}">
      {{ elt.nom }} {{ elt.prenom }}
    </li>
  </ng-container>
</ul>
```

Deuxième solution

```
<ul>
  <ng-container *ngFor="let elt of stagiaires; let i = index;">
    <li [ngClass]="i % 2 != 0 ? 'rouge' : 'bleu'">
      {{ elt.nom + " " + elt.prenom }}
    </li>
  </ng-container>
</ul>
```

Angular

Troisième solution

```
<ul>
  <ng-container *ngFor="let stagiaire of stagiaires">
    <li [ngStyle]="{color: getNextColor()}">
      {{ stagiaire.prenom }} {{ stagiaire. nom }}
    </li>
  </ng-container>
</ul>
```

Angular

Troisième solution

```
<ul>
  <ng-container *ngFor="let stagiaire of stagiaires">
    <li [ngStyle]="{color: getNextColor()}">
      {{ stagiaire.prenom }} {{ stagiaire. nom }}
    </li>
  </ng-container>
</ul>
```


Dans `app.ts`, on définit la méthode `getNextColor`

```
couleur = 'blue';
getNextColor(): string {
    this.couleur = this.couleur === 'red' ? 'blue' : 'red';
    return this.couleur;
}
```

Angular

Remarques

- Dans un projet standalone créé avec **Angular 17** ou une version ultérieure, il est recommandé d'utiliser la nouvelle syntaxe de contrôle de flux : `@if`, `@for`, `@switch`...
- Pour un projet existant devant rester compatible avec **Angular 16** ou une version antérieure, il est préférable de continuer à utiliser le **CommonModule** (`*ngIf`, `*ngFor`...), surtout si une grande base de code repose déjà dessus.

Angular

Avantages de `@if`, `@for`...

- **Plus lisible** : pas d'astérisques ni de templates implicites...
- **Plus performant** : génère moins de code **Angular** interne, optimisé par le compilateur...
- **Plus flexible** : tracking plus clair avec `track`.
- Pas besoin d'importer **CommonModule** pour utiliser `*ngIf`, `*ngFor`...

Angular

Considérons les trois attributs suivants (à déclarer dans `app.ts`

```
valeur1 = 2
valeur2 = 5
resultat = 0
```

Angular

Considérons les trois attributs suivants (à déclarer dans `app.ts`

```
valeur1 = 2
valeur2 = 5
resultat = 0
```

Et le constructeur

```
constructor() {
    setInterval(
        () => this.valeur1 += 10,
        5000
    )
    this.resultat = this.valeur1 + this.valeur2
}
```

Angular

Considérons les trois attributs suivants (à déclarer dans `app.ts`

```
valeur1 = 2
valeur2 = 5
resultat = 0
```

Et le constructeur

```
constructor() {
    setInterval(
        () => this.valeur1 += 10,
        5000
    )
    this.resultat = this.valeur1 + this.valeur2
}
```

Le contenu de `app.html` est le suivant

```
<p>
    {{ valeur1 }} + {{ valeur2 }} = {{ resultat }}
</p>
```

Angular

Question

Que sera le résultat de ce code ?

Angular

Question

Que sera le résultat de ce code ?

Résultat attendu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 2 sera remplacé par 12
- Contenu attendu : `12 + 3 = 15`

Résultat obtenu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, rien ne change
- Contenu affiché : `2 + 3 = 5`

Angular

Conclusion

Quand `valeur1` ne change pas.

Angular

Conclusion

Quand `valeur1` ne change pas.

Solution

Utiliser les valeurs réactives (signaux).

Angular

Conclusion

Quand `valeur1` ne change pas.

Solution

Utiliser les valeurs réactives (signaux).

Signal

- une enveloppe autour d'une valeur qui peut informer les consommateurs intéressés lorsque cette valeur change.
- peut contenir n'importe quel type de valeur : primitive ou complexe.

Angular

Angular propose deux types de signaux

- **Writable Signals** : valeur modifiable via les méthodes
  - `set(newValue)` ou
  - `update(callback)`
- **Computed Signals** (en lecture seule) : obtient une valeur à partir d'un ou plusieurs autres signaux.

Angular

Commençons par initialiser `value1` et `value2` avec la fonction `signal`

```
valeur1 = signal(2)
valeur2 = signal(5)
```

Angular

Commençons par initialiser `value1` et `value2` avec la fonction `signal`

```
valeur1 = signal(2)
valeur2 = signal(5)
```


Remarque

`valeur1` et `valeur2` sont de type `WritableSignal`.

Angular

Commençons par initialiser `value1` et `value2` avec la fonction `signal`

```
valeur1 = signal(2)
valeur2 = signal(5)
```


Remarque

`valeur1` et `valeur2` sont de type `WritableSignal`.

On peut donc typer ainsi

```
valeur1: WritableSignal<number> = signal(2)
valeur2: WritableSignal<number> = signal(5)
```

Angular

Dans le constructeur, utilisons `update` pour modifier `value1`

```
constructor() {
    setInterval(
      () => this.valeur1.update(v => v + 10),
      5000
    )
    this.resultat = this.valeur1() + this.valeur2()
}
```

Angular

Dans le constructeur, utilisons `update` pour modifier `value1`

```
constructor() {
    setInterval(
      () => this.valeur1.update(v => v + 10),
      5000
    )
    this.resultat = this.valeur1() + this.valeur2()
}
```


Utilisons les getters dans `app.html`

```
<p>
    {{ valeur1() }} + {{ valeur2() }} = {{ resultat }}
</p>
```

Angular

Ou avec `set`

```
constructor() {
  setInterval(
    () => this.value1.set(this.valeur1() + 10),
    5000
  )
  this.resultat = this.valeur1() + this.valeur2()
}
```

Angular

Ou avec `set`

```
constructor() {
  setInterval(
    () => this.value1.set(this.valeur1() + 10),
    5000
  )
  this.resultat = this.valeur1() + this.valeur2()
}
```


Rien à changer dans le template

```
<p>
    {{ valeur1() }} + {{ valeur2() }} = {{ resultat }}
</p>
```

Angular

Question

Que sera le résultat de ce code ?

Angular

Question

Que sera le résultat de ce code ?

Résultat attendu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 2 sera remplacé par 12
- Contenu attendu : `12 + 3 = 15`

Résultat obtenu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 2 est remplacé par 12
- Contenu affiché : `12 + 3 = 5`

Angular

Préparons l'attribut `resultat` et initialisons le à 0

```
resultat: Signal<number> = signal(0)
```

Angular

Préparons l'attribut `resultat` et initialisons le à 0

```
resultat: Signal<number> = signal(0)
```


Remarque

`resultat` est de type `Signal<number>` donc il est en lecture-seule.

Angular

Dans le constructeur, Utilisons `computed` pour recalculer la valeur de `resultat` après chaque changement de valeur de `value1` ou `value2`

```
constructor() {
  setInterval(
    () => this.valeur1.update(val => val += 10),
    5000
  )
  this.resultat = computed(() => this.valeur1() + this.valeur2())
}
```

Angular

Dans le constructeur, Utilisons `computed` pour recalculer la valeur de `resultat` après chaque changement de valeur de `value1` ou `value2`

```
constructor() {
  setInterval(
    () => this.valeur1.update(val => val += 10),
    5000
  )
  this.resultat = computed(() => this.valeur1() + this.valeur2())
}
```


Les imports nécessaires

```
import { Component, Signal, WritableSignal, computed, signal } from '
  @angular/core';
```

Angular

Ou avec `set`

```
constructor() {
  setInterval(
    () => this.value1.set(this.valeur1() + 10),
    5000
  )
  this.resultat = computed(() => this.valeur1() + this.valeur2())
}
```

Angular

Pour accéder aux valeurs d'un signal, on utilise le getter (on rajoute donc les parenthèses)

```
<p>
    {{ valeur1() }} + {{ valeur2() }} = {{ resultat() }}
</p>
```

Angular

Pour suivre l'état d'un signal, on utilise `effect`

```
export class ReactiveComponent {
  valeur1: WritableSignal<number> = signal(2)
  valeur2: WritableSignal<number> = signal(5)
  resultat: Signal<number> = signal(0)
  constructor() {
    setInterval(
      () => this.valeur1.update(val => val += 10),
      2000
    )
    this.resultat = computed(() => this.valeur1() + this.valeur2())
  }
  r = effect(() => console.log('effet', this.valeur1(), this.resultat()))
}
```

Angular

Pour suivre l'état d'un signal, on utilise `effect`

```
export class ReactiveComponent {
  valeur1: WritableSignal<number> = signal(2)
  valeur2: WritableSignal<number> = signal(5)
  resultat: Signal<number> = signal(0)
  constructor() {
    setInterval(
      () => this.valeur1.update(val => val += 10),
      2000
    )
    this.resultat = computed(() => this.valeur1() + this.valeur2())
  }
  r = effect(() => console.log('effet', this.valeur1(), this.resultat()))
}
```


Remarque

`effect` observe `resultat` et `valeur1`. Le changement de valeur de tout autre signal ne permettra pas de le déclencher.

Angular

effect peut-être utilisé dans les cas suivants

- enregistrer la valeur d'un signal à l'aide d'un framework de journalisation.
- exporter la valeur d'un signal vers le **Web Storage** ou un **Cookie**.
- enregistrer la valeur d'un signal dans la base de données.

Angular

Pour appliquer un traitement particulier à la destruction de l'`effect`

```
r = effect((onCleanup) => {
  console.log('effet', this.valeur1(), this.resultat())
  onCleanup(() => {
    console.log("Traitement appliqué à la destruction de l'effect");
  });
})
```

Angular

Question

Dans quel cas, faudrait-il utiliser les signaux ?

Angular

Question

Dans quel cas, faudrait-il utiliser les signaux ?

Réponse

- Pour toutes les données qui changent au fil du temps et qui sont utilisées dans le template ou dans les expressions qui alimentent le template.
- Pour les données passées comme des entrées (`@Input()`) et que vous voulez réagir de manière synchrone (en utilisant input() ou model()).

Angular

Question

Quand l'utiliser n'est pas nécessaire ?

Angular

Question

Quand l'utiliser n'est pas nécessaire ?

Réponse

- Pour les données qui sont initialisées une seule fois et qui ne changeront jamais (des constantes).
- Pour des méthodes ou des calculs qui ne sont pas censés être tracés comme une dépendance directe