

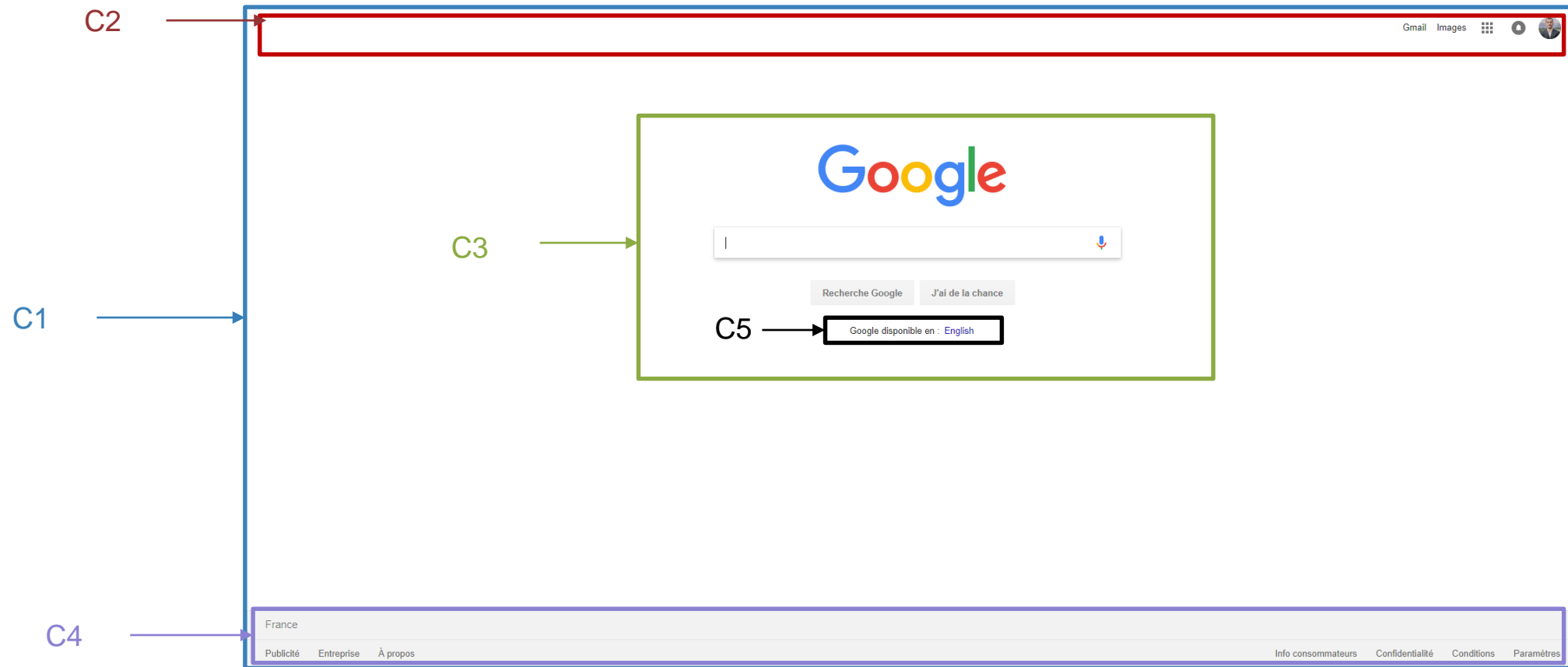
Components

Animé par Mazen Gharbi

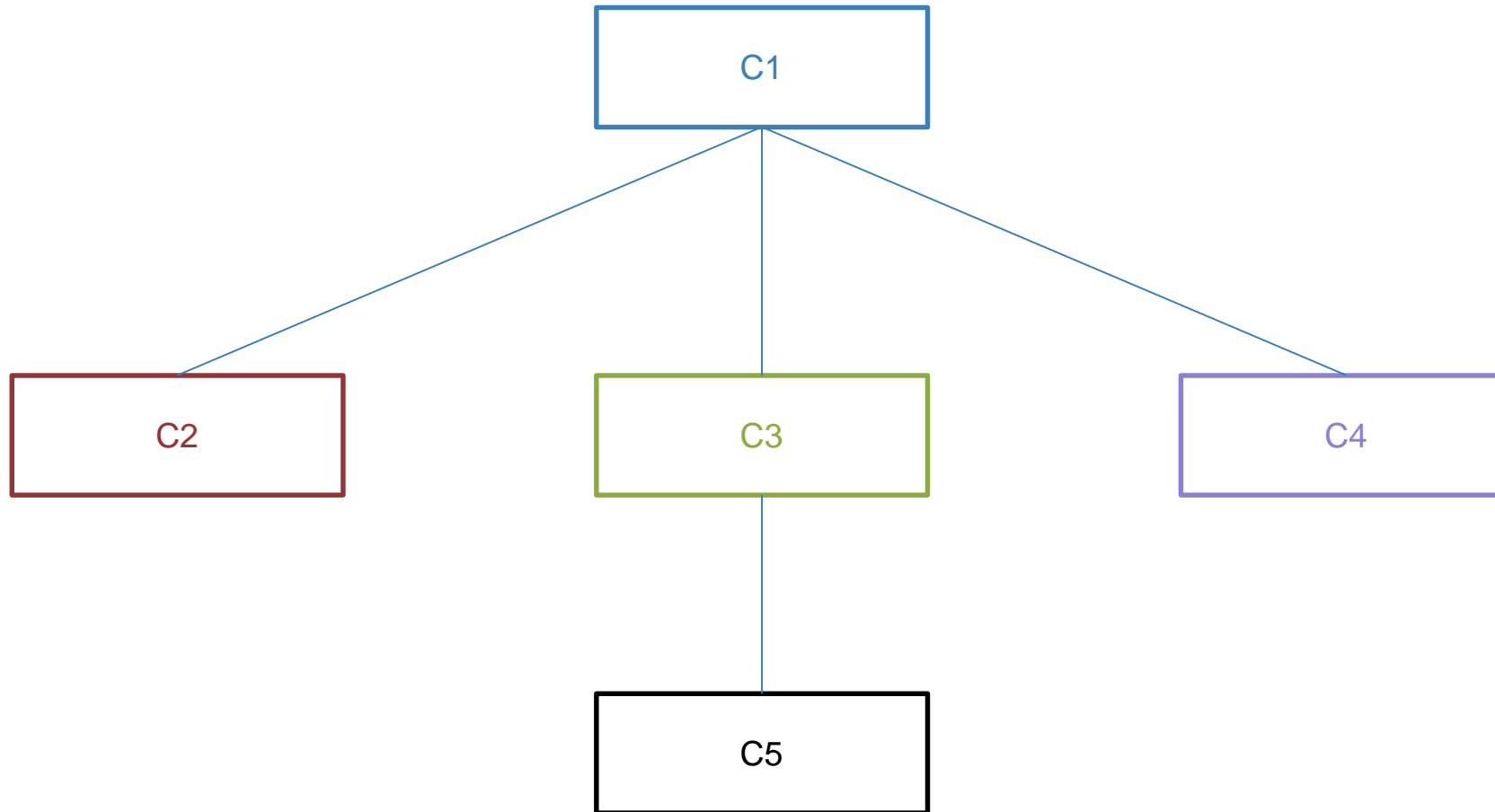
Les piliers du framework

- ▷ **Un composant contrôle une vue ou une partie d'une vue**
- ▷ L'un de principaux concepts d'Angular est de voir une application comme une arborescence de composants.
- ▷ Les composants permettent une meilleure décomposition de l'application, facilitent le refactoring et le testing.
- ▷ Chaque composant est isolé des autres composants. Il n'hérite pas implicitement des attributs des composants parents.

Séparation par composants



Séparation par composants



Notre première application

▷ Rien de mieux qu'un exemple

<https://stackblitz.com/edit/components-first-application>

Macademia

Bonjour Macademia ! Comment tu vas?

La porte d'entrée

On veut lancer l'application Angular sur un navigateur !

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Notre module codé dans app.module.ts



main.ts

Appel du module racine

Module racine

Permet d'utiliser les fonctionnalités basiques Angular

app.module.ts

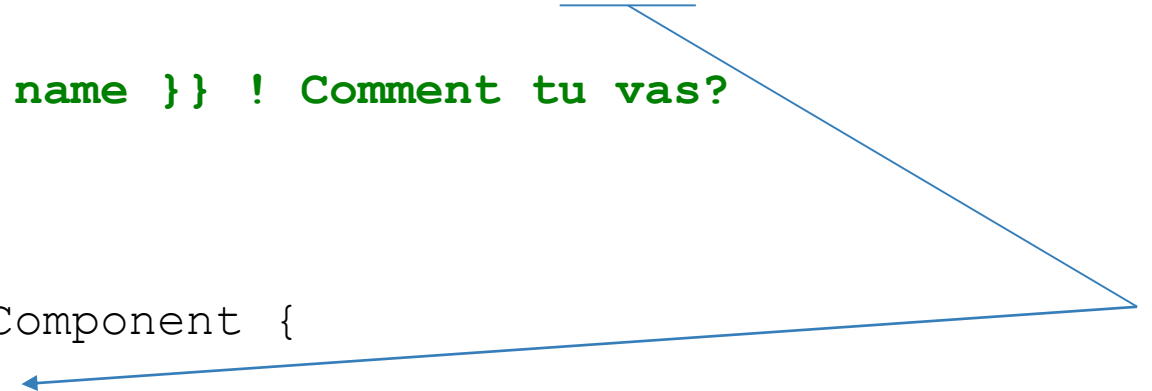
```
@NgModule ( {  
  imports: [  
    BrowserModule,  
    FormsModule  
  ],  
  declarations: [  
    AppComponent,  
  ],  
  bootstrap: [AppComponent] // Notre composant RACINE !  
})  
export class AppModule { }
```

Ajoute une surcouche aux inputs et permet l'utilisation de [(ngModel)]

Composant racine (bootstrap)

```
@Component ({
  selector: 'notre-application', // Identifiant unique !
  template: `
    <input type="text" [(ngModel)]="name">
    <div>
      Bonjour {{ name }} ! Comment tu vas?
    </div>
  `
})
export class AppComponent {
  public name;

  constructor() {
    this.name = 'Macademia';
  }
}
```

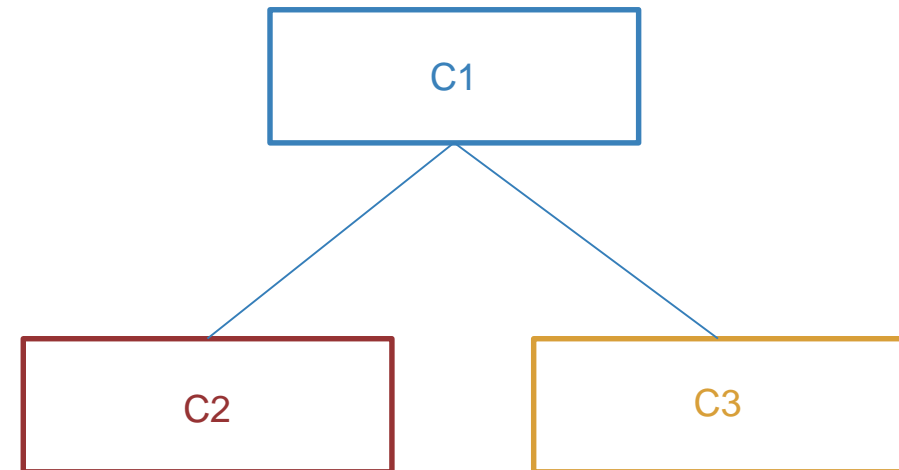
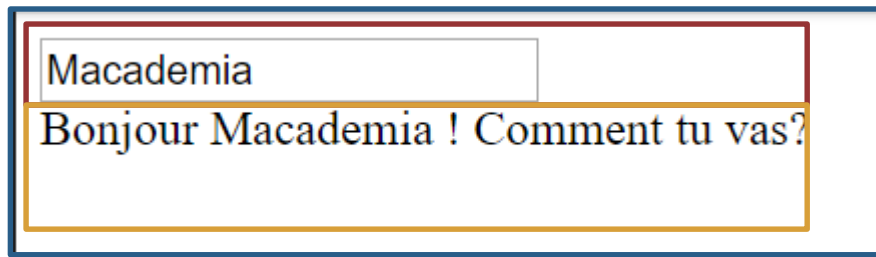


Quelques bonne pratiques

- ▷ Suffixez vos composants avec 'Component' ;
- ▷ Dashcase pour les noms de fichiers ;
- ▷ UpperCamelCase pour les noms de classes ;
- ▷ Préfixez le "selector" avec un identifiant propre à votre produit pour éviter les collisions ;
- ▷ Préférez l'utilisation de templateUrl et styleUrls si le contenu est trop volumineux ;

Créons une structure arborescente

On va tenter de découper notre application précédente en plusieurs composants



<https://stackblitz.com/edit/components-first-application-2>

Envoie de paramètres

app.component.ts

```
@Component ({  
  selector: 'notre-application',  
  template: `  
    <app-ask-name [user]="myUser"></app-ask-name>  
    <app-display-name [user]="myUser"></app-display-name>  
  `,  
})
```

Appel du composant enfant

Envoie d'un paramètre dont l'identifiant est « user » et la valeur est la variable « myUser » déclaré dans le modèle (classe du composant)

Réception du paramètre

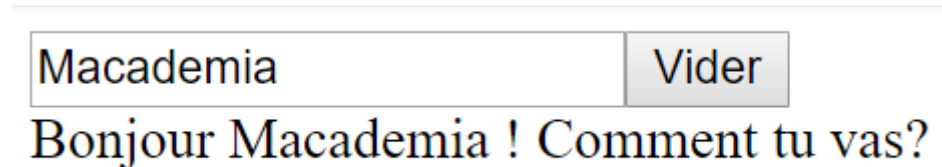
ask-name.component.ts

```
export class AskNameComponent implements OnInit {  
  @Input() user;  
  
  constructor() {}  
  
  ngOnInit() {}  
}
```

Définit la variable en tant que conteneur près à recevoir une propriété du nom de « user »

Une demande de dernière minute

- ▷ Le client adore! Mais il aimerait ajouter un bouton reset pour vider le contenu de l'input ;



Macademia

Vider

Bonjour Macademia ! Comment tu vas?

 La directive « (click) » permet de réagir à l'évènement click sur l'élément auquel elle est appliquée

<https://stackblitz.com/edit/components-first-application-2-problem>

Une demande de dernière minute

```
@Component ({
  selector: 'app-ask-name',
  template: `
    <input type="text" [(ngModel)]="user.name" />
    <button (click)="clean()">Vider</button>
  `
})
export class AskNameComponent implements OnInit {
  @Input() user;

  constructor() { }

  ngOnInit() {
  }

  clean() {
    this.user = {
      name: '';
    };
  }
}
```

ask-name.component.ts

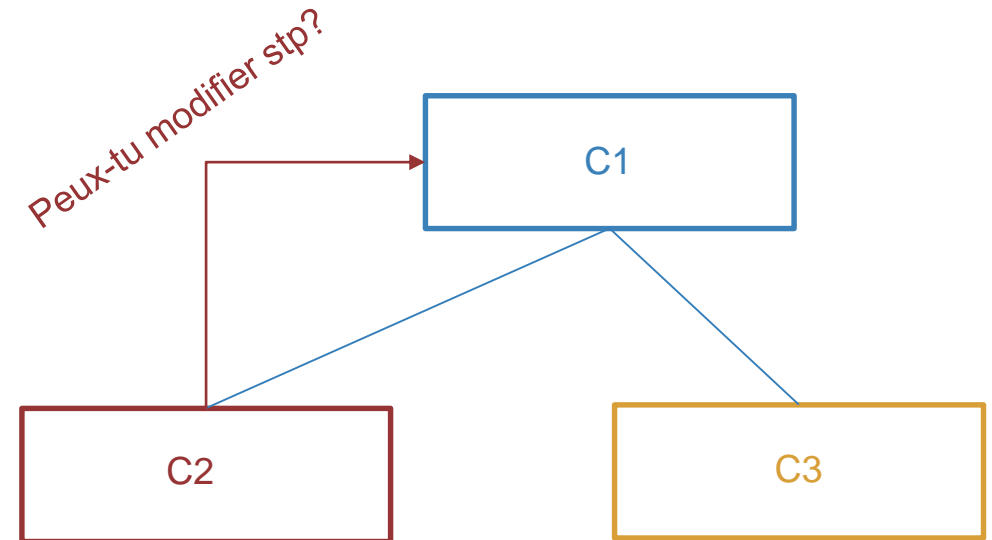
Permet de réagir à l'évènement utilisateur « click »

Ici, une nouvelle référence de « user » est créée

Pas de two-way binding en Angular !

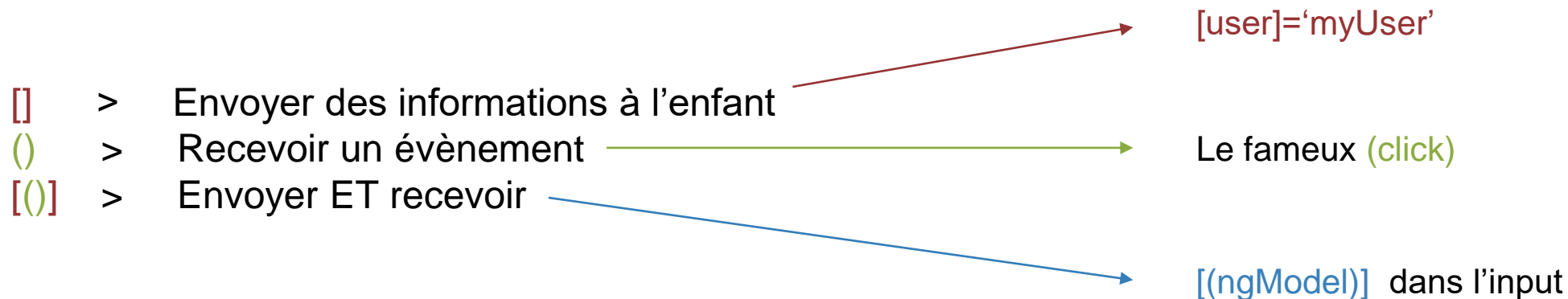
- ▷ L'édition de "user.name" fonctionnait car les trois composants partageaient une référence vers le même objet 'user' ;
- ▷ Les variables sont mises à jour du haut vers le bas, mais l'inverse est « impossible » ;

Ce serait bien d'avoir quelque chose du genre :



EventEmitter

- ▷ Les EventEmitters vont nous permettre d'envoyer un événement de l'enfant vers le père ;
- ▷ Le père décide de ce qu'il veut faire une fois l'évènement reçu
- ▷ Mais avant, petit retour sur la syntaxe Angular :



Mise en place

```
export class AskNameComponent implements OnInit {  
  @Input() user;  
  @Output() cleanText = new EventEmitter();  
}
```



Pour que cela fonctionne, il est nécessaire de créer l'eventEmitter directement lors de l'initialisation, sinon angular lève une erreur

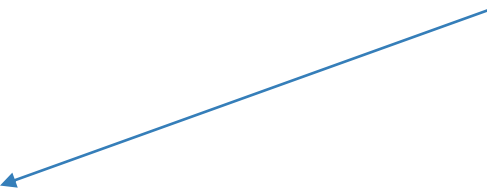
<https://stackblitz.com/edit/components-first-application-eventemitter>

Event Emitter

ask-name.component.ts

```
export class AskNameComponent implements OnInit {  
  @Input() user;  
  @Output() userChange = new EventEmitter();  
  
  constructor() { }  
  
  ngOnInit() {  
  }  
  
  public clean() {  
    this.userChange.emit({ name: '' });  
  }  
}
```

*On enclenche l'événement en envoyant un paramètre au père.
Le paramètre est l'objet « {name: ''} »*




Event Emitter

Réaction à l'évènement reçu par l'enfant, nous aurions également pu appeler une fonction du modèle (classe du composant)



app.component.html

```
<app-ask-name  
  [user]="myUser"  
  (userChange)="myUser = $event"></app-ask-name>  
  
<app-display-name [user]="myUser"></app-display-name>
```



Simuler le two-way binding

- ▷ S'il n'y a pas de two-way binding, comment [(ngModel)] fonctionne-t-il ?
- ▷ [(ngModel)] n'est qu'en fait qu'un raccourci de nommage! Voyons comment le mettre en place pour notre application

-  *La fonction « emit » de nos EventEmitter peut prendre un paramètre correspondant à la valeur que l'on souhaite envoyer au père. Ce paramètre est optionnel*
-  *Le père peut récupérer la valeur envoyée par un enfant avec le mot-clé « \$event »*

<https://stackblitz.com/edit/components-first-application-eventemitter-2waybinding>

Simuler le two-way binding

```
@Input() user;  
@Output() userChange = new EventEmitter();
```

ask-name.component.html

```
<app-ask-name  
  [user]="myUser"  
  (userChange)="myUser = $event"  
></app-ask-name>
```

↔ équivalent

```
<app-ask-name  
  [(user)]="myUser"  
></app-ask-name>
```

app.component.html

Directives structurelles

- ▷ Les directives structurelles modifient le DOM en ajoutant/supprimant des éléments du DOM ;
- ▷ Le préfixe '*' indique qu'il s'agit d'une directive structurelle.
- ▷ Voici les 2 plus importantes :

***ngIf**

<https://stackblitz.com/edit/macademia-components-ng-if>

***ngFor**

<https://stackblitz.com/edit/macademia-components-ng-for>

*ngIf

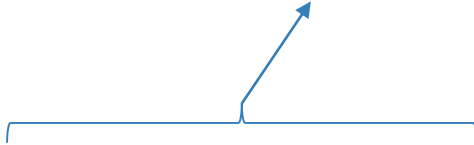
```
<input type="text" [(ngModel)]="name">  
  
<p *ngIf="name; else noName">  
  Salut {{name}}  
</p>  
  
<ng-template #noName>  
  <p>  
    Aucun nom rentré..  
  </p>  
</ng-template>
```



Le « else » n'est pas obligatoire dans le *ngIf. Si précisé, la variable créée doit obligatoirement être appliquée sur un ng-template

*ngFor

Après le « ; », on peut récupérer l'index de l'élément actuellement itérer dans la variable « i »



```
<p *ngFor="let ch of tab; let i = index">
  {{i}}
  - {{ch}}
  <span class="delete" (click)="removeChaine(i)">
    (delete)
  </span>
</p>
```


Smart et Dumb components

- ▷ Vous allez créer un nombre conséquents de composants ;
- ▷ Il va être nécessaire de catégoriser nos composants, il va exister 2 types : les Smart et les Dumb
- ▷ Les composants « Smart » sont des composants "High-Level" qui contrôlent la logique « business » ;
- ▷ Les composants « Dumb » ne contiennent pas de logique « business ». Ils se chargent principalement du design et doivent échanger les données avec les composants parents via les « Inputs/Outputs » du composant.

Style encapsulation

- ▷ Pour définir le style d'un composant, rien de plus simple :

```
@Component({  
  ...  
  styles: [`  
    div { font-weight: bold; }  
    span {text-align: center}  
  ` ,  
    h1 { font-size: 20px }  
  `]  
  ...  
})  
export class UserComponent {}
```

- ▷ Quelle est la portée du style appliquée?

Style encapsulation

```
@Component({  
  ...  
  encapsulation: ViewEncapsulation.Native  
  ...  
})  
export class UserComponent {}
```

- ▷ Emulated (défaut)
 - › Prend en compte le style global, le style ne s'applique pas aux enfants
- ▷ None
 - › Prend en compte le style global, le style s'applique aux enfants
- ▷ Native
 - › Ne prend pas en compte le style globale, le style ne s'applique pas aux enfants

Questions