

Blog .NET

[.NET ▾](#)[DEBUG](#)[WEB ▾](#)[BIG DATA ▾](#)[ARCHITECTURE ▾](#)[AIDE-MÉMOIRES](#)

Quelques découvertes, trucs et astuces sur .NET en général



by [Mathias Montantin](#)

29 mars 2021

Les vues des composants Angular

Cet article fait partie de la série d'articles [Angular from Scratch](#).



[@amyb99](#)

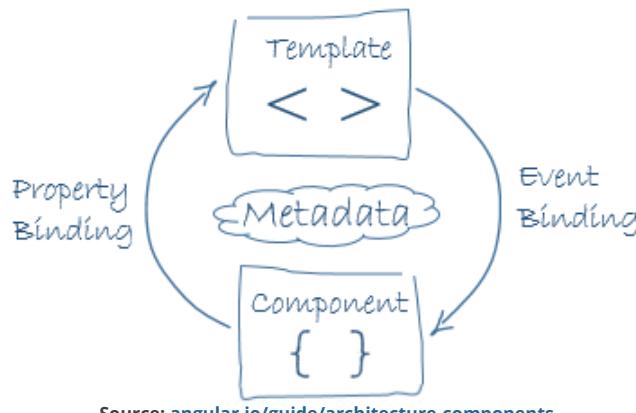
Les composants Angular sont des objets permettant d'afficher une vue. Il s'agit d'une unité d'implémentation correspondant à une partie de l'affichage d'une application Angular. Cette unité comporte de nombreux éléments permettant de faciliter l'implémentation de façon à, dans un 1^{er} temps, initialiser les objets statiques de la vue et dans un 2^e temps, à interagir avec le DOM pour modifier dynamiquement des éléments d'affichage.

Pour ordonner cette unité d'implémentation, un composant est formé de différents éléments:

- **La vue du composant appelée template:** cette partie comporte du code HTML correspondant aux objets statiques. Ce squelette statique est enrichi par du code interprété par Angular pour permettre des interactions dynamiques entre les objets statiques et du code métier se trouvant dans le reste du composant.
Cet article détaille quelques éléments de cette partie du composant.
- **La classe du composant:** c'est une classe Typescript dans laquelle on peut définir:
 - des membres contenant les données affichées par la vue,
 - des méthodes et fonctions pour exécuter des traitements déclenchés par des actions sur la vue ou par des événements divers.

- **Des métadonnées:** il s'agit d'informations supplémentaires d'implémentation qui permettront à Angular d'interfacer le *template* avec la classe du composant.

Ce schéma provenant de la documentation Angular permet de résumer les différents éléments du composant:



Le but de cet article est de passer en revue les éléments les plus importants pour implémenter un *template*:

- Indiquer comment on peut implémenter le *template* d'un composant.
- Détails l'utilité du paramètre `[selector]` qui permettra de placer la vue dans le reste de l'application Angular.
- Présenter les différents types de *bindings* pour permettre les interactions entre le *template* et la classe du composant.
- Indiquer comment on peut nommer des éléments sur le *template* avec une variable référence.
- Détails quelques directives usuelles facilitant l'implémentation du *template*.

Sommaire

Implémenter le template

Paramètre selector

Binding

Interpolation

Property binding

Attribute binding

Event binding

Two-way binding

Composant enfant

`ngModel`

Variable référence dans le template (#variable)

Variable référence contenant une propriété du DOM

Variable référence contenant `ngModel`

Accéder à une variable référence dans la classe du composant

Directives structurelles usuelles

`ngIf`

Syntaxe avec `else`

Syntaxe avec `then` et `else`

`ngFor`

`index`, `count`, `first`, `last`, `odd` et `even`

`ngSwitch`

`ngPlural`

Pour résumer...

Implémenter le *template*

Pour créer un composant Angular, il faut exécuter la commande suivante en utilisant le [CLI Angular](#):

```
ng g c <nom composant>
```

Par exemple, pour créer le composant `Example`:

```
ng g c example
```

Cette instruction va créer 4 fichiers:

- `[src/app/example/example.component.ts]`: il s'agit de la classe du composant. C'est le seul fichier obligatoire.
- `[src/app/example/example.component.html]`: ce fichier est le *template* du composant.
- `[src/app/example/example.component.css]`: ce fichier permet d'implémenter, le cas échéant, les styles ou classes CSS relatifs à la vue (le détail de ce fichier ne sera pas traité dans cet article et fera l'objet d'un article ultérieur).
- `[src/app/example/example.component.spec.ts]`: ce fichier permet d'implémenter des tests liés à la classe du composant (le détail de ce fichier ne sera pas traité dans cet article).

Le contenu du *template* et de la classe du composant est:

Template (example.component.html)	<pre><p>example works!</p></pre>
Classe du composant (example.component.ts)	<pre>import { Component, OnInit } from '@angular/core'; @Component({ selector: 'app-example', templateUrl: './example.component.html', styleUrls: ['./example.component.css'] }) export class ExampleComponent implements OnInit { constructor() { } ngOnInit(): void { } }</pre>

Dans la classe du composant dans le décorateur `@Component()`, quelques paramètres permettent d'enrichir les métadonnées du composant:

- `[selector]`: ce paramètre permet d'indiquer où la vue correspondant au composant sera affichée dans l'application Angular. Ce paramètre sera détaillé dans [Paramètre \[selector\]](#).
- `[templateUrl]`: ce paramètre indique le chemin du fichier *template*.
- `[styleUrls]`: ce paramètre indique les chemins des fichiers CSS contenant les styles utilisés par la vue du composant.

Il est possible de ne pas utiliser un fichier séparé pour l'implémentation du *template*.

Avec le paramètre `[template]` dans le décorateur `@Component()`, on peut implémenter le *template* directement dans la classe du composant, par exemple:

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

@Component({
  selector: 'app-example',
  template: `<p>example works!</p>`,
  styleUrls: ['./example.component.css']
})
export class ExampleComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}
```

La valeur du paramètre `[template]` doit être indiquée entre les caractères `[[...]]` (accessible avec les touches `[AltGr] + [7]`) et non de simples quotes `'[[...]]'`.

Paramètre *selector*

Le paramètre `[selector]` peut être utilisé au niveau du décorateur `[@Component()]` d'un composant et plus généralement d'une [directive](#). Il permet d'indiquer où le rendu de la vue du composant sera effectué. Ce paramètre n'est pas obligatoire, si on utilise un [router](#) Angular il peut être omis.

Pour comprendre l'intérêt de ce paramètre, on prend l'exemple du composant `[Example]` créé précédemment:

Template	<pre><p>example works!</p></pre>
Classe du composant	<pre>import { Component } from '@angular/core'; @Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent {</pre>

La valeur du paramètre `[selector]` de ce composant est:

```
app-example
```

Cela signifie que le rendu sera effectué sur la vue d'un composant si le *template* de ce composant contient:

```
<app-example></app-example>
```

Ainsi si:

1. On supprime tout le contenu de la vue du composant principal dans le fichier `[src/app/app.component.html]`.
2. On importe le composant `[ExampleComponent]` dans la classe du composant principal (dans `[src/app/app.component.ts]`) en ajoutant:

```
import { ExampleComponent } from './example/example.component';
```

3. Dans le *template* du composant principal (dans `[src/app/app.component.html]`), on indique l'appel au `[selector]`:

```
<app-example></app-example>
```

L'implémentation du composant principal devient:

Template	<pre><app-example></app-example></pre>
Classe du composant	<pre>import { Component } from '@angular/core'; import { ExampleComponent } from './example/example.component'; @Component({ selector: 'app-root', templateUrl: './app.component.html', styleUrls: ['./app.component.css'] }) export class AppComponent {</pre>

Après avoir exécuté les commandes suivantes:

```
npm install
ng serve
```

L'affichage est:

example works!

Ainsi le rendu de la vue du composant `[ExampleComponent]` (dans `src/app/example/example.component.html`) a été effectué dans la vue du composant principal `[AppComponent]` (dans `src/app/app.component.html`).

Le paramètre `[selector]` peut servir dans le cadre de composant enfant c'est-à-dire si on imbrique un composant dans un autre (pour plus de détails voir [les composants enfants](#)).

Enfin le paramètre `[selector]` peut être utilisé sous des formes différentes de celle présentée dans cet exemple:

- **Attribut d'un élément HTML:**

Si le paramètre `[selector]` est défini de cette façon dans le composant:

```
selector: '[custom-directive]'
```

La vue du composant sera affichée si le `[selector]` est utilisé sous forme d'attribut d'un élément HTML, par exemple:

```
<span custom-directive></span>
```

L'attribut peut comporter une valeur:

```
<span custom-directive="attribute value"></span>
```

La directive est reconnue aussi dans le cas d'un [property binding](#) (cette notion est définie plus bas):

```
<span [custom-directive]="bindedProperty"></span>
```

Toutefois dans ce dernier cas, la directive devra comporter un [paramètre d'entrée](#) avec le même nom ou le même alias que le paramètre `[selector]`.

- **Classe CSS:**

Si le paramètre `[selector]` est défini de cette façon dans le composant:

```
selector: '.custom-directive'
```

Dans ce cas, la directive doit être utilisée dans une classe CSS d'un élément HTML, par exemple:

```
<span class="custom-directive"></span>
```

D'autres types de conditions sont possibles pour afficher le composant (voir [les directives](#) pour plus détails sur ces conditions).

Binding

Les interactions entre la vue et la classe du composant sont possibles avec différents types de *bindings*. L'implémentation de ces *bindings* permet d'enrichir la vue avec des données ou de déclencher l'exécution de code dans la classe du composant.

Le *binding* permet l'interfaçage entre le *template* et la classe du composant en:

- Faisant passer des données de la classe du composant vers le *template*.
- Déclanchant l'exécution de méthodes dans la classe du composant à la suite du déclenchement d'évènements dans le *template*.

Le *binding* permet de faciliter l'implémentation de tous ces mécanismes en rendant, par exemple, automatique:

- La mise à jour d'un élément graphique quand la valeur d'un membre a changé dans la classe du composant,
- Déclanchant l'exécution d'une méthode quand un évènement est survenu dans un élément HTML de la vue.

Ces mécanismes sont possibles grâce à la détection de changements effectuée par Angular (pour plus de détails, voir le [Fonctionnement de la détection de changements](#)).

Il existe plusieurs méthodes pour effectuer le *binding*:

- L'**interpolation** permettant d'exécuter dans le *template* une expression Javascript pouvant contenir des attributs provenant de la classe du composant.
- Le **property binding** qui permet d'effectuer un *binding* d'un membre de la classe du composant vers le *template*.
- L'**event binding** permettant de déclencher un évènement dans la classe du composant à partir d'un évènement déclenché dans un objet du DOM.
- L'**attribute binding** permettant d'effectuer un *binding* entre l'attribut d'un élément HTML et un attribut de la classe du composant.
- Le **two-way binding** permettant à la fois l'échange de données entre la classe du composant et le *template* (1^{er} sens) et le déclenchement d'évènements dans la classe du composant (2^e sens).

On va détailler chacune de ces méthodes.

Interpolation

C'est le *binding* le plus simple qui permet l'échange de données dans un sens: de la classe du composant vers le *template*. Il permet d'exécuter une expression et d'utiliser directement des membres ou des fonctions publiques de la classe du composant dans le *template* avec la syntaxe `[[[...]]]`.

Par exemple, si on considère le composant suivant:

Template	<code><h1>{{title}}</h1></code>
Classe du composant	<pre>@Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent { title = 'Page title'; }</pre>

L'instruction `[[[title]]]` permet d'utiliser la valeur de le membre `title` de la classe dans le *template*.

Il est possible d'exécuter une fonction et d'utiliser sa valeur de retour, par exemple:

Template	<code><h1>{{getPageTitle()}}</h1></code>
Classe du composant	<pre>@Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent { private title = 'Page title'; getPageTitle: string { return this.title; } }</pre>

On exécute la fonction `getPageTitle()` pour afficher sa valeur de retour.

Dans la partie entre les crochets, on peut exécuter une expression, par exemple:

Template	<code>[[[getPageTitle().toUpperCase()]]]</code>
Classe du composant	<pre>@Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent { private title = 'Page title'; getPageTitle: string { return this.title; } }</pre>

`getPageTitle().toUpperCase()` permet de transformer les lettres du titre en majuscules.

Pour que le code de l'expression soit valide, il ne doit pas provoquer la création d'un nouvel objet ou le changement de valeur d'un objet existant. L'instanciation d'un objet ou les affectations ne sont, par exemple, pas possibles.

Property binding

Le *property binding* permet d'échanger des données de la classe du composant vers le *template*. Il permet de renseigner une propriété d'un objet dans le DOM avec une valeur provenant de le membre de la classe du composant. Indirectement la propriété dans le DOM se reflète sur son équivalent au niveau de la vue.

Par exemple, pour renseigner la propriété `[alt]` d'un élément HTML `img` avec un membre `imgAltText` du composant, le code est:

Template	<pre><p>Example Componant</p> </pre>
Classe du composant	<pre>@Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent { imgAltText = 'image to display'; }</pre>

Le code `[alt]='imgAltText'` permet de renseigner la propriété `[alt]` de l'élément HTML avec la valeur du membre `imgAltText` de la classe du composant.

L'affichage est:

Example Componant

image to display

Ainsi:

- `[]` est le propriété de l'élément cible du DOM
- `' '` est la source du *binding* dans la classe du composant.

Avec l'interpolation, on pourrait avoir un résultat similaire en utilisant l'implémentation suivante dans le *template*:

```
<img alt='{{imgAltText}}' />
```

Si dans cet exemple on omet les crochets `[]` autour de la propriété de l'élément HTML il n'y a pas de *binding* et le texte `'imgAltText'` est affecté statiquement à l'attribut `alt` de `img`, par exemple:

Example Componant

imgAltText

Si la propriété de l'élément HTML n'existe pas, une erreur se produit, par exemple:

```
<img [unknown]='imgAltText' />
```

L'erreur est du type:

```
Can't bind to 'unknown' since it isn't a known property of 'img'.
```

Le *property binding* permet d'effectuer un *binding* avec une propriété d'un élément HTML mais il permet aussi d'effectuer un *binding* dans des cas plus complexes comme:

- Effectuer une *binding* avec un paramètre d'entrée d'un composant enfant: voir [les composants enfant](#) pour plus de détails.
- Plus généralement effectuer un *binding* avec le paramètre d'une directive: voir [les directives](#) pour plus de détails.

Ne pas confondre propriété d'un objet du DOM et attribut d'un élément HTML

Le *property binding* permet d'interagir avec une propriété d'un objet du DOM et non directement avec un attribut d'un élément HTML. Une propriété du DOM et un attribut HTML ne correspondent pas à la même chose même s'ils possèdent le même nom, par exemple:

- Des attributs HTML peuvent ne pas avoir de propriétés correspondantes comme `[colspan]`.
- Des propriétés du DOM peuvent ne pas avoir d'équivalent au niveau des attributs HTML comme `[textContent]`.
- Des attributs HTML et des propriétés DOM n'ont pas une correspondance directe comme `[value]`. Dans le cas du DOM c'est la valeur actuelle alors que dans le cas de l'attribut c'est la valeur initiale.

Si on prend l'exemple de l'attribut `[colspan]` de l'élément HTML `[td]`, il faut utiliser la propriété DOM `[colSpan]`. Ainsi, pour effectuer un *property binding* entre la propriété `[colSpan]` de l'objet du DOM et un attribut nommé `[spanValue]` dans la classe du composant, on écrira:

```
<td [colSpan]='spanValue'></div>
```

Le code suivant produit une erreur car la propriété DOM `[colspan]` n'existe pas:

```
<td [colspan]='spanValue'></div>
```

Il n'est pas évident de connaître les équivalents entre attribut HTML et propriété du DOM, on peut s'aider de ce tableau dans le code source d'Angular:

[dom_element_schema_registry.ts](#)

Attribute binding

L'*attribute binding* est similaire au *property binding* sauf qu'on utilise la syntaxe `[attr.<nom de l'attribut>]` pour désigner l'attribut de l'élément HTML.

Ainsi pour effectuer un *binding* entre le membre `[imgAltText]` de la classe du composant et l'attribut `[alt]` d'un élément `[img]`, la syntaxe est:

Template	<pre><p>Example Composant</p> </pre>
Classe du composant	<pre>@Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent { imgAltText = 'image to display'; }</pre>

Définition de l'*attribute binding* et le *property binding*

La différence entre l'*attribute binding* et le *property binding* est que l'*attribute binding* intervient sur l'attribut de l'élément HTML pour affecter dynamiquement une valeur. Le *property binding* passe par l'intermédiaire d'une propriété dans l'objet dans le DOM correspondant à l'élément HTML.

L'*attribute binding* fait toujours référence à un attribut existant alors que le *property binding* peut faire référence à une propriété Angular qui n'a pas forcément d'équivalent dans l'objet dans le DOM.

Par exemple, si on considère l'exemple suivant:

```
<div [ngStyle]="{'color': yellow}"></div>
```

Ce code permet d'affecter un style à l'élément HTML `[div]` en passant par la directive Angular `[ngStyle]`. `[ngStyle]` ne correspond pas à un attribut HTML. On ne peut pas écrire:

```
<div ngStyle="{'color': yellow}"></div>
```

De même, l'affectation de la propriété `[colspan]` de l'élément HTML `[td]` peut prêter à confusion:

- `[colspan]` fait référence à l'attribut de l'élément HTML `[td]`.
- `[colSpan]` fait référence à la propriété de l'objet dans le DOM qu'Angular pourra modifier.

Ainsi, pour effectuer un *attribute binding* entre l'attribut `[colspan]` de l'élément HTML `[td]` et un attribut nommé `[spanValue]` dans la classe du composant, on écrira:

```
<td [attr.colspan]='spanValue'></div>
```

Pour effectuer un *property binding* entre la propriété `[colSpan]` de l'objet du DOM et un attribut nommé `[spanValue]` dans la classe du composant, on écrira:

```
<td [colSpan]='spanValue'></div>
```

Event binding

L'*event binding* permet d'exécuter du code dans la classe du composant à partir d'événements déclenchés sur un élément du DOM.

Par exemple, pour exécuter la méthode `[displayText()]` dans la classe du composant à partir d'un clique sur un bouton, l'implémentation est:

Template	<pre><p>{{textToDisplay}}</p> <p><button (click)='displayText()'>Click Me</button></p></pre>
Classe du composant	<pre>@Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent { TextToDisplay = ''; displayText(): void { this.TextToDisplay = 'My Text'; } }</pre>

Ainsi l'événement `[click]` dans le code HTML déclenche l'exécution de la méthode `[displayText()]`. Par *binding*, la valeur de la propriété `[textToDisplay]` est mise à jour directement dans le DOM.

Cet exemple permet de montrer l'exemple d'*event binding* entre un élément HTML et une méthode dans la classe du composant toutefois il existe des cas d'utilisations plus complexes dans le cadre:

- De composants enfant: pour plus de détails se reporter à [@Output\(\) + EventEmitter](#).
- De directives: pour plus de détails se reporter au paramètre `[outputs]` du décorateur `[@Directive()]` ou au décorateur `[@Output()]`.

Two-way binding

Le *two-way binding* permet d'effectuer un *binding* dans plusieurs directions:

- Un *property binding* pour injecter la valeur d'un membre d'une classe de composant dans une propriété d'un composant.
- Un *event binding* permettant d'exécuter une méthode après le déclenchement d'un événement.

On indiquera 2 cas d'application d'un *two-way binding*:

- Dans le cas d'un composant enfant et
- Dans le cas d'un formulaire avec l'objet Angular `ngModel`.

Composant enfant

L'implémentation du *two-way binding* avec un composant enfant est possible en implémentant:

- un paramètre d'entrée avec le décorateur `@Input()`.
- Un événement auquel on peut s'abonner à l'extérieur du composant avec le décorateur `@Output()`. Pour que le *two-way binding* fonctionne, l'événement doit s'appeler `<nom du paramètre d'entrée>Change`.

Par exemple, on crée le composant `counter` en exécutant:

```
ng g c counter
```

On modifie l'implémentation pour que ce composant contienne un compteur qui sera incrémenté en cliquant sur un bouton. La valeur initiale de ce compteur sera injectée avec un paramètre d'entrée. Chaque événement de *click* sur le bouton déclenchera un événement à l'extérieur du composant. L'implémentation est:

Template <pre><p>Counter: {{count}}</p> <button (click)='incrementValue()'>Increment</button></pre>	Classe du composant <pre>import { Component, Input, Output, EventEmitter } from '@angular/core'; @Component({ selector: 'counter', templateUrl: './counter.component.html' }) export class CounterComponent { @Input() count: number; @Output() countChange: EventEmitter<number> = new EventEmitter<number>(); incrementValue(): void { this.count++; this.countChange.emit(this.count); } }</pre>
---	--

On peut remarquer que le paramètre d'entrée s'appelle `count`. Pour permettre le *two-way binding* l'événement doit s'appeler `countChange`.

On utilise ce composant à l'intérieur d'un autre composant en tant que [composant enfant](#).

On crée un composant parent qui contiendra le composant `counter` en exécutant:

```
ng g c parent
```

L'implémentation du composant `parent` est:

Template <pre><p>Parent Component</p> <counter [(count)]="bindedCount"></counter> <p>Two-way binded counter: {{bindedCount}}</p></pre>	Classe du composant <pre>import { Component } from '@angular/core'; @Component({ selector: 'app-parent', templateUrl: './parent.component.html' }) export class ParentComponent {</pre>
--	--

```

    }  

}

```

De point de vue de l'implémentation:

- Le *template* du composant `[parent]` contient le code pour afficher le composant enfant `[counter]`.
- L'attribut `[(count)]=`bindedCount`` permet d'implémenter un *two-way binding* avec la notation `[(...)]`:
 - Un *property binding* permet d'injecter dans la propriété `[count]` du composant `[counter]` la valeur du membre `bindedCount` provenant de la classe du composant `[parent]` (1^{er} sens du *binding*).
 - Un *event binding* permet de mettre la valeur `bindedCount` dans la classe du composant `[parent]` à chaque fois que la propriété `[count]` est mise à jour dans le composant `[counter]` (2^e sens du *binding*).

Ainsi, à l'exécution, on peut voir que:

- La paramètre d'entrée `[count]` est bien paramétrée avec la valeur d'initialisation `[10]` provenant du membre `bindedCount` du composant `[parent]`.
- A chaque *click* sur le bouton, le membre `[count]` dans le composant `[counter]` est incrémenté et le membre `[binedCount]` dans le composant `[parent]` est aussi mis à jour.

L'interface se présente de cette façon:

Parent Componant

Counter: 12

Increment

Two-way binded counter: 12

ngModel

Une autre implémentation permet d'effectuer un *two-way binding* dans le cas de formulaire en utilisant `:ngModel`.

Par exemple, si on utilise l'élément HTML `:input`

Template <pre> <p><input [(ngModel)]='textToDisplay' /></p> <p><button (click)='eraseInputValue()'>Erase</button></p> <p>{{textToDisplay}}</p> </pre>	Classe du composant <pre> @Component({ selector: 'app-example', templateUrl: './example.component.html' }) export class ExampleComponent { TextToDisplay = ''; eraseInputValue(): void { this.TextToDisplay = 'My Text'; } } </pre>
---	--

Si on exécute ce code directement, un erreur se produira:

```
Can't bind to 'ngModel' since it isn't a known property of 'input'.
```

Pour corriger le problème, il faut importer le module `:FormsModule` dans le module du composant. Dans notre exemple, le module du composant est le module `:root`. Ainsi il faut rajouter dans le fichier `src/app/app.module.ts`, l'import de `:FormsModule`:

```

import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [ CommonModule, FormsModule ]
})

```

```
})
export class AppModule {}
```

Dans cet exemple, si on écrit dans la partie `[input]`, la propriété `[textToDisplay]` sera directement mise à jour et l'interpolation `[[textToDisplay]]` affichera directement la nouvelle valeur. Ainsi l'échange se fait du *template* vers le membre de la classe du composant.

Si on clique sur le bouton, la valeur de la propriété `[textToDisplay]` est effacée par la méthode `eraseInputValue()` et la nouvelle valeur est directement répercutée dans la vue. L'échange se fait dans l'autre sens c'est-à-dire du membre de la classe du composant vers la vue.

Si on n'utilisait pas `[ngModel]`, il aurait fallu écrire le code coté *template* de cette façon:

```
<input [value]='textToDisplay' (input)='textToDisplay = $event.target.value'
```

Cette implémentation associe plusieurs comportements:

- Un *property binding* entre la propriété `[value]` de l'élément HTML `[input]` et le membre `[textToDisplay]` de la classe du composant. Ce *binding* permet l'échange de valeur du composant vers la vue.
- Un *event binding* sur l'événement `[input]` qui permet d'exécuter le code: `[textToDisplay = $event.target.value]` permettant d'affecter une nouvelle valeur au membre `[textToDisplay]` du composant à partir du contenu de l'élément `[input]`.

Utiliser `[ngModel]` simplifie la syntaxe en permettant d'effectuer un *binding* entre les membres de la classe du composant avec des éléments HTML de type `[input]`, `[select]` ou `[textarea]`. Par exemple cette directive s'utilise de cette façon pour un élément de type `[input]`:

```
<input [ngModel]='textToDisplay' (ngModelChange)='textToDisplay = $event' /
```

Avec cette syntaxe, le *property binding* et l'*event binding* apparaissent plus explicitement. L'*event binding* (`(ngModelChange)`) notifie quand un changement est détecté dans la vue, il extrait la cible de l'événement déclenché à partir de `[event]` au lieu d'utiliser `[event.target.value]` comme dans la syntaxe précédente.

Variable référence dans le *template* (#variable)

On peut définir des variables dans le *template* pour référencer un élément du DOM. La portée de cette variable sera le *template* c'est-à-dire qu'elle sera accessible et qu'elle doit être unique dans le cadre du *template*.

Par défaut, la variable contient une référence vers un élément du DOM et non un objet Angular. Toutefois il est possible de modifier le comportement pour que la variable contienne un objet Angular utilisé dans le *template* comme `[ngModel]` ou `[ngForm]`.

Pour déclarer une variable référence, il faut utiliser la syntaxe `#[nom de La variable]` ou `ref-<nom de La variable>` (cette dernière syntaxe n'est pas courante).

Par exemple:

```
<p>
  <input #InputElement type='text' value='initialValue' />
  <button #clickButton>Click Me</button>
</p>
```

`#InputElement` et `#clickButton` permettent d'accéder à l'objet dans le DOM de, respectivement, l'élément `[input]` et `[button]` dans le reste du code du *template*. Ces variables ne sont toutefois pas directement accessible dans la classe du composant.

On peut, ainsi, utiliser les variables référence pour accéder aux objets en utilisant le nom de la variable sans le `[#]`:

```
<p>Input element value: {{inputElement.value}}</p>
<p>Button textContent: {{clickButton.textContent}}</p>
```

Le résultat est:

Input element value: initialValue

Button textContent: Click Me

Il est important de comprendre que ces variables font référence aux objets du DOM, c'est la raison pour laquelle elles ne sont pas directement accessibles dans la classe du composant. Elles donnent la possibilité d'accéder aux 3 types d'objets dans le *template*:

- **Les éléments HTML**: si on exécute le code suivant, on peut accéder à l'attribut `[value]` de l'élément HTML `[input]`. La valeur de cet attribut ne changera pas si on change la valeur de l'objet DOM correspondant.
- **Les objets du DOM** et
- **Certains objets Angular** comme `[ngModel]` ou `[ngForm]`.

Variable référence contenant une propriété du DOM

Pour comprendre les différences, on peut considérer l'exemple suivant:

```
<p><input #inputElement type='text' value='initialValue' /></p>
<p>Input element: {{inputElement.getAttribute('value')}}</p>
```

On peut accéder à l'attribut `[value]` de l'élément HTML `[input]`. La valeur de cet attribut ne changera pas même si on change la valeur de l'objet DOM correspondant. Le résultat de `[inputElement.getAttribute('value')]` est `'initial value'`.

Si on modifie le code en effectuant un *binding* dans les 2 directions (i.e. *two-way binding*) avec `[ngModel]`, on peut modifier le contenu de `[value]` de l'élément `[input]` sans modifier la valeur de l'attribut HTML `[value]`:

```
<p><input #inputElement type='text' value='initialValue' [(ngModel)]= 'input'></p>
<p>Input element: {{inputElement.getAttribute('value')}}</p>
<p>Input element: {{inputElement.value}}</p>
```

Il faut rajouter la propriété `[inputValue]` dans la classe du composant:

```
@Component({ ... })
export class ExampleComponent {
  inputValue: string;
}
```

Le résultat est:

Input element: initialValue

Input element: entered value

Dans cet exemple, l'attribut HTML initial n'est pas modifié toutefois la propriété de l'objet DOM contenant la valeur est modifiée.

On peut voir le détail de l'objet dans le DOM en rajoutant la méthode suivante dans la classe du composant:

```
log(elementToLog) {
  console.log(elementToLog);
}
```

Et en modifiant le *template*:

```
<p><input #inputElement type='text' value='initialValue' [(ngModel)]= 'input'>
<p>Input element: {{inputElement.getAttribute('value')}}</p>
<p>Input element: {{inputElement.value}}</p>
<p>{{log(inputElement)}}</p>
```

Dans la console, on peut voir la structure de l'objet dans le DOM:

```
input
  > __zone_symbol__ngModelChangefalse: Array [ ... ]
  accept: ""
  accessKey: ""
  accessKeyLabel: ""
  align: ""
  alt: ""
  assignedSlot: null
  attributes: NamedNodeMap(3) [ _ngcontent-esn-c23="", type="text", value="initialValue" ]
  autocomplete: ""
  autofocus: false
  baseURI: "http://localhost:4200/"
  checked: false
  childElementCount: 0
  childNodes: NodeList []
  children: HTMLCollection { length: 0 }
  classList: DOMTokenList []
  className: ""
  clientHeight: 34
  clientLeft: 0
  clientTop: 0
  clientWidth: 186
  contentEditable: "inherit"
  contextMenu: null
  dataset: DOMStringMap(0)
  defaultChecked: false
  defaultValue: "initialValue"
  dir: ""
  disabled: false
```

Variable référence contenant *ngModel*

On peut modifier le comportement pour que la variable référence `[inputElement]` contienne un autre objet que l'objet du DOM. Par exemple, en utilisant l'exemple précédent on peut affecter l'objet Angular `[ngModel]` dans la variable référence `[inputElement]:`

```
<p><input #inputElement='ngModel' type='text' [(ngModel)]= 'inputValue' /></p>
<p>Input element: {{inputElement.value}}</p>
```

En ajoutant la ligne:

```
  {{log(inputElement)}}
```

On peut voir qu'à la différence de l'exemple précédent, la variable référence

`[inputElement]` ne contient plus l'objet du DOM mais l'objet Angular de type `[ngModel]:`

```
{...}
  > __ngContext__: Array(57) [ app-example, ..., 147, ... ]
  > __ngSimpleChanges__: Object { previous: ..., current: null }
  parent: null
  rawAsyncValidators: Array []
  rawValidators: Array []
  registered: true
  control: Object { pristine: true, touched: false, status: "VALID", ... }
  model: undefined
  name: null
  update: Object { _isScalar: false, closed: false, isStopped: false, ... }
  valueAccessor: Object { _renderer: ..., _compositionMode: true, _composing: false, ... }
  viewModel: undefined
  > <prototype>: Object { ... }
```

Accéder à une variable référence dans la classe du composant

Il est possible d'accéder dans la classe du composant à un objet identifié avec une variable référence dans le *template*. L'accès à cet objet n'est pas direct et doit se faire

en effectuant une requête sur les éléments de la vue. Le décorateur `@ViewChild()` permet d'effectuer cette requête en utilisant le nom de la variable référence.

Par exemple, si on considère le composant suivant:

Template	<code><div #divElement>Texte à afficher</div></code>
Classe du composant	<pre>import { Component, OnInit, ViewChild, ElementRef } from '@angular/core'; @Component({ ... }) export class ExampleComponent implements OnInit { @ViewChild('divElement', {static: true}) htmlDivElement: ElementRef; ngOnInit(): void { console.log(this.htmlDivElement.nativeElement.innerText); } }</pre>

Dans cet exemple, le `template` contient un élément HTML `[div]` identifié avec une variable référence `[divElement]`. Dans la classe du composant, on peut requérir la vue en utilisant le décorateur `@ViewChild()` et obtenir un objet permettant d'accéder à l'élément `[div]` du `template`.

Les paramètres du décorateur `@ViewChild()` sont:

- `['divElement']`: ce paramètre correspond à la variable référence utilisée dans le `template`.
- `{ static: true }`: ce paramètre est optionnel et permet d'indiquer que l'élément à requérir fait partie du contenu statique de la vue en opposition au contenu dynamique. L'intérêt de ce paramètre est qu'il autorise à effectuer la requête au début du cycle de vie du composant de façon à ce que l'objet requêté soit accessible lors de l'exécution de `ngOnInit()`.

Si on ne précise pas `{ static: true }`, la valeur par défaut est `{ static: false }`. Cela signifie que la requête sera exécutée plus tard lors du cycle de vie du composant. L'objet ne sera pas disponible à l'exécution de `ngOnInit()`, à ce moment la variable `htmlDivElement` est indéfinie (i.e. `undefined`). La variable `htmlDivElement` sera renseignée à l'exécution de `ngAfterViewInit()`:

```
import { Component, OnInit, AfterViewInit, ViewChild, ElementRef } from '@angular/core';

@Component({ ... })
export class ExampleComponent implements OnInit, AfterViewInit {
    @ViewChild('divElement') htmlDivElement: ElementRef;

    ngOnInit(): void {
        // ERREUR undefined
        // console.log(this.htmlDivElement.nativeElement.innerText);
    }

    ngAfterViewInit(): void {
        // OK
        console.log(this.htmlDivElement.nativeElement.innerText);
    }
}
```

Pour plus de détails sur [le paramètre `static`](#), sur le cycle de vie d'un composant et sur le requêtage d'une vue, voir [Requérir les éléments d'une vue d'un composant Angular](#).

Directives structurelles usuelles

Les directives sont des objets Angular permettant de modifier ou d'enrichir un élément du DOM en rajoutant ou en modifiant une propriété par programmation. L'intérêt des directives est de pouvoir les utiliser dans un composant en les implémentant dans le `template`.

Fonctionnellement les directives peuvent paraître semblables aux composants enfant toutefois la grande différence entre les directives et les composants est que la directive n'a pas de vue. Dans la documentation Angular, les directives sont découpées en 3 catégories:

- **Les composants:** ce sont des directives avec une vue implémentée dans un fichier *template*.
- Les autres types de directives ne disposent pas de vue mais elles permettent de modifier le DOM en ajoutant ou en supprimant des éléments du DOM:
 - **Les directives attribut (i.e. attribute directives):** ces directives peuvent modifier l'apparence et le comportement des éléments, composants et d'autres directives.
 - **Les directives structurelles (i.e. structural directives):** elles se distinguent des directives attribut car elles utilisent un modèle (i.e. *template*) pour modifier le DOM. Il ne faut pas confondre ce modèle avec le *template* d'un composant.

Il est possible d'implémenter complètement des directives toutefois Angular propose des directives usuelles qu'on peut directement utiliser dans le fichier *template*. Le but de cette partie est d'expliquer certaines directives les plus usuelles comme:

- `[ngIf]` pour implémenter une instruction conditionnelle `[if...then...else]`.
- `[ngFor]` pour implémenter l'équivalent d'une boucle `[for]`.
- `[ngSwitch]` pour implémenter un `[switch...case]`.

L'article sur les directives permet d'indiquer davantage de détails pour les implémenter: cdiese.fr/angular-directives.

ngIf

`[ngIf]` permet d'afficher des éléments après avoir évalué si une condition est vraie.

Cette directive transpose l'instruction conditionnelle `[if...then...else]` pour qu'elle soit utilisable dans le *template* d'un composant.

La syntaxe sous sa forme la plus simple si on l'utilise dans un élément HTML `[div]` est:

```
<div *ngIf="<condition>">Contenu affiché si la condition est vraie</div>
```

Ainsi si `[<condition>]` est égale à `[true]`, l'élément HTML sera affiché de cette façon:

```
<div>Contenu affiché si la condition est vraie</div>
```

Si `[<condition> == false]`, rien ne sera affiché.

La syntaxe découle directement du fonctionnement des directives:

- La directive est implémentée sous la forme d'un attribut d'un élément HTML car le paramètre `[selector]` de la directive est indiqué sous la forme `[selector: '[ngIf]'` (voir paramètre `[selector]` pour plus de détails).
- La caractère `*` devant `[ngIf]` indique qu'il s'agit d'une **directive structurelle**.

La syntaxe indiquée plus haut est une forme compacte dont l'équivalent dans sa forme plus étendue est:

```
<ng-template [ngIf]="<condition>">
  <div>Contenu affiché si <condition> == true</div>
</ng-template>
```

`[<ng-template>]` est un objet complexe permettant d'implémenter un modèle utilisé pour créer des vues intégrées (i.e. *embedded view*). Ces vues seront directement ajoutées au DOM suivant l'implémentation de modèle.

Par exemple, si on considère le composant suivant:

Template	<pre><div *ngIf="condition">condition==true</div> <div *ngIf="!condition">condition==false</div> <p>Value of 'condition': {{condition}}</p></pre>
Classe du composant	<pre>@Component({ ... }) export class NgifExampleComponent { condition = true; }</pre>

Dans le fichier `.css` du composant, on ajoute le style:

```
div {
    border: 1px solid black;
    display: block;
    background-color: lightgreen;
}
```

La résultat sera du type:

`condition=true`

Value of 'condition': true

Ainsi comme `[condition == true]` alors seul le 1^{er} élément `[div]` est affiché.

La forme étendue de la syntaxe indiquée plus haut est:

```
<ng-template [ngIf]="condition">
    <div>condition=true</div>
</ng-template>
<ng-template [ngIf]="!condition">
    <div>condition=false</div>
</ng-template>
```

Syntaxe avec `else`

On peut utiliser une syntaxe correspondant à une clause `[else]`, toutefois il faut passer par une [variable référence](#). Cette variable référence contiendra la vue intégrée à afficher si la condition est fausse.

Avec `else`, la syntaxe équivalent à l'exemple précédent est:

```
<div *ngIf="condition else whenFalse">
    condition=true
</div>
<ng-template #whenFalse><div>condition=false</div></ng-template>
```

Dans cet exemple:

- `["condition else whenFalse"]` est la configuration de la directive `[ngIf]` sous la forme d'une microsyntaxe (voir [Configurer une directive structurelle par microsyntaxe](#) pour plus de détails).
- `#[whenFalse]` est la variable référence utilisée pour indiquer la vue intégrée à afficher quand la condition est fausse.

La forme étendue de la syntaxe avec `[else]` est:

```
<ng-template [ngIf]="condition" [ngIfElse]="whenFalse">
    <div>condition=true</div>
</ng-template>
<ng-template #[whenFalse><div>condition=false</div></ng-template>
```

Syntaxe avec `then` et `else`

On peut utiliser une syntaxe indiquant explicitement une clause `[then]`:

```
<div *ngIf="condition then whenTrue else whenFalse"></div>
<ng-template #[whenTrue><div>condition=true</div></ng-template>
<ng-template #[whenFalse><div>condition=false</div></ng-template>
```

Cette syntaxe utilise 2 variables références `[whenTrue]` et `[whenFalse]` correspondant aux vues intégrées à afficher suivant la valeur de la condition.

La forme étendue équivalente de la syntaxe est:

```
<ng-template [ngIf]="condition" [ngIfThen]="whenTrue" [ngIfElse]="whenFalse">
</ng-template>
<ng-template #[whenTrue><div>condition=true</div></ng-template>
<ng-template #[whenFalse><div>condition=false</div></ng-template>
```



ngFor

[ngFor] permet de parcourir une liste d'objets. Cette directive transpose la boucle [for] pour qu'elle soit utilisable dans le *template* d'un composant.

La syntaxe sous sa forme la plus simple si on l'utilise dans un élément HTML [div], est:

```
<div *ngFor="let nom variable Locale d'un élément of liste d'éléments">
  {{nom variable Locale d'un élément}}
</div>
```

L'intérêt est de pouvoir répéter l'affichage d'un élément HTML. Par exemple si on considère l'exemple suivant:

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

Si la liste d'éléments [items] contient les entiers [0], [1], [2], [3], [4] le résultat de l'exécution sera:

```
<div>0</div>
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
```

Le détail de la syntaxe utilisée est:

- [ngFor] correspond à une directive implémentée sous la forme d'un attribut d'un élément HTML car le paramètre [selector] de la directive est indiqué sous la forme [selector: '[ngFor]'] (voir [paramètre selector](#) pour plus de détails).
- La caractère [*] devant [ngFor] indique qu'il s'agit d'une [directive structurelle](#).
- Le code "[let item of items]" correspond à la configuration de la directive en utilisant une microsyntaxe (voir [Configurer une directive structurelle par microsyntaxe](#) pour plus de détails). Cette configuration comporte 2 expressions:
 - [let item] permettant de définir une variable locale nommée [item]. Cette variable est affectée de façon implicite (cf. [Propriété implicit](#)) par la directive lors de l'exécution de la boucle.
 - [of items]: cette expression permet d'affecter le contenu de la variable [items] au paramètre d'entrée (cf. [Paramètre inputs](#)) nommé [ngForOf] de la directive. [items] contient la liste des éléments qui seront répétés par la boucle.

Les caractères [;], [:] et le retour à la ligne sont facultatifs dans les expressions par microsyntaxe

Dans la suite d'expressions par microsyntaxe utilisées pour configurer la directive, les caractères [;], [:] et le retour à la ligne n'ont pas d'incidence. Si on considère l'exemple:

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

Les expressions suivantes sont équivalentes malgré la présence des caractères [;], [:] et du retour à la ligne:

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

Ou

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

La syntaxe indiquée plus haut est une forme compacte dont l'équivalent dans sa forme plus étendue est:

```
<ng-template [ngFor] let-item="$implicit" [ngForOf]="items">
  <div>{{item}}</div>
</ng-template>
```

Voir [\[ng-template\]](#) pour plus de détails.

Par exemple si on considère le composant suivant:

Template	<pre><p>Car types are:</p> <table> <tr> <th>Id</th> <th>Name</th> </tr> <tr *ngFor="let carType of carTypes"> <td>{{carType.id}}</td> <td>{{carType.name}}</td> </tr> </table></pre>
Classe du composant	<pre>@Component({ ... }) export class NgForOfExampleComponent { carTypes = [{ id: "A", name: "Pick-Up" }, { id: "B", name: "Van" }, { id: "C", name: "Truck" }, { id: "D", name: "Sedan" }, { id: "E", name: "Cabriolet" },]; }</pre>

Dans le fichier [\[.css\]](#) du composant, on ajoute le style:

```
table, td, th {
  border: 1px solid black;
}
```

La résultat sera du type:

Car types are:

Id	Name
A	Pick-Up
B	Van
C	Truck
D	Sedan
E	Cabriolet

Dans cet exemple, le tableau [\[carTypes\]](#) est parcouru de façon à répéter l'affichage d'une ligne pour chaque élément du tableau.

La forme étendue de la syntaxe indiquée plus haut est:

```
<ng-template [ngFor] let-carType="$implicit"[ngForOf]="carTypes">
<tr>
<td>{{carType.id}}</td>
<td>{{carType.name}}</td>
</tr>
</ng-template>
```

index, count, first, last, odd et even

D'autres propriétés dans [\[ngFor\]](#) peuvent être utilisées pour indiquer des informations supplémentaires:

- [\[index\]](#) indique l'index de l'élément courant dans la liste.
- [\[count\]](#) contient le nombre d'éléments de la liste.
- [\[first\]](#) est un booléen contenant [\[true\]](#) si l'élément courant est le premier élément de la liste.
- [\[last\]](#) est un booléen contenant [\[true\]](#) si l'élément courant est le dernier élément de la liste.

- `[odd]` contient `true` si l'index de l'élément courant est impair.
- `[even]` contient `true` si l'index de l'élément courant est pair.

Si on modifie l'exemple précédent en utilisant ces propriétés dans le *template*:

```
<p>Car types are:</p>
<table>
  <tr>
    <th>Index</th>
    <th>Id</th>
    <th>Car type</th>
    <th>Is first type?</th>
    <th>Is last type?</th>
    <th>Is odd index?</th>
    <th>Is even index?</th>
    <th>Type count</th>
  </tr>
  <tr *ngFor="let carType of carTypes
            index as typeIndex
            first as isFirstType
            last as isLastType
            odd as isOddIndex
            even as isEvenIndex
            count as typeCount">
    <td>{{typeIndex}}</td>
    <td>{{carType.id}}</td>
    <td>{{carType.name}}</td>
    <td>{{isFirstType}}</td>
    <td>{{isLastType}}</td>
    <td>{{isOddIndex}}</td>
    <td>{{isEvenIndex}}</td>
    <td>{{typeCount}}</td>
  </tr>
</table>
```

Dans les différentes expressions en microsyntaxe, la forme `[index as typeIndex]` permet d'affecter la valeur de la propriété `[index]` à une variable locale nommée `[typeIndex]`.

Le résultat de cette implémentation est:

Car types are:

Index	Id	Car type	Is first type?	Is last type?	Is odd index?	Is even index?	Type count
0	A	Pick-Up	true	false	false	true	5
1	B	Van	false	false	true	false	5
2	C	Truck	false	false	false	true	5
3	D	Sedan	false	false	true	false	5
4	E	Cabriolet	false	true	false	true	5

ngSwitch

`[ngSwitch]` est une directive permettant d'afficher un élément HTML si une expression est vraie. Cette directive transpose `[switch...case]` pour qu'il soit utilisable dans le *template* d'un composant.

La syntaxe sous sa forme la plus simple si on l'utilise dans un élément HTML `[div]`, est:

```
<div [ngSwitch]="<expression utilisée pour La comparaison>">
  <div *ngSwitchCase="<expression comparée 1>">...</div>
  <div *ngSwitchCase="<expression comparée 2>">...</div>
  ...
  <div *ngSwitchCase="<expression comparée N>">...</div>
  <div *ngSwitchDefault>...</div>
</div>
```

Ainsi:

- L'attribut `[ngSwitch]` permet d'indiquer l'expression qui sera utilisée pour effectuer la comparaison.
- `[ngSwitchCase]` va permettre d'indiquer l'expression pour laquelle l'égalité sera vérifiée.
- `[ngSwitchDefault]` correspond au cas par défaut si l'égalité n'a été trouvée pour aucune expression.

Par exemple si on considère le composant suivant:

Template	<pre><p>Vehicle size is:</p> <div [ngSwitch]="vehicleSize"> <div *ngSwitchCase="'big'">Big</div> <div *ngSwitchCase="'medium'">Medium</div> <div *ngSwitchCase="'little'"><i>Little</i></div> <div *ngSwitchDefault>(unknown)</div> </div></pre>
Classe du composant	<pre>@Component({ ... }) export class NgSwitchExampleComponent { vehicleSize='big' }</pre>

Le résultat sera du type:

Vehicle size is:

Big

Dans cet exemple, le membre `[vehicleSize]` dans la classe du composant contient la valeur `'big'`. Ainsi seul l'élément HTML `<div>Big</div>` sera affiché.

Il est possible d'utiliser une expression pour l'attribut `[ngSwitch]` mais aussi pour l'attribut `[ngSwitchCase]`. Par exemple si on considère le composant suivant:

Template	<pre><table> <tr> <th>Car type</th> <th>Size</th> </tr> <tr *ngFor="let carType of carTypes" [ngSwitch]="getVehicleSizeRange(carType.size)"> <td>{{carType.name}}</td> <td *ngSwitchCase="bigSize" class='big'>Big</td> <td *ngSwitchCase="mediumSize" class='medium'>Medium</td> <td *ngSwitchCase="littleSize" class='little'>Little</td> <td *ngSwitchDefault>(unknown)</td> </tr> </table></pre>
Classe du composant	<pre>@Component({ ... }) export class NgSwitchExampleComponent { vehicleSize='big'; carTypes = [{ id: 'A', name: 'Pick-Up', size:4}, { id: 'B', name: 'Van', size:6}, { id: 'C', name: 'Truck', size:7}, { id: 'D', name: 'Sedan', size:4}, { id: 'E', name: 'Cabriolet', size:3},]; bigSize='big'; mediumSize='medium'; littleSize='little'; getVehicleSizeRange(size: number): string { if (size < 4) return this.littleSize; else if (size >= 4 & size < 6) return this.mediumSize; else return this.bigSize; } }</pre>

On ajoute les styles suivants dans le fichier `style.css`:

```
table, td, th {
  border: 1px solid black;
}

.big {
  background-color: red;
}

.medium {
  background-color: orange;
}

.little {
  background-color: green;
}
```

Le résultat est du type:

Car type	Size
Pick-Up	Medium
Van	Big
Truck	Big
Sedan	Medium
Cabriolet	Little

Cet exemple est plus élaboré et permet de montrer qu'une expression est utilisée à la fois pour `ngSwitch` et les différentes clauses `ngSwitchCase`.

ngPlural

`ngPlural` permet d'apporter une solution pour la gestion des pluriels. Suivant le nombre d'éléments d'une liste, il va permettre d'afficher une forme au singulier ou au pluriel, par exemple:

- ["Un élément"]
- ["Des éléments"]
- ["Aucun élément"]
- Etc..

`ngPlural` donne la possibilité d'indiquer toutes les formes possibles en utilisant l'attribut `ngPluralCase`:

```
<p [ngPlural]="<nom d'éléments>">
  <ng-template ngPluralCase="0">Aucun élément</ng-template>
  <ng-template ngPluralCase="1">Un élément</ng-template>
  <ng-template ngPluralCase="other">Des éléments</ng-template>
</p>
```

Ainsi:

- `ngPlural` est utilisé pour indiquer le nombre d'éléments.
- `ngPluralCase` permet d'indiquer les différents cas de figures.

On ne peut pas utiliser n'importe quelle valeur avec `ngPluralCase`. La liste des valeurs acceptées est:

- Directement des valeurs numériques:** ["0"], ["1"], ["2"] et ["other"] pour indiquer le cas par défaut.
- Des expressions comportant une valeur précédée de [=]** [= "other"] peut être utilisé pour indiquer le cas par défaut.

Par exemple:

```
<p [ngPlural]="itemCount">
  <ng-template ngPluralCase="=0">Aucun élément</ng-template>
  <ng-template ngPluralCase="=1">Un élément</ng-template>
  <ng-template ngPluralCase="other">Des éléments</ng-template>
</p>
```

- Des valeurs correspondants aux CDLR (i.e. Common Locale Data**

Repository): ces valeurs sont:

- ["zero"] pour indiquer aucun élément.
- ["one"] pour indiquer exactement un élément.
- ["two"] pour indiquer exactement 2 éléments.
- ["few"] pour indiquer entre 3 et 10 éléments.
- ["many"] pour indiquer entre 11 et 99 éléments.
- ["other"] pour les autres cas de figure.

Les valeurs prises en compte dépendent du paramètre de langue. Par exemple pour le français ou l'anglais, il n'existe qu'une forme singulière ou une forme plurielle. Donc seules les valeurs ["one"] ou ["other"] sont utilisées. Les valeurs ["zero"], ["two"], ["few"] et ["many"] ne sont pas prises en compte.

Dans le cas où aucune valeur ne peut satisfaire le nombre d'élément évalué, une erreur du type suivant peut se produire:

```
Error: No plural message found for value "..."
```

Par exemple, si on considère l'exemple du composant suivant:

Template	<pre> <ng-template ngPluralCase="=0">aucun élément</ng-template> <ng-template ngPluralCase="=1">un élément</ng-template> <ng-template ngPluralCase="=2">deux éléments</ng-template> <ng-template ngPluralCase="other">des éléments</ng-template> . <p><button (click)="addItem()">Add item</button></p> <p>{{ items.length }}</p> </pre>
Classe du composant	<pre> import { Component, OnInit, Inject } from '@angular/core'; @Component({ ... }) export class NgPluralExampleComponent implements OnInit { items: number[]; ngOnInit(): void { this.items = []; } addItem(): void { this.items.push(Math.random()); } } </pre>

Ce composant permet de montrer le comportement si on augmente le nombre d'éléments dans la liste en cliquant sur `Add item`.

Dans le cas d'une utilisation explicite des valeurs `"=0"`, `"=1"`, `"=2"` et `"other"`, les 4 formes seront affichées successivement:

- 0: `La liste contient aucun élément.`
- 1: `La liste contient un élément.`
- 2: `La liste contient deux éléments.`
- Au delà: `La liste contient des éléments.`

Le comportement est le même si on indique les valeurs `"0"`, `"1"`, `"2"` et `"other"`:

```

<ng-template ngPluralCase="0">aucun élément</ng-template>
<ng-template ngPluralCase="1">un élément</ng-template>
<ng-template ngPluralCase="2">deux éléments</ng-template>
<ng-template ngPluralCase="other">des éléments</ng-template>

```

En revanche si on utilise les valeurs `"zero"`, `"one"`, `"two"`, `"few"`, `"many"` et `"other"`, seules les valeurs `"one"` et `"other"` sont prises en compte:

```

<ng-template ngPluralCase="zero">aucun élément</ng-template>
<ng-template ngPluralCase="one">un élément</ng-template>
<ng-template ngPluralCase="two">deux éléments</ng-template>
<ng-template ngPluralCase="few">un peu d'éléments</ng-template>
<ng-template ngPluralCase="many">quelques éléments</ng-template>
<ng-template ngPluralCase="other">des éléments</ng-template>

```

L'affichage est:

- 0: `La liste contient des éléments.`
- 1: `La liste contient un élément.`
- Au delà: `La liste contient des éléments.`

En anglais ou en français, seules les formes singulières et plurielles sont prises en compte. Ce n'est pas le cas de toutes les langues. Dans le cas de l'arabe les autres formes peuvent être prises en compte.

Par exemple si on change les paramètres locaux de langues en effectuant les étapes suivantes:

1. Dans le module du composant (par exemple `app.module.ts`), si on modifie la langue pour choisir `"ar-AE"` correspondant à l'arabe des Emirats Arabes Unis:

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
...
import { LOCALE_ID } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeAr from '@angular/common/locales/ar-AE';

registerLocaleData(localeAr);

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [
    { provide: LOCALE_ID, useValue: 'ar-AE' }
  ],
  bootstrap: [AppComponent]
})

export class AppModule { }

```

2. On modifie le composant pour afficher le paramètre de langue:

```

import { Component, OnInit, Inject } from '@angular/core';
import { LOCALE_ID } from '@angular/core';

@Component({
  ...
})
export class NgPluralExampleComponent implements OnInit {
  items: number[];

  constructor(@Inject(LOCALE_ID) localeId) {
    console.log(localeId);
  }

  ngOnInit(): void {
    this.items = [];
  }

  addItem(): void {
    this.items.push(Math.random());
  }
}

```

A l'exécution, on peut voir que l'affichage se fait de cette façon:

- 0: [La liste contient aucun élément.]
- 1: [La liste contient un élément.]
- 2: [La liste contient deux éléments.]
- Si $n \% 100 = 3..10$: [La liste contient un peu d'éléments.]
- Si $n \% 100 = 11..99$: [La liste contient quelques éléments.]

Pour résumer...

Architecture d'un composant

Un composant est composé de:

- Une vue appelée **template** dans laquelle on implémente les éléments visibles.
- Une classe du composant implémentée en Typescript.
- Des métadonnées permettant d'apporter à Angular des informations supplémentaires concernant le composant.

La vue d'un composant correspond au **template**. Ce **template** peut être implémenté en utilisant du code HTML enrichi de façon à rendre la vue dynamique.

L'implémentation de cet aspect dynamique de la vue est possible grâce à différents mécanismes spécifiques à Angular:

- Le **binding** qui permet la mise à jour de données et l'exécution de fonctions à partir du déclenchement d'événements dans la vue.

- Des directives qui facilitent l'implémentation de comportements dans la vue.

A la création d'un composant:

- La classe du composant** se trouve dans un fichier `[.ts]`; cette classe doit comporter le décorateur `@Component()`:

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

@Component({
})
export class ExampleComponent {
```

- Le template** se trouve généralement dans un fichier séparé `[.html]`; le paramètre `templateUrl` dans le décorateur `@Component()` indique le chemin de ce fichier:

```
@Component({
  templateUrl: './example.component.html'
})
export class ExampleComponent {
```

Le *template* peut aussi être implémenté directement dans le fichier de la classe du composant en utilisant le paramètre `template`:

```
@Component({
  template: `<p>Contenu du template</p>`
})
export class ExampleComponent {
```

- Les métadonnées** sont renseignées dans des paramètres dans le décorateur `@Component()`. Les paramètres les plus courants sont:
 - `selector` indiquant où la vue sera affiché dans l'application
 - `styleUrls` qui est un tableau indiquant les chemins des fichiers CSS permettant de définir des styles ou des classes CSS utilisés dans la vue du composant.

Ces paramètres sont facultatifs:

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

Paramètre `selector`

Le paramètre `selector` peut être utilisé si l'affichage du composant n'est pas géré par un *router* Angular. La valeur de ce paramètre peut être utilisée pour indiquer où la vue du composant sera rendue dans l'application.

Par exemple, si la valeur du paramètre `selector` est `'app-example'` alors la vue du composant sera rendue si un autre composant contient dans son *template*:

```
<app-example></app-example>
```

D'autres conditions sont possibles pour afficher le composant:

- Si le paramètre `selector` est `[selector: '[app-example]']` alors le composant sera affiché dans l'élément HTML contenant l'attribut `app-example`, par exemple:

```
<span app-example></span>
```

- Si le paramètre `[selector]` est `[selector: '.app-example']` alors le composant sera affiché dans l'élément HTML dont la classe CSS est `[app-example]`, par exemple:

```
<span class="app-example"></span>
```

Binding

Il existe différents types de *bindings* correspondant à des interactions différentes entre le *template* et la classe du composant:

Interpolation

Exécution d'une expression Typescript contenant des membres ou des fonctions publics dans la classe du composant:

Template	<code><div>{{textToDisplay}}</div></code>
Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { textToDisplay = 'Texte à afficher'; }</pre>

Property binding

Permet de mettre à jour le contenu d'une propriété DOM d'un élément affiché avec la valeur d'un membre dans la classe du composant:

Template	<code><div [innerText]="textToDisplay"></div></code>
Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { textToDisplay = 'Texte à afficher'; }</pre>

Attribute binding

Permet de mettre à jour le contenu d'un attribut d'un élément HTML avec la valeur d'un membre dans la classe du composant:

Template	<code><input type="text" [attr.value]="textToDisplay" /></code> ou <code><input type="text" attr.value="{{textToDisplay}}" /></code>
Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { textToDisplay = 'Texte à afficher'; }</pre>

Event binding

Déclenche l'exécution d'une fonction dans la classe du composant à partir du déclenchement d'un événement dans un objet du DOM:

Template	<code><p>{{valueToDisplay}}</p> <button (click)="incrementValue()">Increment</button></code>
Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { valueToDisplay = 0; incrementValue(): void { this.valueToDisplay++; } }</pre>

Two-way binding

Permet de renseigner la valeur d'une propriété d'un composant enfant et d'être notifié des changements de valeur de cette propriété de façon à mettre à jour le membre d'un composant parent:

- Composant enfant:

Template	<pre><button (click)='incrementValue()>Increment</button></pre>
Classe du composant	<pre>@Component({ selector: 'counter', templateUrl: './counter.component.html' }) export class CounterComponent { @Input() count: number; @Output() countChange: EventEmitter<number>= new EventEmitter<number>(); incrementValue(): void { this.count++; this.countChange.emit(this.count); } }</pre>
• Composant parent:	

Variable référence

Une variable référence permet de nommer un élément HTML dans le *template* de façon à utiliser des propriétés de cet élément dans d'autres parties du *template*:

- Référencer un élément HTML (contenu statique):

Template	<pre><p><input #inputElement type='text' value="Valeur initial"/></p> <p>Contenu de input: {{inputElement.value}}</p></pre>
Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { }</pre>
• Accéder à une variable référence à partir de la classe du composant:	

- Accéder à une variable référence à partir de la classe du composant:

Template	<pre><div #divElement>Texte à afficher</div></pre>
Classe du composant	<pre>@Component({ ... }) export class ExampleComponent implements OnInit { @ViewChild('divElement', {static: true}) htmlDivElement: ElementRef; ngOnInit(): void { console.log(this.htmlDivElement.nativeElement.innerText); } } Ou @Component({ ... }) export class ExampleComponent implements AfterViewInit { @ViewChild('divElement', {static: false}) htmlDivElement: ElementRef; ngAfterViewInit(): void { console.log(this.htmlDivElement.nativeElement.innerText); } }</pre>

Directives usuelles

Ces directives permettent d'implémenter des comportements dans le *template*:

ngIfPermet d'implémenter l'équivalent d'un `[if...then...else]`:

- Équivalent de `[if...then:]`:

Template	<code><div *ngIf="condition">Afficher si vrai</div></code>
----------	--

Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { condition = true; }</pre>
---------------------	--

- Équivalent de `[if...then...else:]`:

Template	<code><div *ngIf="condition else whenFalse">Afficher si vrai</div></code> <code><ng-template #whenFalse><div>Afficher si faux</div></ng-template></code>
----------	--

Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { condition = true; }</pre>
---------------------	--

ngForPermet d'implémenter l'équivalent d'une boucle `[for]`:

Template	<code><div *ngFor="let item of items">{{item}}</div></code>
----------	---

-
- | | |
|---------------------|---|
| Classe du composant | <pre>@Component({ ... }) export class ExampleComponent { items = [0, 1, 2, 3, 4]; }</pre> |
|---------------------|---|
- Avec une propriété `[index]` contenant l'index de l'élément courant:

Template	<code><div *ngFor="let item of items index as itemIndex"></code> <code> Index {{itemIndex}}: {{item}}</code> <code></div></code>
----------	---

Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { items = [0, 1, 2, 3, 4]; }</pre>
---------------------	---

Les autres propriétés disponibles sont:

- `[count]` contient le nombre d'éléments de la liste.
- `[first]` est un booléen contenant `true` si l'élément courant est le premier élément de la liste.
- `[last]` est un booléen contenant `true` si l'élément courant est le dernier élément de la liste.
- `[odd]` contient `true` si l'index de l'élément courant est impair.
- `[even]` contient `true` si l'index de l'élément courant est pair.

ngSwitchPermet d'implémenter l'équivalent de `[switch...case]`:

Template	<code><p>La taille est:</p></code> <code><div [ngSwitch]="size"></code> <code> <div *ngSwitchCase="'little'">Petite</div></code> <code> <div *ngSwitchCase="'medium'">Moyenne</div></code> <code> <div *ngSwitchCase="'great'">Grande</div></code> <code> <div *ngSwitchDefault>(inconnue)</div></code> <code></div></code>
----------	---

Classe du composant	<pre>@Component({ ... }) export class ExampleComponent { size = 'big' }</pre>
---------------------	---

Références

- *Introduction to components and templates:*
<https://angular.io/guide/architecture-components>
- *Angular 5 Interpolation, Property Binding & Event Binding Tutorial:*
<https://coursetro.com/posts/code/108/Angular-5-Interpolation,-Property-Binding-&-Event-Binding-Tutorial>
- *Property & Event Binding:*
<https://codecraft.tv/courses/angular/quickstart/property-and-event-binding/>
- *Can't bind to 'ngModel' since it isn't a known property of 'input':*
<https://stackoverflow.com/questions/38892771/cant-bind-to-ngmodel-since-it-isnt-a-known-property-of-input>
- *Two-way Data Binding in Angular:*
<https://blog.thoughttram.io/angular/2016/10/13/two-way-data-binding-in-angular-2.html>
- *Angular NgModel Directive: NgModel in Angular:*
<https://appdividend.com/2018/10/03/angular-n.setModel-directive-example-tutorial/>
- *Passing data into Angular components with @Input :*
<https://ultimatecourses.com/blog/passing-data-angular-2-components-input>
- *Component events with EventEmitter and @Output in Angular :*
<https://ultimatecourses.com/blog/component-events-event-emitter-output-angular-2>
- *Code source nglf:*
https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_if.ts
- *Nglf: <https://angular.io/api/common/Nglf>*
- *Angular 11 NgSwitch Directive Tutorial with Examples:*
<https://www.positronx.io/angular-8-ngswitch-directive-tutorial-with-examples/>
- *NgPlural: <https://angular.io/api/common/NgPlural>*
- *9 Most Popular Directives Used In Angular Development :*
<https://www.cmarix.com/blog/9-most-popular-directives-used-in-angular-development/>
- *Angular ng-template, ng-container and ngTemplateOutlet – The Complete Guide To Angular Templates:* <https://blog.angular-university.io/angular-ng-template-ng-container-ngtemplateoutlet/>
- *ngPlural & ngPluralCase : Pluralization in Angular:*
<https://www.angularjswiki.com/angular/ngplural/>
- *Localizing your app:* <https://next.angular.io/guide/i18n#setting-up-locale>
- *Missing locale data for the locale “XXX” with angular:*
<https://stackoverflow.com/questions/46419026/missing-locale-data-for-the-locale-xxx-with-angular/48849230>
- *Change default Locale in angular:*
<https://stackoverflow.com/questions/61010958/change-default-locale-in-angular>
- *angular/packages/common/locales/ on Github:*
<https://github.com/angular/angular/tree/master/packages/common/locales>
- *Localizing your app:* <https://angular.io/guide/i18n>
- *Angular – Correct singular/plural form of a noun using custom pipe or NgPlural:* <https://trungk18.com/experience/angular-pipe-singular-plural/>
- *DOM Property vs HTML Attribute in Property Binding:*
<https://dev.to/deerawan/dom-property-vs-html-attribute-in-property-binding-48e4>
- *https://la-cascade.io/le-dom-cest-quoi-exactement/:* <https://la-cascade.io/le-dom-cest-quoi-exactement/>
- *What is the difference between properties and attributes in HTML?:*
<https://stackoverflow.com/questions/6003819/what-is-the-difference-between-properties-and-attributes-in-html>

Angular, Typescript, Web

attribute binding, event binding, interpolation, ngfor, ngif, ngmodel, ngplural, ngswitch, property binding, reference variable, selector, template, two-way binding

Previous Post:

[Fonctionnalités C# 8.0](#)

Next Post:

[Appliquer des styles CSS aux vues Angular](#)

Leave a Reply

Name (required)

Email (required)

Website URL

Enregistrer mon nom, mon e-mail et mon site dans le navigateur pour mon prochain commentaire.

[Post Comment](#)

Quelques découvertes, trucs et astuces sur .NET en général - Blog .NET © 2023