Creation Application Web

Valeria BORDEI

La norme **HTML 5** a été finalisée en octobre **2014**. (HTML 4 date de 1997)

HTML 5 est une Révolution !

Il a permis de pouvoir créer des composants graphiques réutilisables et totalement indépendants. Et ces composants ont un nom :
les ***Web components***.

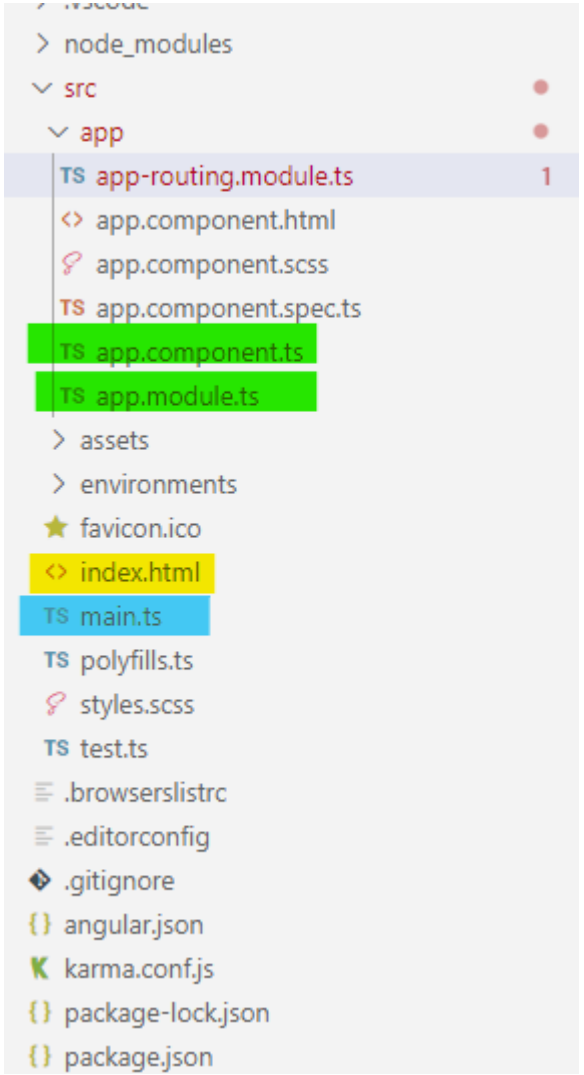
Pour parvenir à ce miracle, plusieurs technologies du W3C sont utilisées :
•*Les customs Elements* : API permettant de déclarer de nouveaux éléments HTML.
•*Les imports HTML* : permet d'importer des fichiers HTML dans un autre.
•*Les templates* : permet de charger une portion de HTML sans l'afficher tout de suite.
•*Le shadow DOM* : permet de masquer la complexité d'un composant et d'isoler plus facilement les composants entre eux via un système d'encapsulation.

C'est ce savant mélange qui permet la création de composants Web réutilisables et totalement étanches entre eux.

Ng serve

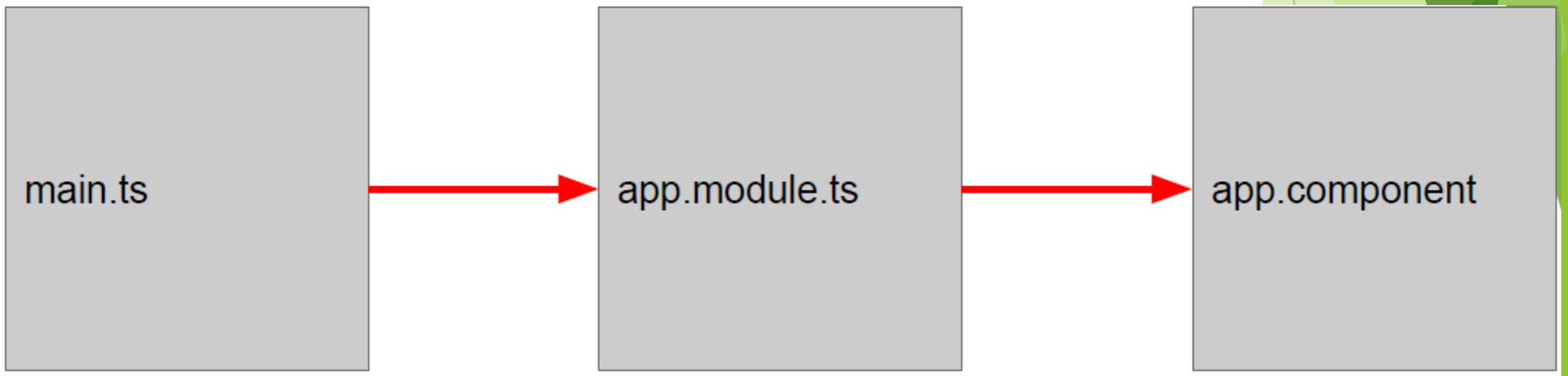
- `ng serve` / `npm run start` permet de lancer l'appli en local
  - L'appli s'ouvre par défaut dans un localhost (voir l'url dans le terminal, cela change selon la version d'angular)
  - On constate qu' `<app-root>` reste présent dans le DOM et qu'entre la balise `<app-root></app-root>` tout le html présent sur app-component.html est rendu dans de DOM

"root component" qui sera l'élément le plus haut de la hiérarchie de composants Angular.


node_modules
src
app
app-routing.module.ts
app.component.html
app.component.scss
app.component.spec.ts
app.component.ts
app.module.ts
assets
environments
favicon.ico
index.html
main.ts
polyfills.ts
styles.scss
test.ts
.browserslistrc
.editorconfig
.gitignore
angular.json
karma.conf.js
package-lock.json
package.json

L'ordre de chargement de l'application est le suivant :

index.html > main.ts > app.module.ts > app.component.ts



Le fichier *app.component.ts*

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

@Component({
  selector: 'wt-root’,
  template: '<p>Hello Word :  </p>’,
 // templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {}
```

Un espace réservé aux importations de code d'autres modules.
On charge la classe « `Component` » du module `@angular/core`
Qui est le module « cœur » d'Angular

Un composant doit au minimum importer l'annotation Component, bien sûr.

l'annotation `@Component` – (un décorateur) permet de définir un composant. Un composant doit au minimum comprendre deux éléments : `selector` et `template`:

- `selector` permet de donner un nom à notre composant afin de l'identifier par la suite.
- `template` permet de définir le code HTML du component

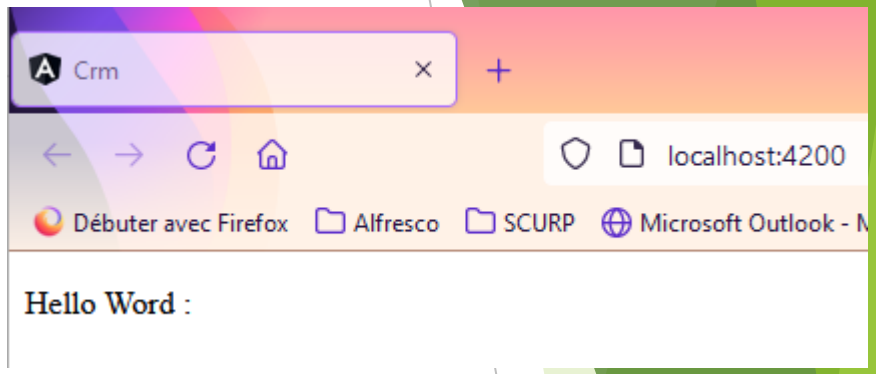
  On peut bien sûr définir notre template dans un fichier séparé avec l'instruction `templateUrl` à la place de template, afin d'avoir un code plus découpé et plus lisible.

Le fichier *app.component.ts* *(suite…)*

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

@Component({
  selector: 'wt-root', ’],
  template: '<p>Hello Word :  </p>’
 // templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss
})
```

```
export class AppComponent { }
```



la classe `AppComponent` contiendra la **logique** de notre composant. qui contrôle l'apparence et le comportement de notre Component.
Le mot-clef *export* permet de rendre le composant accessible pour d'autres fichiers

Par convention, on suffixe le nom des composants par Component : la classe du composant *app* est donc AppComponent.

app.module.ts le **module racine**

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
        imports: [ BrowserModule ],
        declarations: [ AppComponent ],
        bootstrap: [ AppComponent ] })

export class AppModule { }
```

BrowserModule est un module qui fournit des éléments essentiels pour le fonctionnement de l'application, comme les directives *ngIf* et *ngFor* dans tous nos *templates*

`@NgModule` - permet de déclarer un module:

- `imports` : permet de déclarer tous les éléments que l'on a besoin d'importer dans notre module. Les modules racines ont besoin d'importer le *BrowserModule* (contrairement aux autres modules que nous ajouterons par la suite dans notre application) ;
- `declarations` : une liste de tous les composants et directives qui appartiennent à ce module.
- `bootstrap` : permet d'identifier le composant racine, qu'Angular appelle au démarrage de l'application. Comme le module racine est lancé automatiquement par Angular au démarrage de l'application, et qu'*AppComponent* est le composant racine du module racine, c'est donc *AppComponent* qui apparaîtra au démarrage de l'application.

Le fichier *main.ts*

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Nous précisons donc que notre application est destinée aux navigateurs web, et que l'on désigne l'*AppModule* comme module racine, qui lui-même lancera l'*AppComponent*.

Le fichier *index.html* à la racine du projet **(et non dans le dossier app !)**

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Crm</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <wt-root></wt-root>
</body>
</html>
```

Ce fichier est une page HTML classique, qui contiendra toute notre application.

Pourquoi créer *main.ts*, *app.module.ts* et *app.component.ts* dans trois fichiers différents?

de mettre en place notre application de la bonne manière ; son démarrage est indépendant de la description de notre module, qui est également indépendant des composants qui le constituent. Ces trois éléments doivent donc être séparés !

Dans Angular, les components sont partout : l'application est un Component qui affiche des components contenant des components… Bref, Angular, c'est des components, components et des components !

En résumé:

•*SystemJS* est la bibliothèque par défaut choisie par Angular pour charger les modules.

•On a besoin au minimum d'un module racine et d'un composant racine par application.

•Le module racine se nomme par convention *AppModule*.

•Le composant racine se nomme par convention *AppComponent*.

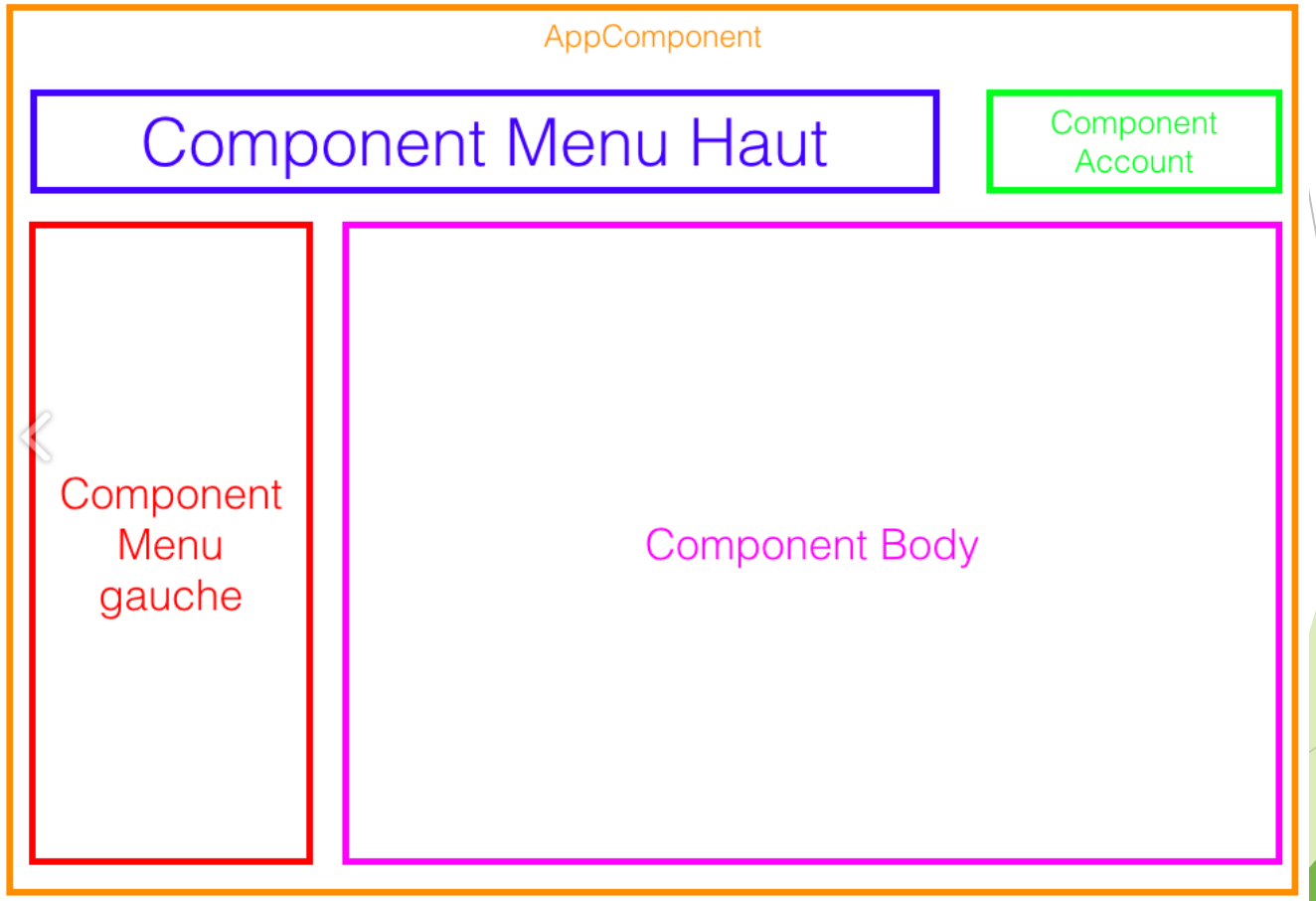
•L'ordre de chargement de l'application est le suivant :

index.html > main.ts > app.module.ts > app.component.ts.

•Le *package.json* initial est fourni avec des commandes prêtes à l'emploi comme la commande *npm start*, qui nous permet de démarrer notre application sans trop d'efforts.

Imbrication de Components

Une application classique Angular peut ressembler à ceci :



Le Data Binding

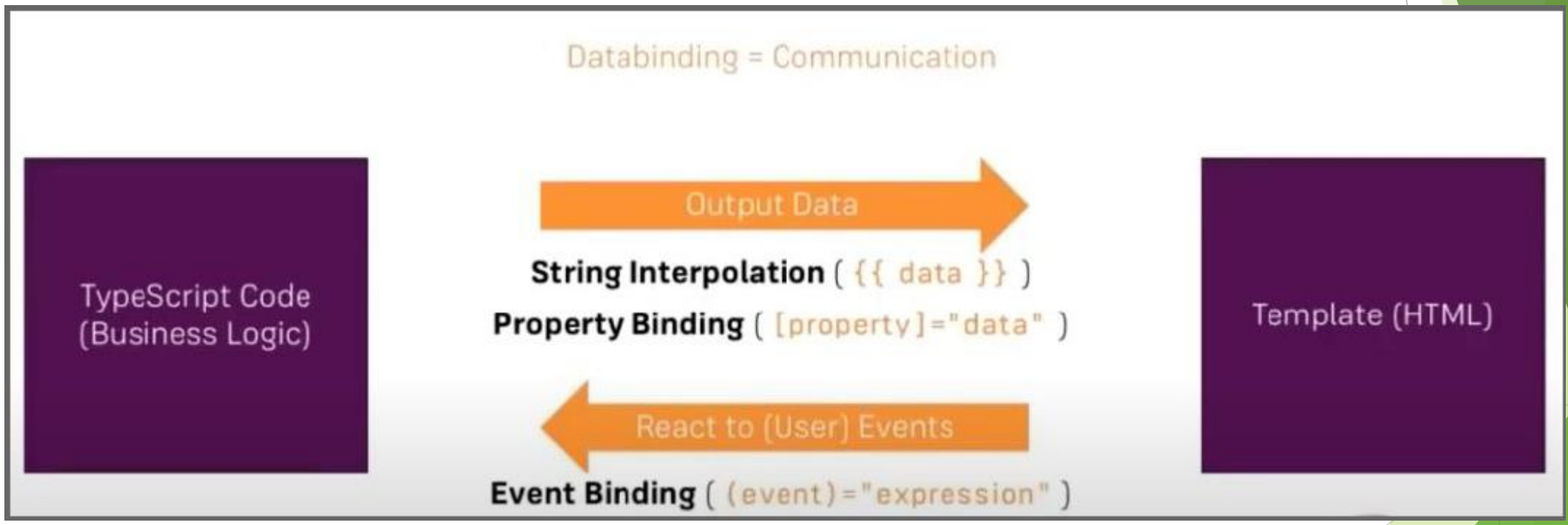
(liaison de données)

Le Data Binding est un élément essentiel dans les frameworks de Single Page Application. Il permet de synchroniser la vue au modèle JavaScript sous-jacent.

quatre sortes de Data Binding pour synchroniser le template et le component:

- **Interpolation** : permet de modifier le DOM à partir du modèle, si un changement est intervenu sur une valeur de ce dernier.
- **Property Binding** : permet de valoriser une propriété d'un composant ou d'une directive à partir du modèle, si un changement est intervenu sur une valeur de ce dernier.
- **Event Binding** : ce mécanisme permet d'exécuter une fonction portée par un component suite à un évènement émis par un élément du DOM.
- **Le "two-way" Data Binding** : c'est une combinaison du Property Binding et du Event Binding sous une unique annotation. Dans ce cas-là, le component se charge d'impacter le DOM en cas de changement du modèle et le DOM avertit le component d'un changement via l'émission d'un évènement.

Communication à l'intérieur d'un même composant



Data Binding

{{L'interpolation}} ce mécanisme permet de modifier le DOM à partir du modèle, si un changement est intervenu sur une valeur de ce dernier.

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

@Component( {
selector: 'app-root',
template: ` <div>Personne : {{person}} | Age : {{age}} | Adresse :
{{address}}</div> `
})

export class AppComponent {
      person:string= 'John Doe';
      age:number= 30;
      address:any= {street:'rue du Paradis', city:'75010 Paris'};
}
```

```
Personne : John Doe | Age : 30 | Adresse : [object Object]
```

{{address.street}}

La méthode `ngOnInit()` est appelée automatiquement par Angular au moment de la **création de chaque instance** du component. Elle permet d'initialiser des propriétés.

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

@Component( {
selector: 'app-root',
template: ` <div>Personne : {{person}} | Age : {{age}} | Adresse :
{{address}}</div> `
})
export class AppComponent implements OnInit{
       person!:string;
       age!:number;
       street!: string;

 ngOnInit() {
        this.person! = 'John Doe';
        this.age!= 30;
        this.street!'rue du Paradis';}
}
```

Data Binding

[Le Property Binding]

Le property Binding est également un mécanisme de Data Binding "one way". Tout comme l'interpolation, il permet de répercuter dans le DOM les valeurs des propriétés du component. L'annotation associée est **[...]** mais pour les réfractaires, il est possible de préfixer par `bind-` les propriétés. Le Property Binding est utilisable :

- Sur un élément du DOM. Par ex: `<img [src] ="…" />` ou `<img bind-src ="…" />` .
- Sur un attribut directive. Par ex: `<div [ngClass] ="…">` ou `<div bind-ngClass ="…">.`
- Sur la propriété d'un component. Par ex: `<page [color] ="…">` ou `<page bind-color ="…">` .

Le Property Binding sur un élément du DOM

avec l'interpolation:

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

@Component( {
selector: 'app-root',
template: ` <div align="{{alignement}}" >Personne : {{person}} | Age : {{age}} | Adresse :
{{address.street}}</div> `
})

export class AppComponent {
        person:string= 'John Doe';
        age:number= 30;
        address:any= {street:'rue du Paradis', city:'75010 Paris'};
        alignement:string = 'right';

}
```

avec le Property Binding

```
@Component({
selector: 'app-root',
template: ` <div [align]="alignement" >Personne : {{person}} | Age : {{age}} | Adresse :
{{address.street}}</div> ` })
```

Le Property Binding sur un attribute directive

un attribute directive est une directive modifiant le comportement ou l'apparence d'un élément

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

 @Component({
selector: 'app-root',
template: ` <div [align]="alignement" [ngStyle]="{color:couleur}">Personne
: {{person}} | Age : {{age}} | Adresse : {{address.street}}</div> ` })

 export class AppComponent {
        person:string= 'John Doe';
        age:number= 30;
        address:any= {street:'rue du Paradis', city:'75010 Paris'};
        alignement:string = 'right';
        couleur:string = 'red'; }
```

Le Property Binding sur une propriété d'un component

Crée un nouveau components : Ng g c comp1

```
import { Component } from '@angular/core';
 @Component({
selector: 'app-root',
template: ` <div [align]="alignement" [ngStyle]="{color:couleur}">Personne : {{person}} |
Age : {{age}} | Adresse : {{address.street}}</div>
<app-comp1 [monAdresse]="address"></app-comp1> `
})

export class AppComponent {
        person: string= 'John Doe';
        age: number= 30;
        address: any= {street: 'rue du Paradis', city: '75010 Paris'};
        alignement: string = 'right';
        couleur: string = 'red'; }
```

Comp1.component.ts

```
import {Component, Input} from '@angular/core';
 @Component({
        selector: 'app-comp1',
        template: ` {{monAdresse.street}} ` })

export class Comp1Component {
        @Input() monAdresse: any;
}
```

Grâce au décorateur `@Input`, le template de `Comp1Component` affiche la propriété `monAdresse.street` sans problème

(L'Event Binding) à partir d'un évènement du DOM, interagir avec un component

Pour réaliser ce Data Binding, Angular utilise les évènements ; d'où le nom de `Event Binding`.
Avec ce mécanisme, vous pouvez être averti des évènements utilisateurs tels que le clic, la frappe sur le clavier, le touch, …
La syntaxe est proche du Property Binding : à gauche de l'égalité, on retrouve l'évènement que l'on veut émettre entre parenthèses ou précédé de `on-`, et à droite, on trouve la fonction que l'on appellera lorsque le component interceptera l'évènement.

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

@Component({
          selector: 'app-root',
          template: ` <div [align]="alignement" [ngStyle]="{color:couleur}">Personne : {{person}} |
                    Age : {{age}} |     Adresse : {{address.street}}</div>
                    <app-comp1 [monAdresse]="address"></app-comp1>
          <button (click)="modifierPersonne()">Modifier adresse</button> `
})

export class AppComponent {
person: string= 'John Doe';
age: number= 30;
address: any= {street: 'rue du Paradis', city: '75010 Paris'};
alignement: string = 'right';
couleur: string = 'red';

          modifierPersonne() {
                    this.person = 'Another man'; }
}
```

créer vos propres `Event Binding` très simplement afin d'envoyer un évènement d'un component vers un component parent

- Crée un nouveau components : Ng g c comp2

comp2.component.ts

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

 @Component({
          selector: 'app-comp2',
          template: ` <button (click)="decrement()" > - </button>
                      <button (click)="increment()"> + </button> ` })

export class Comp2Component {
     counterValue: number = 0;

     @Output() counterChange = new EventEmitter();

     increment() {
          this.counterValue++;
          this.counterChange.emit({ value: this.counterValue });
                    }

     decrement() {
          this.counterValue--;
          this.counterChange.emit({ value: this.counterValue });
          }
}
```

`@Input` se charge de recevoir une valeur

`@Output` se charge d'envoyer une donnée

`counterChange` est une instance de classe `EventEmitter` en charge d'envoyer un évènement lors une pression sur un des boutons.

```
import {Component} from '@angular/core';
 @Component({
selector: 'app-root',
template: ` <div [align]="alignement"
[ngStyle]="{color:couleur}">Personne : {{person}} | Age : {{age}} |
Adresse : {{address.street}}</div> <app-comp1
[monAdresse]="address"></app-comp1>
<button (click)="modifierPersonne()">Modifier adresse</button>

<h1>Event binding - Compteur</h1>
<div> <app-comp2 (counterChange)="myValueChange($event);">
</app-comp2> </div>
 <br/> <div>Valeur récupérée : {{compteur}}</div> `
})

export class AppComponent {
person: string= 'John Doe'; age: number= 30; address: any=
{street: 'rue du Paradis', city: '75010 Paris'}; alignement:
string = 'right'; couleur: string = 'red'; compteur: any =
'N/A';

 myValueChange (event: any){ this.compteur = event.value; }

 modifierPersonne () { this.person = 'Another man'; } }
```

EventEmitter est une classe utilisée dans une directive ou un component pour émettre un évènement. Sa méthode principale est `emit(value? : T)` qui permet d'envoyer un évènement contenant potentiellement un objet ou une valeur.

la méthode `myValueChange` est appelée à chaque fois qu'un évènement `counterChange` est reçu. À noter la possibilité de passer `$event` en paramètre pour récupérer le contenu de l'évènement.

[(Le two-way Data Binding)]

Ce mécanisme permet, à partir d'une même notation, de modifier le modèle à partir du DOM et de modifier le DOM à partir du modèle

- Crée un nouveau components : Ng g c ma-taille

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

@Component({
selector: 'ma-taille',
template: ` <div>
          <button (click)="dec()" title="plus petit">-</button>
          <button (click)="inc()" title="plus grand">+</button>
          <label [style.font-size.px]="taille">FontSize: {{taille}}px</label> </div> `
          })

 export class maTailleComponent {
          @Input() taille: number;
          @Output() tailleChange = new EventEmitter<number>();

          dec() { this.resize(-1); }
          inc() { this.resize(+1); }
          resize(delta: number) {
                    this.taille = +this.taille + delta;
                    this.tailleChange.emit(this.taille);
}
 }
```

Le component aura un @Input nommé taille et un @output nommé tailleChange.

la règle du Data Binding 2-way : si l'on considère la notation [(x)], x est la propriété à setter (@Input) et xChange est l'évènement lancé lors d'une modification de sa valeur.
Cet évènement sera donc intercepté par son component parent afin de récupérer cette valeur.

Pour l'utiliser, il est nécessaire de déclarer, dans le component parent, une propriété `maTaille` qui sera bindée, puis de l'utiliser comme ceci :

```
<ma-taille [(taille)]="maTaille"></ma-taille>
<div [style.font-size.px]="maTaille">texte resizer</div>
```

Les pipes

un pipe reçoit des données en entrée, et les transforme dans le format de sortie souhaité.
Ce fonctionnement est valable, quel que soit le pipe que vous appliquez : pipe **avec ou sans paramètres**, **personnalisé** ou **natif**

le symbole « | » est l'opérateur des pipes.

```
{{ birthday | date }}
```

Les pipes natifs:

- **DatePipe** : permet de formater l'affichage d'une date ;
- **UpperCasePipe** : transforme le texte passé en entrée en majuscule ;
- **LowerCasePipe** : transforme le texte passé en entrée en minuscule ;
- **TitleCasePipe** : pour une phrase passée en paramètre, permet de mettre la première lettre de chaque mot en majuscule, et le reste en minuscule ;
- **CurrencyPipe** : prend en paramètre une devise pour mettre en forme un nombre comme monnaie ;
- **PercentPipe** : formate un nombre en tant que pourcentage. Par exemple, le chiffre 0.26 devient 26 % lorsque vous appliquez ce pipe ;
- **JsonPipe** : convertit la valeur passée en paramètre en chaîne JSON, grâce à la méthode *JSON.stringify*. Ce pipe peut être utile pour débugger votre code et retrouver plus facilement vos erreurs.

Les pipes

https://awesome-angular.developpez.com/tutoriels/angular-pipes/#LII-A