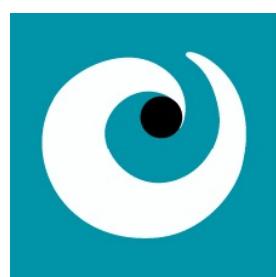


Angular, maîtriser le Framework Front-End de Google

KLEE GROUP

Du Mardi 02 au Vendredi 05 avril 2024



ORSYS
formation

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

Chapitre 1 : Introduction et Environnement



PRÉ-REQUIS

- ▶ HTML 5 / CSS 3 / Bootstrap



- ▶ JavaScript



- ▶ Programmation Orientée Objet

POO

CE QUI NOUS ATTEND...

Jour 1 :

- ▶ Présentation de Angular
- ▶ Introduction à Typescript
- ▶ Principe du SPA
- ▶ Installation de l'environnement
- ▶ Les Composants
- ▶ Interaction entre les composants
- ▶ Les Directives

Jour 2 :

- ▶ Les pipes
- ▶ Pipes pures vs Pipes impurs
- ▶ Les services et DI
- ▶ DI hiérarchique
- ▶ Routing et lazy loading
- ▶ Navigation
- ▶ Route Params et Query Params

CE QUI NOUS ATTEND...

Jour 3 :

- ▶ Template Driven Forms
- ▶ Reactive Forms
- ▶ Les observables
- ▶ Mise en place du Backend
- ▶ Effectuer des requêtes HTTP standards.

Jour 4 :

- ▶ Http Params et Http Headers
- ▶ Authentification et autorisations
- ▶ Intercepteurs
- ▶ Guards
- ▶ Tests unitaires avec Karma
- ▶ Déploiement sur GitHub Pages

PRÉSENTATION D'ANGULAR

- ▶ Angular est un framework (cadre de travail) Javascript
- ▶ Permet de développer des sites web, des applications web, des applications mobiles hybrides, de manière robuste et efficace.
- ▶ Supporte plusieurs langages : JS (ES5), TypeScript
- ▶ Développé par Google en 2009
- ▶ AngularJS vs Angular 2+
- ▶ Principaux concurrents : Vue.Js et React

PRÉSENTATION D'ANGULAR



Google



WebComponent



Observables



TypeScript

Microsoft

SITE WEB



Serveur



APPLICATION WEB

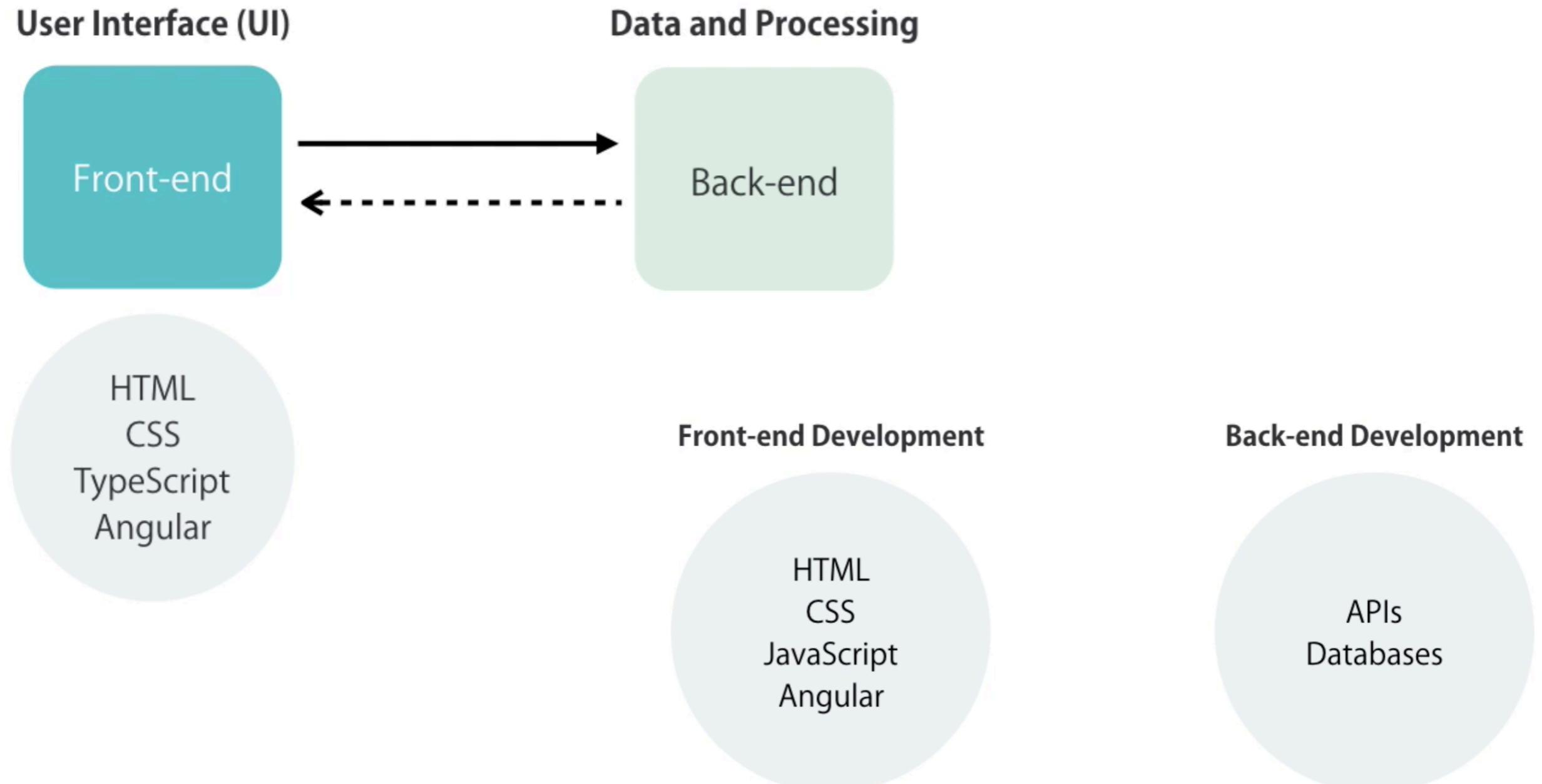
Intervention du JS



Serveur

Application web en SPA
(*Single Page Application*)

FRONT VS BACK



ILS UTILISENT ANGULAR



PHILOSOPHIE

- ▶ Angular est un **orienté composants**. Une application Angular est donc un ensemble de petits composants assemblés entre-eux.
- ▶ Un composant est l'assemblage d'un morceau de code HTML et d'une classe JavaScript dédiée à une tache particulière.
- ▶ Ces composants reposent sur le standard des **web components** (pas encore supporté par tous les navigateurs). Ce standard a été pensé pour découper une page web en fonction de leurs rôles (navbar, discussion, etc...).
- ▶ Un composant est censé être une partie qui fonctionne de manière autonome dans une application.

ECMASCRIPT 6 .. ECMASCRIPT 2015 .. ES6 .. JAVASCRIPT 6

- ▶ ECMAScript 6 est le nom de la dernière version standardisé de JavaScript.
- ▶ Cette standardisation a été approuvé par l'organisme de normalisation en Juin 2015.
- ▶ Le besoin d'une nouvelle standardisation s'est fait sentir pour fournir à JavaScript les moyens de développer des applications web robustes.
- ▶ ES6, via sa nouvelle syntaxe, apporte plusieurs nouvelles évolutions à JavaScript.

{ES6}

NOUVEAUTÉS DE ES6

- ▶ Les **classes**
- ▶ Le nouveau mot-clef **let** : Le mot-clé let permet de déclarer une variable locale, dans le contexte où elle a été assignée.
- ▶ Le nouveau mot-clef **const**
- ▶ Les **fonctions fléchées** : Les fonctions fléchées, ou arrow functions, ne sont pas des fonctions classiques, parce qu'elles ne définissent pas un nouveau contexte comme les fonctions traditionnelles. Les **paramètres** de fonctions **par défaut**
- ▶ Le **template string** : on peut désormais utiliser des templates strings, qui commencent et se terminent par un **backtick** `.

TYPESCRIPT

- ▶ « *TypeScript est un langage de programmation libre et open-source, développé par Microsoft, qui a pour but d'améliorer et de sécuriser la production de code JavaScript. C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code Javascript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript* ».
- ▶ Il est fortement recommandé d'utiliser TypeScript pour développer en Angular. C'est ce que Google recommande explicitement dans la documentation officielle d'Angular.
- ▶ En effet, Angular lui-même est développé avec TypeScript. C'est pour cela que nous utiliserons TypeScript dans ce cours.

ENVIRONNEMENT DE DÉVELOPPEMENT

- ▶ Commençons par installer la dernière version de Node depuis <https://nodejs.org/en/>
- ▶ Node : Environnement d'exécution pour programmes en JS à l'extérieur du navigateur.
- ▶ Vérifier l'installation en tapant : node --version
- ▶ Ceci nous permettra d'utiliser Node Package Manager (**NPM**) pour pouvoir utiliser des librairies tierces telle que **Angular CLI (Command Line Interface)**
- ▶ npm i -g @angular/cli
- ▶ ng version

ENVIRONNEMENT DE DÉVELOPPEMENT

- ▶ Pour créer un nouveau projet Angular : `ng new nom_projet`
- ▶ Rajouter `--no-standalone` si vous êtes avec une version ≥ 17
- ▶ Vérifions que votre projet s'exécute avec : `ng serve`
- ▶ Il nous reste à choisir un éditeur. Faites votre choix...

Visual Studio



Visual Studio 2017



Visual Studio Code



Visual Studio 2015

And More...



Sublime Text



Emacs



Atom



WebStorm



Eclipse



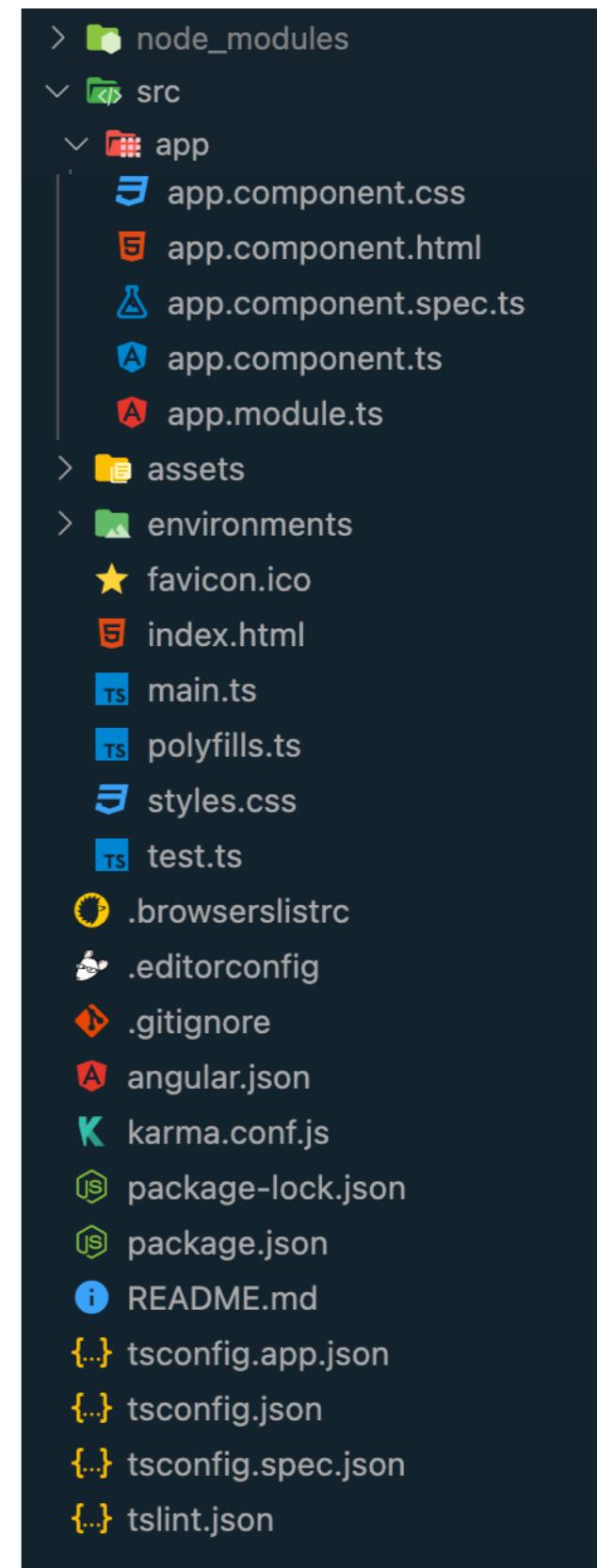
Vim

QUELQUES COMMANDES DU CLI

Commande	Utilisation
Component	ng g component my-new-component
Directive	ng g directive my-new-directive
Pipe	ng g pipe my-new-pipe
Service	ng g service my-new-service
Class	ng g class my-new-class
Interface	ng g interface my-new-interface
Module	ng g module my-module

ARBORESENCE D'UN PROJET

- ▶ **node_modules** : les dépendances déclarés dans package.json
- ▶ **src** : dossier contenant l'index, vos modules, le code source, les assets, les styles.
- ▶ **app** : Les sources du projet par défaut ainsi que le module appModule, le fichier app.component.ts ainsi que son template (app.component.html), son style (app.component.css) et app.component.spec.ts pour les tests unitaires.



ARBORESENCE D'UN PROJET

- ▶ 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 fichier `package.json` initial est fourni avec des commandes prêtes à l'emploi comme la commande `npm start` (ou `ng serve`), qui nous permet de démarrer notre application web.

WEBPACK

▶ Terminal



```
Initial Chunk Files | Names | Size
vendor.js          | vendor | 2.41 MB
polyfills.js       | polyfills | 141.30 kB
main.js            | main | 56.91 kB
runtime.js         | runtime | 6.15 kB
styles.css         | styles | 119 bytes

| Initial Total | 2.61 MB

Build at: 2020-12-25T21:53:33.971Z - Hash: 9f43cf2a88506a456fac - Time: 10246ms

** Angular Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **

✓ Compiled successfully.
✓ Browser application bundle generation complete.

Initial Chunk Files | Names | Size
styles.css          | styles | 119 bytes

4 unchanged chunks

Build at: 2020-12-25T21:53:34.791Z - Hash: 01f5bc998a2509ae21dd - Time: 561ms

✓ Compiled successfully.
```

▶ Navigateur



```
▼ <body>
  ▶ <app-root _nghost-c0 ng-version="7.1.4">...</app-root>
    <script type="text/javascript" src="runtime.js"></script>
    <script type="text/javascript" src="polyfills.js"></script>
    <script type="text/javascript" src="styles.js"></script>
    <script type="text/javascript" src="vendor.js"></script>
    <script type="text/javascript" src="main.js"></script>
  </body>
```

ANGULAR

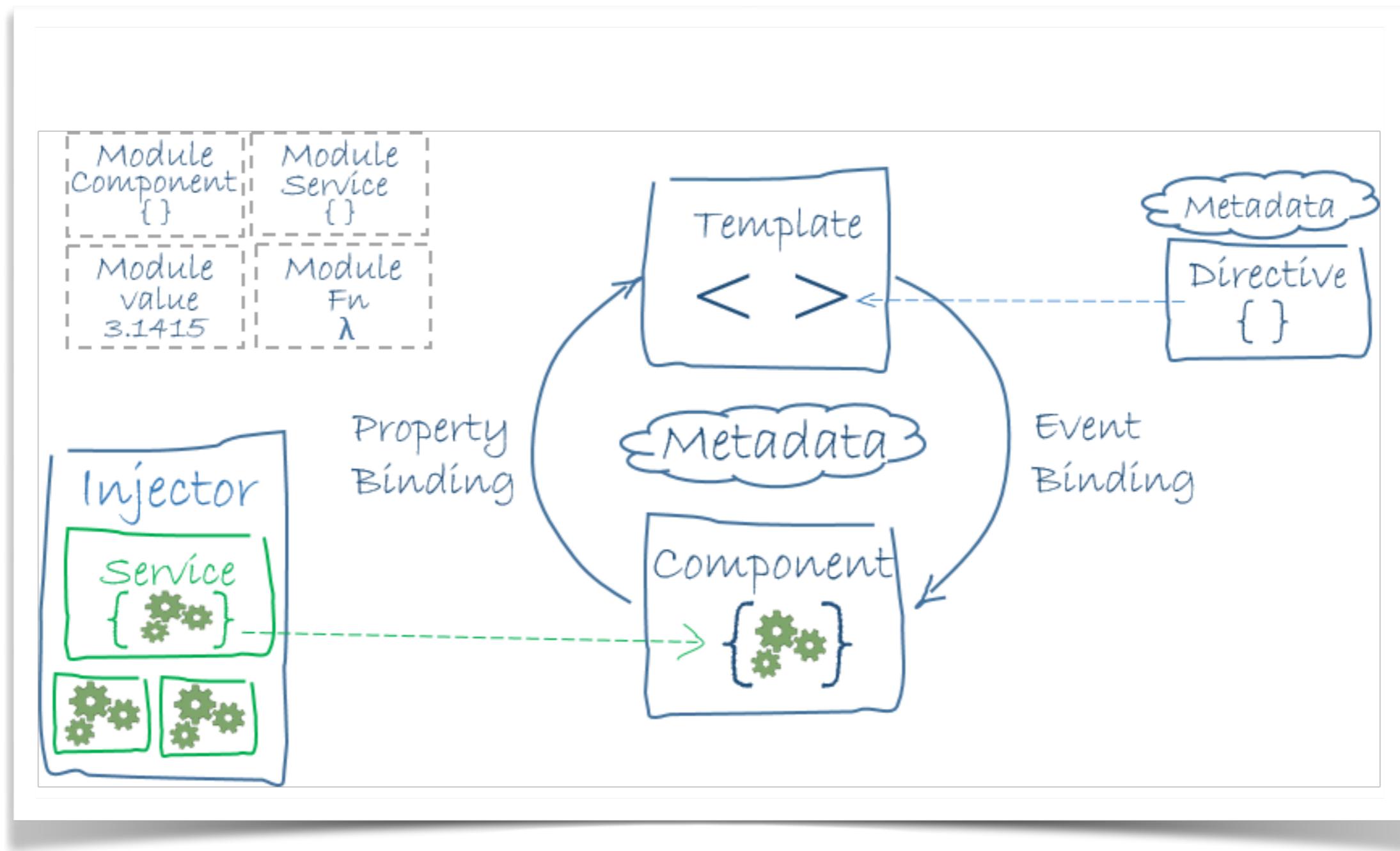


- ▶ Angular est **modulaire**
 - ▶ Chaque application va définir Angular Modules ou NgModules
 - ▶ Chaque module Angular est une classe avec une annotation `@NgModule`
 - ▶ Chaque application a au moins un module, c'est le module principale.
- ▶ Rapide
- ▶ Orienté Composants

ANGULAR

- ▶ Le module principal est le module qui permet de lancer l'application.
- ▶ L'annotation (decorator) **@NgModule** identifie AppModule comme un module Angular.
- ▶ L'annotation prend en paramètre un objet spécifiant à Angular comment compiler et lancer l'application.
 - ▶ **imports** : tableau contenant les modules utilisés.
 - ▶ **declarations** : tableau contenant les composants, directives et pipes de l'application.
 - ▶ **bootstrap** : indique le composant exécuter au lancement de l'application.
- ▶ Il peut y avoir aussi d'autres attributs dans cet objet comme provider

ANGULAR EN UN SCHÉMA



AJOUTER BOOTSTRAP À VOTRE PROJET

- ▶ On peut ajouter Bootstrap de plusieurs façons. La plus recommandée est la suivante :

1- Via npm avec la commande : `npm install bootstrap --save`

2- Ensuite, on ajoute dans le chemin des dépendances dans les tableaux styles et scripts dans le fichier angular.json:

```
"styles": [  
  "./node_modules/bootstrap/dist/css/bootstrap.min.css",  
  "styles.css",  
]
```

AJOUTER BOOTSTRAP À VOTRE PROJET

- ▶ 2ème méthode : Vous pouvez aussi ajouter dans le fichier src/style.css un import de vos bibliothèques.
 - @import "~bootstrap/dist/css/bootstrap.min.css";
- ▶ 3ème méthode : Ajouter l'appel au bootstrap donc le head de index.html
- ▶ Essayer la même chose avec *font-awesome* qu'on pourra également utiliser dans notre projet.

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

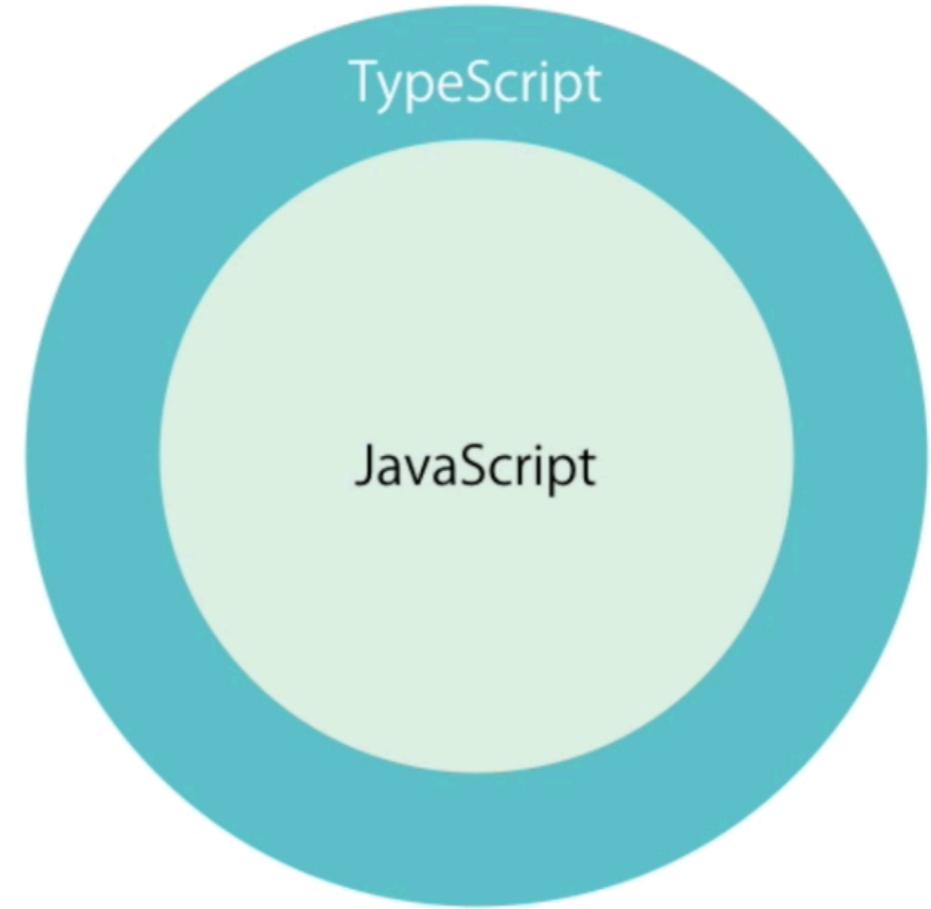
ANGULAR

Chapitre 2 : Introduction à Typescript 



TYPESCRIPT, C'EST QUOI ?

- ▶ Typescript est un sur-ensemble (superset) de Javascript.
- ▶ Ne peut pas être intégré directement à une page web.
- ▶ Permet de tirer partie des fonctionnalités modernes de ES6 comme les modules, les classes, les interfaces, fonctions fléchées, etc..
- ▶ Open source.



TYPESCRIPT, C'EST QUOI ?

- ▶ Doit être transpiler en Javascript.



- ▶ Un langage transcompilé est un langage pouvant être compilé en un autre langage.
- ▶ Autrement dit, on peut parler d'un langage en surcouche d'un langage existant.
- ▶ Amène une phase de compilation aux langages interprétés.

TYPAGE

- ▶ Typescript permet le **typage implicite** et le **typage explicite**. Vous n'êtes donc pas obligés de "typer" vos variables. Toutefois, c'est fortement recommandé.
- ▶ Le typage en TS ne sert qu'au compilateur pour prévenir les erreurs.
- ▶ Types proposés par Typescript :
 - ◆ Boolean
 - ◆ Number
 - ◆ String
 - ◆ Array
 - ◆ Enum
 - ◆ Any



TYPAGE

- ▶ Une variable déclaré selon un type précis ne peut plus recevoir des valeurs d'un autre type.

```
// legal
let isReady : boolean = false;
let decimal : number = 20;
let name    : string = "Dev.to";
let numbers : number[] = [1,2, 3];
```

```
// illegal
let isReady : boolean = 10;
let decimal : number = "not a number";
let name    : string = true;
let numbers : number[] = "not an array of numbers";
```

- ▶ En cas de besoin, on peut utiliser le pipe (|) pour déclarer les différents types de valeurs qui peuvent être stockés dans une variable.

```
let multiple : boolean | number = 10;
multiple = true; // still valid
```

- ▶ Ou encore le type **any** :

```
let unknown : any = 30;
unknown = "what is this variable?";
```

TYPE ASSERTIONS

- ▶ Typescript peut avoir des confusions concernant les types de variables.
- ▶ Ceci peut poser des problèmes pour le développeur.

```
let msg = 'abc';

let test = msg..
    endsWith  (method) String.endsWith
    fixed
    fontcolor
    fontsize
    includes
    index0f
    italics
    lastIndex0f
    length
    link
    localeCompare
    match
```

```
let msg;
msg = 'abc';

let test = msg..
```

IntelliSense ne propose rien car il ne reconnaît pas msg comme un String

CONVERSION DE TYPES

- ▶ Pour contourner ces problèmes, Typescript donne la possibilité de traiter une variable selon un type bien précis. Ce qu'on appelle plus communément le "cast".
- ▶ Ainsi, si une variable est déclaré avec le type any, et qu'à un certain moment de votre programme, vous avez besoin d'utiliser certaines méthodes dédiées au chaines de caractères (String), vous pouvez utiliser <string> comme dans l'exemple ci-dessous :

```
let unknown : any;
let unknown = "hello";

console.log("length is : ", (<string>unknown).length);
```

TYPAGE AVEC LES FONCTIONS

- ▶ On peut également typer les paramètres et la valeur de retour d'une fonction, toujours avec la syntaxe des deux points:

```
function creerCours(nb: number, nom: string): Cours {  
    var cours = new Cours();  
    cours.nb = nb;  
    cours.nom = nom;  
    return Cours;  
}
```

- ▶ Là aussi, le tapage des valeurs de retour des fonctions peut être d'une grande utilité et est donc fortement recommandé.

```
// illegal  
function test(name: string, surname: string): string {  
    return name.length; // will return a number here which is not expected  
}
```

TUPLES ET ENUMS

- ▶ TypeScript introduit les Tuples (comme dans Python) qui permettent de créer des tableaux où les types des éléments (fixe) sont connus. Ces types peuvent être différents.

```
let x: [boolean, number];
x = [true, 10]; // initialization
x = [10, "hello"]; // illegal initialization
```

- ▶ Autre différence avec Javascript, c'est l'arrivée des Enums, très connu en Java. Un Enum est un ensemble de constantes. Il existe trois types d'Enums :
- ▶ Enum Numériques
- ▶ Enum Strings
- ▶ Enum hétérogène

```
enum Direction {
    Up = 1,
    Down,
    Left,
    Right,
}
movePlayer(Direction.Up);
```

ARROW FUNCTIONS

- ▶ Les fonctions fléchées (ES6, TypeScript) sont une syntaxe simplifiée pour les expressions de fonction. Elles éliminent le mot-clé **function** et ajoutent une flèche **=>** entre les arguments et le corps de la fonction.

```
let log = function(message) {  
    console.log(message);  
}
```



```
let log = (message) => {  
    console.log(message);  
}
```



```
let log = (message) => console.log(message);
```

```
let resultat = function(a, b) {  
    return a + b;  
}
```



```
let resultat = (a, b) => {  
    return a + b;  
}
```



```
let resultat = (a, b) => a + b;
```

CLASSES

- ▶ JavaScript s'appuie sur les fonctions pour la création des composants
- ▶ JavaScript s'appuie sur les prototypes pour spécialiser ces composants
- ▶ JavaScript ne propose pas de notion de classe jusqu'à ES 2015.
- ▶ TypeScript propose d'utiliser les notions de programmation orientée objet tel que :
 - ◆ Le constructeur
 - ◆ L'encapsulation
 - ◆ Les accesseurs (ES5)
 - ◆ L'héritage
 - ◆ Les classes et interfaces

```
class Personne {  
    name : string;  
  
    constructor(name : string) {  
        this.name = name;  
    }  
  
    sayHello() {  
        console.log("Hello, i am " + this.name);  
    }  
}
```

```
let p : Personne;  
p = new Personne("Nidhal");  
p.sayHello();
```

CONSTRUCTOR

- ▶ Comme en Java, le constructeur est responsable de la création des objets à partir d'une classe.
- ▶ Contrairement au Java, une classe ne peut implémenter qu'un seul constructeur. D'où la possibilité de déclarer des arguments optionnels avec "?".
- ▶ Regardez bien la différence entre les 2 bouts de code. Alors ?

```
class Point {  
    private x : number;  
    private y : number;  
    constructor(x:number, y?:number) {  
        this.x = x;  
        this.y = y;  
    }  
  
    showPoint() {  
        console.log(`X: ${this.x} Y: ${this.y}`);  
    }  
}
```



```
class Point {  
    constructor(private x:number, private y:number) {}  
  
    showPoint() {  
        console.log(`X: ${this.x} Y: ${this.y}`);  
    }  
}
```

ACCESSEURS

- ▶ Comme en Java ou en C#, le principe d'encapsulation adopté par Typescript nous propose 3 modes d'accès:
 - ◆ Public
 - ◆ Private
 - ◆ Protected
- ▶ Contrairement à Java, le mode par défaut est Public.

PROPRIÉTÉS

- ▶ Pour accéder en lecture ou en écriture à un attribut de classe, nous avons l'habitude d'utiliser des accesseurs (getters + setters).

```
class Point {  
    constructor(private x:number, private y:number) {  
    }  
  
    showPoint() {  
        console.log(`X: ${this.x} Y: ${this.y}`);  
    }  
  
    getX() {  
        return this.x;  
    }  
  
    getY() {  
        return this.y  
    }  
}
```

```
class Point {  
    constructor(private x:number, private y:number) {  
    }  
  
    showPoint() {  
        console.log(`X: ${this.x} Y: ${this.y}`);  
    }  
  
    get X() {  
        return this.x;  
    }  
  
    set X(newValue){  
        this.x = newValue;  
    }  
}
```

Testons ces bouts de code...

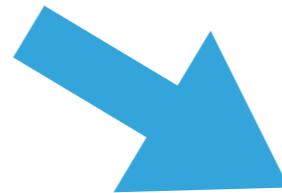
INTERFACES

- ▶ Les Interfaces nous permettent d'établir des contraintes par rapport aux objets et aux classes qu'on implémente.
- ▶ Peuvent être utilisés pour définir des types bien définies sans passer par les classes.
- ▶ **Il est important de noter que les interfaces ne sont pas compilés en Javascript.**
- ▶ Ils servent juste à la validation de nos variables ou autre objet par le compilateur de Typescript.

GENERICCS

- ▶ Generics en Typescript donne la possibilité de passer plusieurs types à un composant, en ajoutant une couche supplémentaire d'abstraction et de réutilisation à notre code.
- ▶ Les génériques peuvent être appliqués aux fonctions, interfaces et classes dans Typescript.

```
function identity<T>(arg: number): number {  
    return arg;  
}
```



```
function identity<T>(arg: T): T {  
    return arg;  
}
```

MODULES

- ▶ Les modules servent à partager du code entre divers fichiers sources.
Utilisation des mots-clés **Export** et **Import**.
- ▶ 2 types d'export : Export nommée et Export par défaut.
- ▶ On ne peut avoir qu'une seul Export par défaut par fichier.
- ▶ Possibilité d'utiliser des alias.

```
export default class Etudiant {  
    id : number;  
    nom : string;  
    prenom : string;
```

```
export function testFct() {  
    console.log("This is a text");  
}
```

```
import Etudiant, { testFct } from './component';
```

```
import Etudiant, { testFct as tf} from './component';
```

DÉCORATEURS

- ▶ Les décorateurs (appelés également annotations) TypeScript permettent d'ajouter des informations sur nos classes, pour indiquer par exemple que telle classe est un composant de l'application, ou telle autre un service. On utilise @ comme syntaxe :

```
@Pipe({  
  name: 'filter',  
  pure : false  
})  
export class FilterPipe {
```

```
@Directive({  
  selector: '[appCustomDir]'  
})  
export class CustomDirDirective {
```

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

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

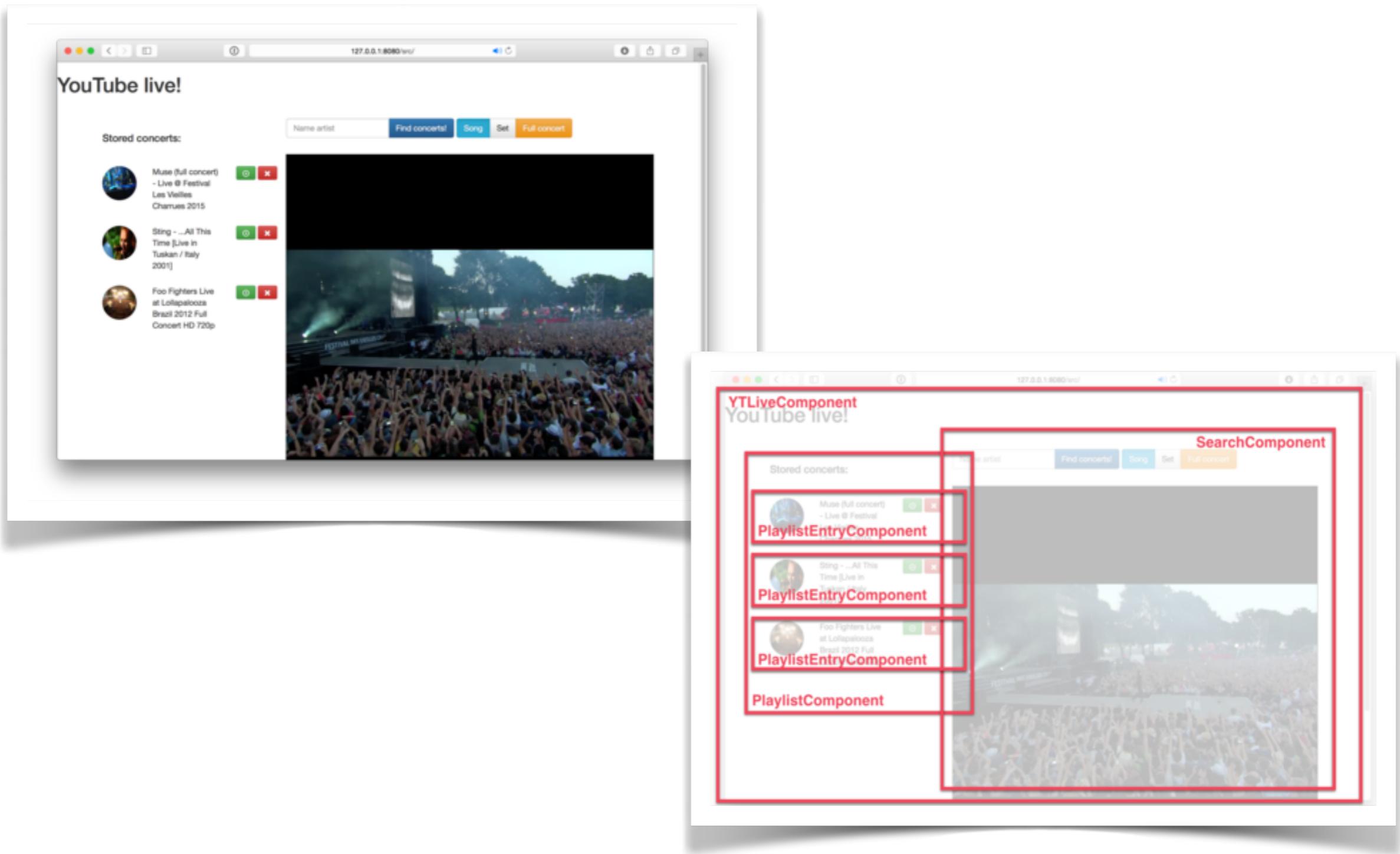
Chapitre 3 : Les composants



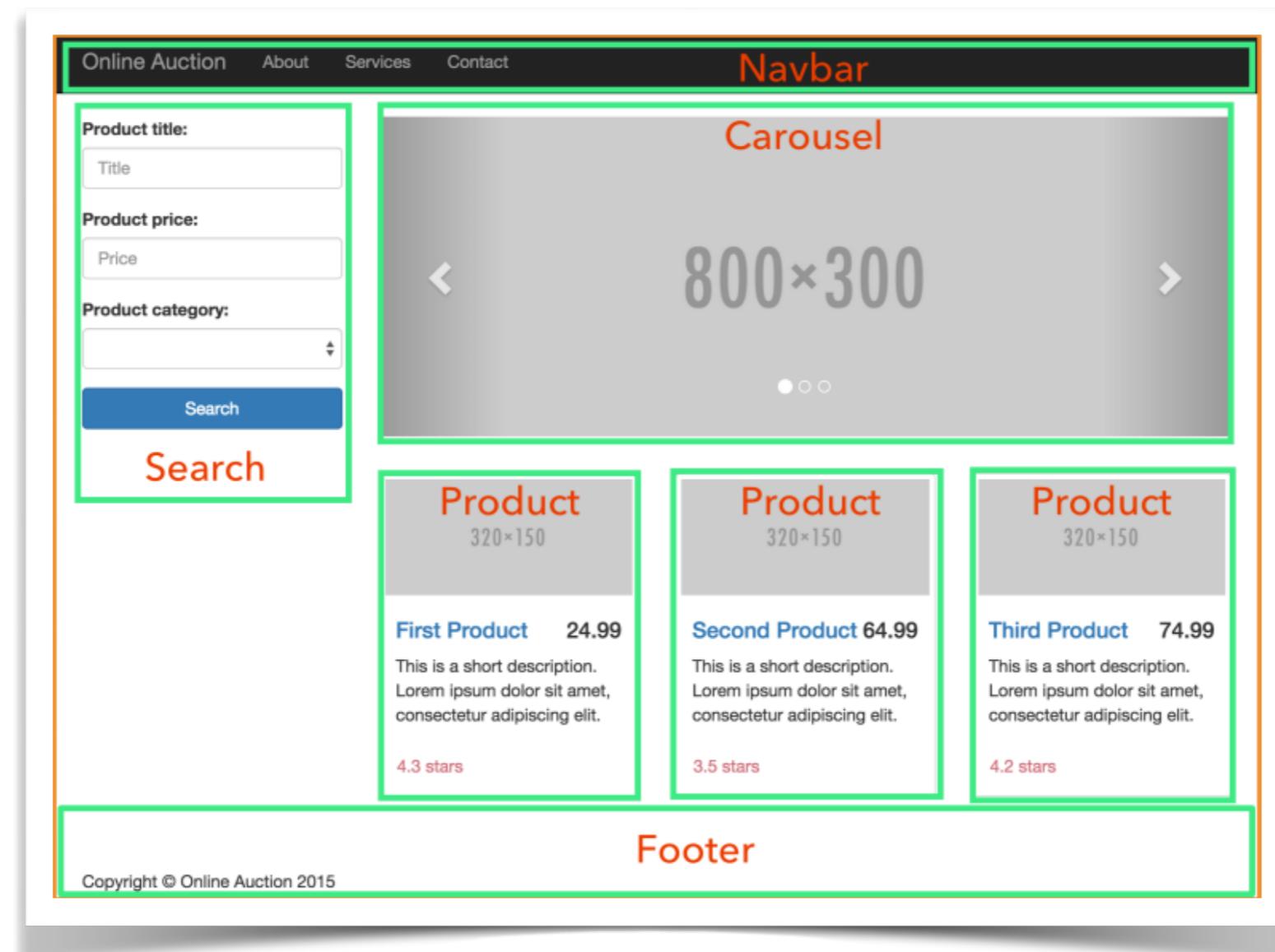
OBJECTIFS

- ▶ Comprendre la définition du composant
- ▶ Assimiler et pratiquer la notion de Binding
- ▶ Gérer les interactions entre composants.

EXEMPLE 1



EXEMPLE 2



QU'EST CE QU'UN COMPOSANT (COMPONENT) ?

- ▶ Un composant est une **classe** qui permet de gérer une **vue**. Il se charge uniquement de cette vue-là.
- ▶ Plus simplement, un composant est un fragment HTML géré par une classe JS (component en angular et controller en angularJS)
- ▶ Une application Angular est un arbre de composants
- ▶ La racine de cet arbre est l'application lancée par le navigateur au lancement.

QU'EST CE QU'UN COMPOSANT (COMPONENT) ?

- ▶ Le composant est donc la partie principale d'Angular. Il s'occupe d'une partie de la vue.
- ▶ L'interaction entre le composant et la vue se fait à travers différents types de binding.

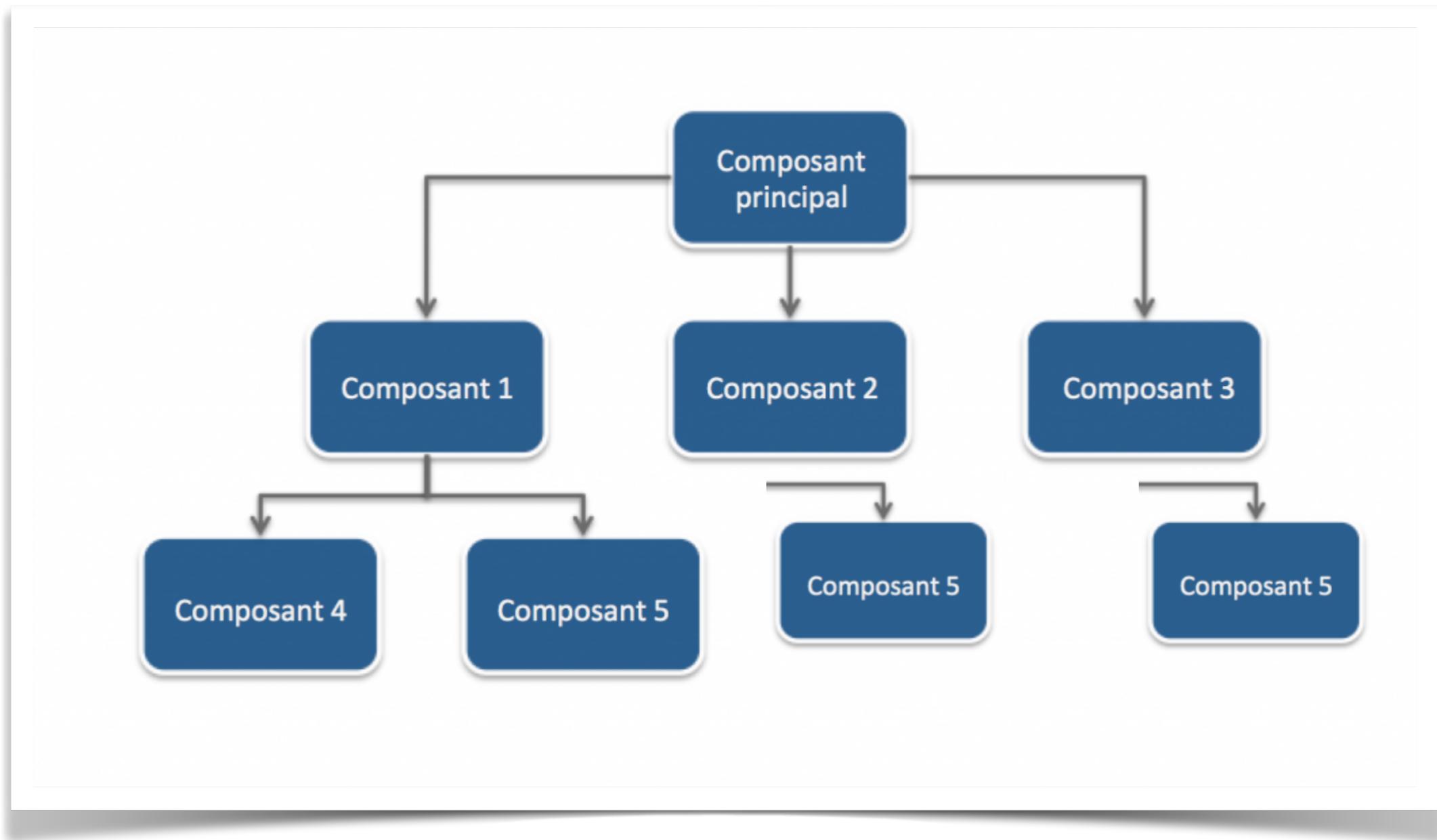


- ▶ Un Template est le complément du composant.
- ▶ C'est la vue associée au composant.
- ▶ Elle représente le code HTML géré par le composant.

QU'EST CE QU'UN COMPOSANT (COMPONENT) ?

- ▶ Ainsi, un composant est :
 - **Composable** (évidemment puisque c'est un composant 😊)
 - **Réutilisable**
 - **Hiérarchique** (n'oubliez pas c'est un arbre)
- ▶ Dans le reste du cours les mots composant et component représentent toujours un composant Angular.

ARBRE DE COMPOSANTS



VOTRE PREMIER COMPOSANT

- ▶ Chargement de la classe **Component**
- ▶ Le décorateur **@Component** permet d'ajouter un comportement à notre classe et de spécifier que c'est un Composant Angular.
- ▶ c'est un Composant Angular.
 - ▶ **selector** permet de spécifier le tag (nom de la balise) associé ce composant
 - ▶ **templateUrl**: spécifie l'url du template associé au composant
 - ▶ **styleUrls**: tableau des feuilles de styles associé à ce composant
- ▶ **Export** de la classe afin de pouvoir l'utiliser

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

@Component({
  selector: 'liste-cours',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular App for ISIE 1st year';
```

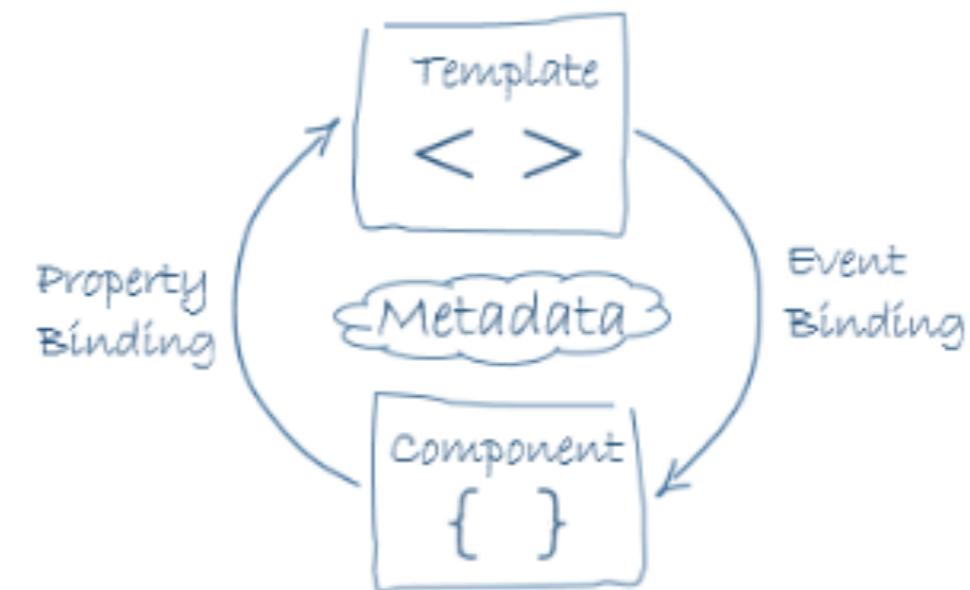
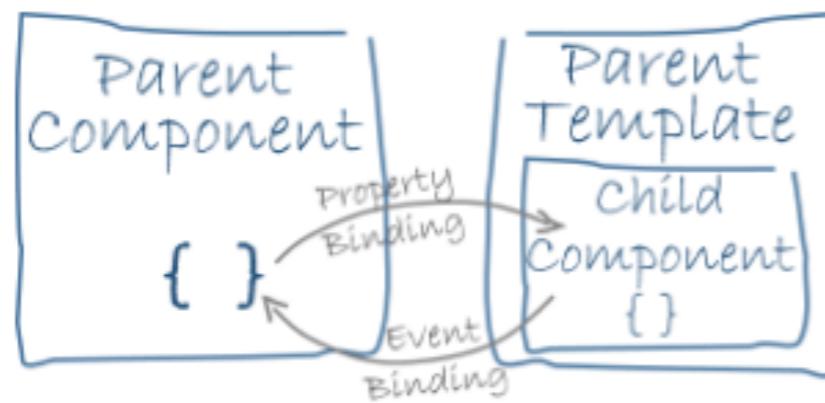
CRÉATION D'UN COMPOSANT

Pour créer un composant, il existe 2 méthodes possibles :

- ▶ **Manuelle**
 - ▶ Créer la classe
 - ▶ Importer Component
 - ▶ Ajouter l'annotation et l'objet qui la décore
 - ▶ Ajouter le composant dans le AppModule (`app.module.ts`) dans l'attribut declarations ou dans *imports* d'un standalone component.
- ▶ **Cli**
 - ▶ Avec la commande **ng generate component my-new-component** ou son raccourci **ng g c my-new-component**

BINDING

- Le Binding est le mécanisme qui permet de mapper des éléments du DOM avec des propriétés et des méthodes du composant. Le Binding permettra également de faire communiquer les composants.



- Les Metadata sont des informations permettant de décrire les classes. Ainsi, @Component permet par exemple d'identifier la classe comme étant un composant angular.

INTERPOLATION

- ▶ L'interpolation permet de projeter des valeurs de propriétés ou de variables dans votre template.
- ▶ Elle permet également d'évaluer toute expression dynamique.
- ▶ Angular utilise la syntaxe "double accolades " pour l'interpolation :

```
export class CartComponent {  
  name : string = "Nidhal";
```

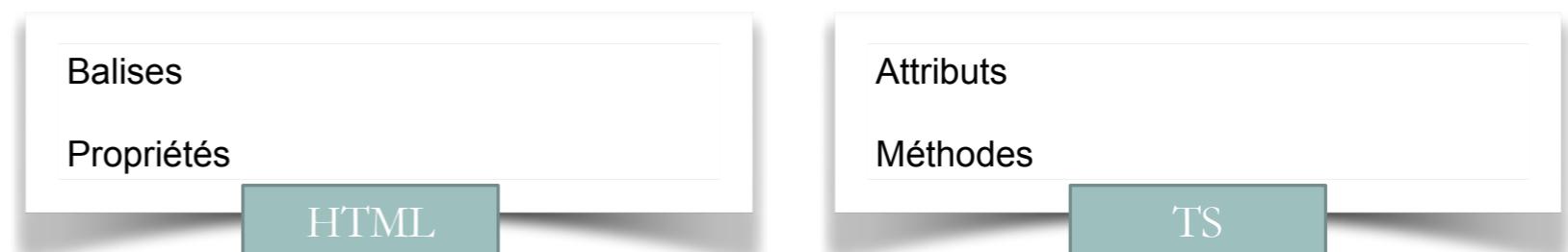
CartComponent.html



```
<p> Cours d'Angular avec {{ name }} </p>
```

PROPERTY BINDING

- ▶ Binding **unidirectionnel**.
- ▶ Permet aussi de récupérer dans le DOM des propriétés du composant.
- ▶ La propriété liée au composant est interprétée avant d'être ajoutée au Template.
- ▶ Deux possibilités pour la syntaxe:
 - ▶ [propriété]
 - ▶ **bind**-propriété



PROPERTY BINDING (EXEMPLE)

```
<p>Je suis {{ name }}</p>  
  
<div [hidden]="hd" [style.backgroundColor]="myColor">  
| Je suis une DIV  
</div>  
  
<button (click)="traitement()" class="btn btn-danger">  
| Cliquez-ici  
</button>
```

HTML

```
@Component({  
  selector: 'app-color',  
  templateUrl: './color.component.html',  
  styleUrls: ['./color.component.css']  
})  
export class ColorComponent implements OnInit {  
  name : string = "Nidhal"  
  
  myColor = "red";  
  
  hd = false;  
  
  constructor() { }  
  
  ngOnInit() {  
  }  
  
  traitement() {  
    alert('Vous avez cliqué !')  
  }  
}
```

TS

EVENT BINDING

- ▶ Binding unidirectionnel.
- ▶ Permet d'interagir du DOM vers le composant.
- ▶ L'interaction se fait à travers les événements.
- ▶ Deux possibilités pour la syntaxe :
 - ▶ (evenement)
 - ▶ **on**-evenement

EVENT BINDING (EXEMPLE)

```
<p>Je suis {{ name }}</p>

<div [hidden]="hd" [style.backgroundColor]="myColor">
  Je suis une DIV
</div>

<button (click)="traitement()" class="btn btn-danger">
  Cliquez-ici
</button>
```

HTML

```
@Component({
  selector: 'app-color',
  templateUrl: './color.component.html',
  styleUrls: ['./color.component.css']
})
export class ColorComponent implements OnInit {
  name : string = "Nidhal"

  myColor = "red";

  hd = false;

  constructor() { }

  ngOnInit() {
  }

  traitement() {
    alert('Vous avez cliqué !')
  }
}
```

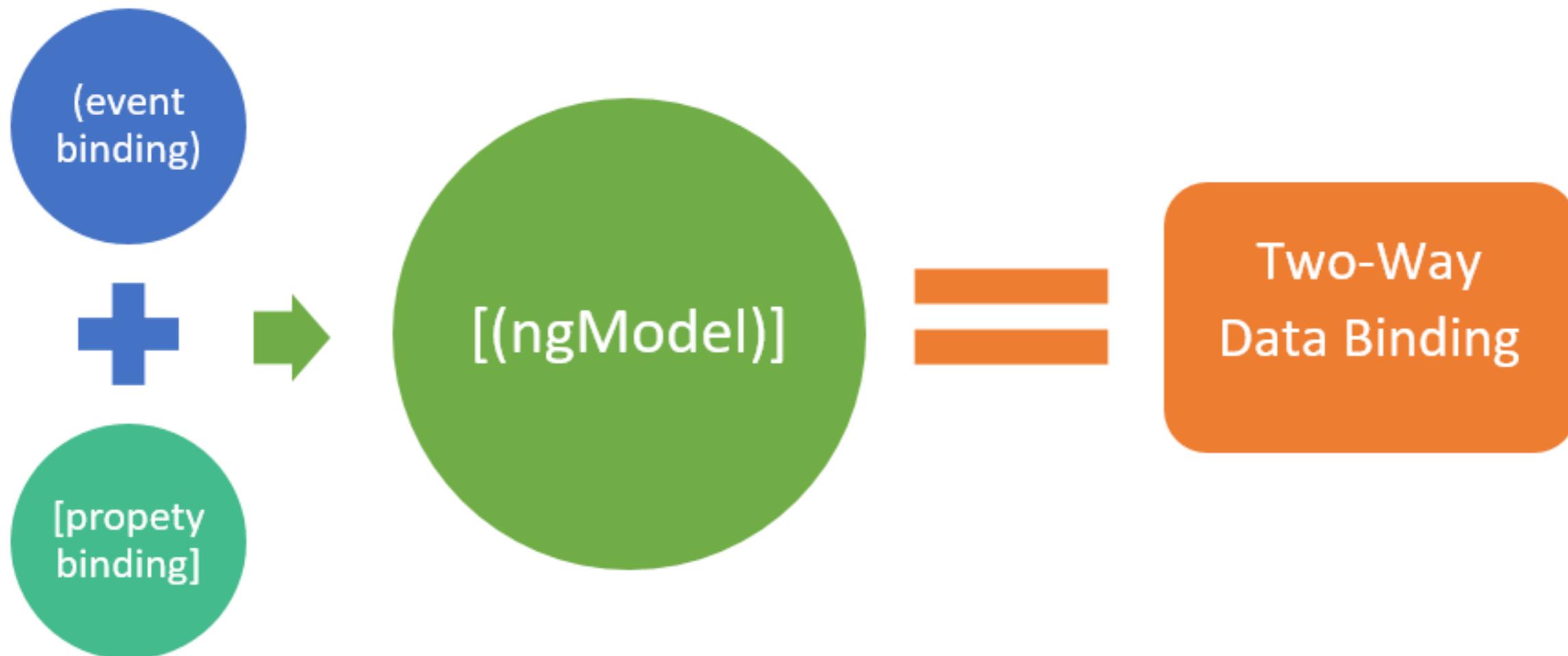
TS

TWO-WAY BINDING

- ▶ Binding **Bi-directionnel**
- ▶ Permet d'interagir du DOM vers le composant et du composant vers le DOM.
- ▶ Se fait à travers la directive **ngModel** (*on reviendra sur le concept de directive plus en détail*)
- ▶ Syntaxe : **[(ngModel)]**=property

N.B : Afin de pouvoir utiliser ngModel vous devez importer le **FormsModule** du **@angular/forms** dans **app.module.ts**

PROPERTY BINDING + EVENT BINDING



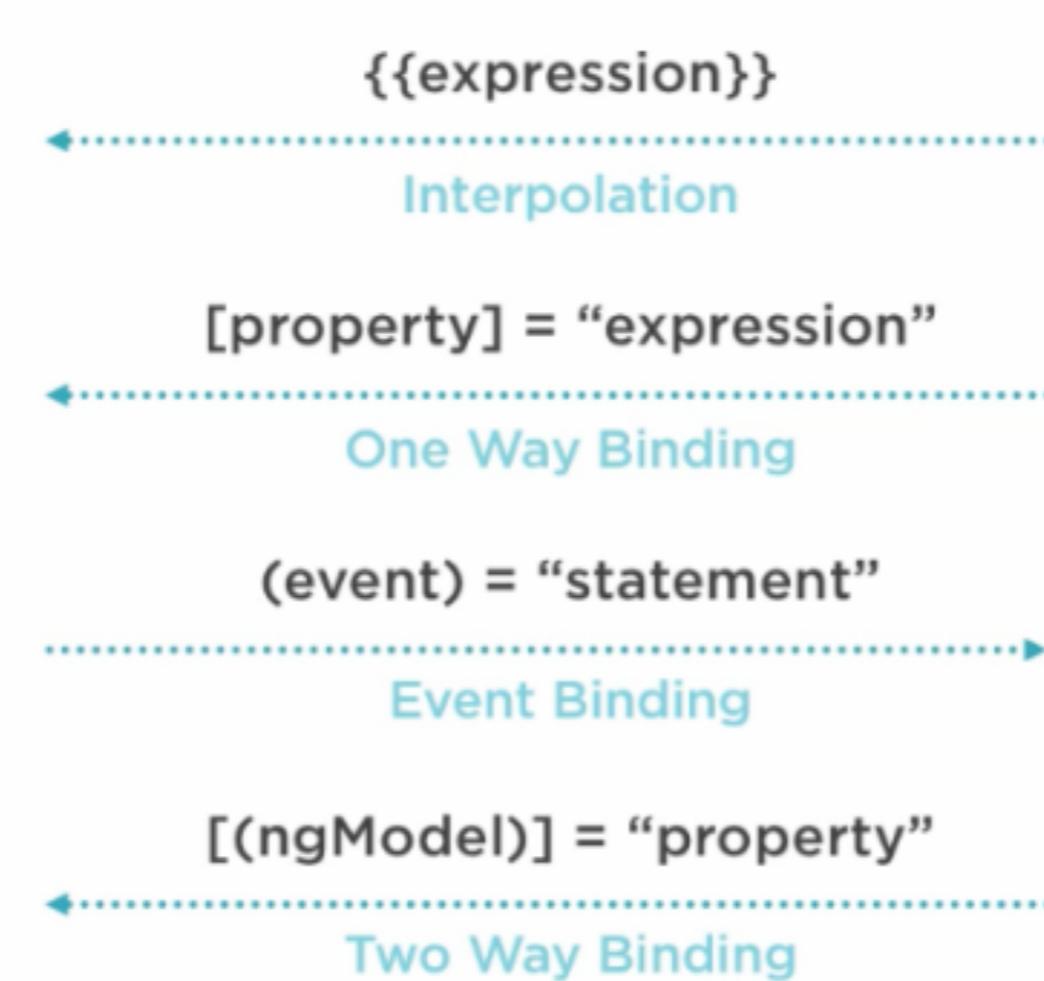
```
<hr>
Change me <input [(ngModel)]="nom">
<br>My new name is {{nom}}
```

Template

RÉCAPITULONS... AVANT D'AVANCER



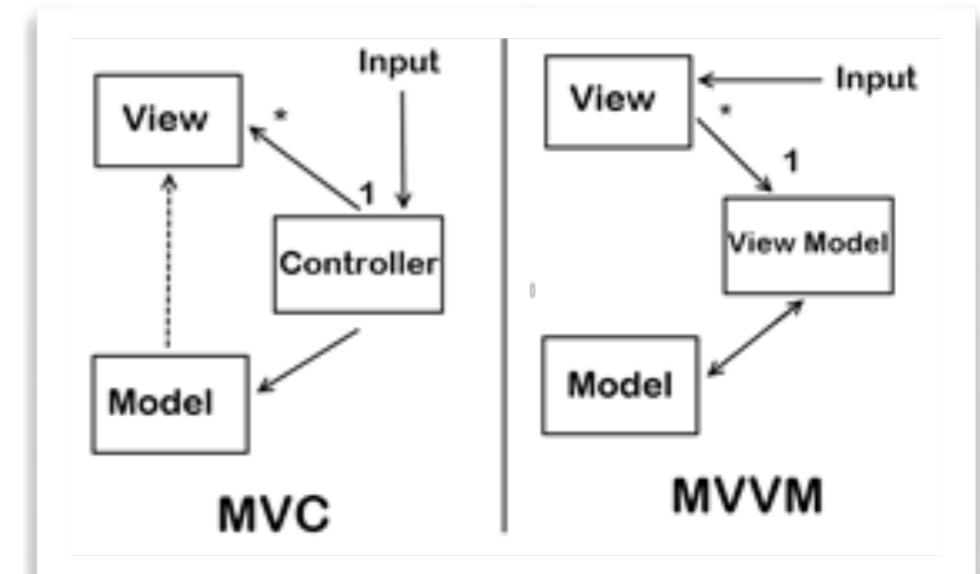
DOM



Component

MVVM DESIGN PATTERN

- ▶ Avec le two-ways-binding, Angular adopte une architecture MVVM.
- ▶ Le contrôleur (le C dans MVC) est remplacé par le ViewModel (la VM dans MVVM). Le ViewModel, qui est comme un contrôleur, est responsable du maintien de la relation entre la vue et le modèle.
- ▶ Cependant, la différence ici est que, si nous mettons à jour quoi que ce soit en vue, il est mis à jour dans le modèle. De même, changez quoi que ce soit dans le modèle, cela apparaît dans la vue.
- ▶ En MVC, les informations de la vue sont envoyées au serveur pour mettre à jour le modèle. Or, dans Angular, nous fonctionnons côté client, nous pouvons donc mettre à jour les objets associés en temps réel.



TEMPLATE REFERENCE

- ▶ Une référence en Angular est une variable qui permet de référencer un élément du DOM dans le template du composant. L'objectif est de permettre d'accéder facilement à cet élément n'importe où dans le template.
- ▶ On utilise le symbole **#** pour la création d'une variable de référence.

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

@Component({
  selector: 'app-card',
  templateUrl: './card.component.html',
  styleUrls: ['./card.component.css']
})
export class CardComponent {
  name : string;

  getName() : string {
    return this.name;
  }

  setName(newName) {
    this.name = newName;
  }
}
```

Component

```
Name :
<input #myname [value]="getName()" class="form-control">
<br>
<button (click) = "setName(myname.value)" class="btn btn-primary">
  Change name
</button>
```

Template

APPLICATION

Nous allons créer un aperçu de carte visite. Pour cela, vous devez :

- ▶ Créer un composant.
- ▶ Préparer une interface permettant de saisir d'un coté les données à insérer dans une carte visite. De l'autre coté et instantanément les données de la carte seront mis à jour.
- ▶ Préparer l'affichage de la carte visite. Vous pouvez utiliser ce thème gratuit : <https://www.creative-tim.com/product/rotating-css-card>

CYCLE DE VIE D'UN COMPOSANT

- ▶ Chaque composant a un cycle de vie géré par Angular lui-même.
 1. Angular crée le composant
 2. S'occupe de l'afficher
 3. Crée et affiche ses composants fils
 4. Vérifie quand les données des propriétés changent
 5. Détruit les composants, avant de les retirer du DOM quand cela est nécessaire.

CYCLE DE VIE D'UN COMPOSANT

- ▶ Angular offre aux développeurs la possibilité d'agir sur ces moments clés (les phases du cycle de vie) quand ils se produisent, en implémentant (par défaut) une ou plusieurs interfaces, pour chacun des événements disponibles.
- ▶ Ces interfaces sont disponibles dans la librairie **core** d'Angular.
- ▶ Chaque interface permettant d'interagir sur le cycle de vie d'un composant fournit **une et une seule** méthode, dont le nom est le nom de l'interface préfixé par **ng**. Par exemple, l'interface **OnInit** fournit la méthode **ngOnInit**, et permet de définir un comportement lorsque le composant est initialisé.

CYCLE DE VIE D'UN COMPOSANT

- ▶ Ci-dessous la liste des *principales* méthodes disponibles pour interagir avec le cycle de vie d'un composant, **dans l'ordre chronologique** du moment où elles sont appelées par Angular.
1. **ngOnChanges** : C'est la méthode appelée en premier lors de la création d'un composant, avant même *ngOnInit*, et à chaque fois que Angular détecte que les valeurs d'une propriété du composant sont modifiées. La méthode reçoit en paramètre un objet représentant les valeurs actuelles et les valeurs précédentes disponibles pour ce composant.
 2. **ngOnInit** : Cette méthode est appelée juste après le premier appel à *ngOnChanges*, et **elle initialise le composant après qu'Angular ait initialisé les propriétés du composant**.

CYCLE DE VIE D'UN COMPOSANT

3. **ngDoCheck:** On peut implémenter cette interface pour étendre le comportement par défaut de la méthode `ngOnChanges`, afin de pouvoir détecter et agir sur des changements qu'Angular ne peut pas détecter par lui-même.
4. **ngAfterViewInit:** Cette méthode est appelée juste après la mise en place de la vue d'un composant, et des vues de ses composants fils s'il en a. A partir de ce moment, les propriétés initialisées par `@ViewChild` et `@ViewChildren` sont valorisées. Il n'est appelé qu'une seule fois après `ngAfterContentChecked`.

CYCLE DE VIE D'UN COMPOSANT

5. **ngOnDestroy**: Appelée en dernier, cette méthode est appelée avant qu'Angular ne détruise et ne retire du DOM le composant. **Ex :** *Cela peut se produire lorsqu'un utilisateur navigue d'un composant à un autre par exemple. Afin d'éviter les fuites de mémoire, c'est dans cette méthode que nous effectuerons un certain nombre d'opérations afin de laisser l'application "propre" (nous détacherons les gestionnaires d'événements par exemple).*



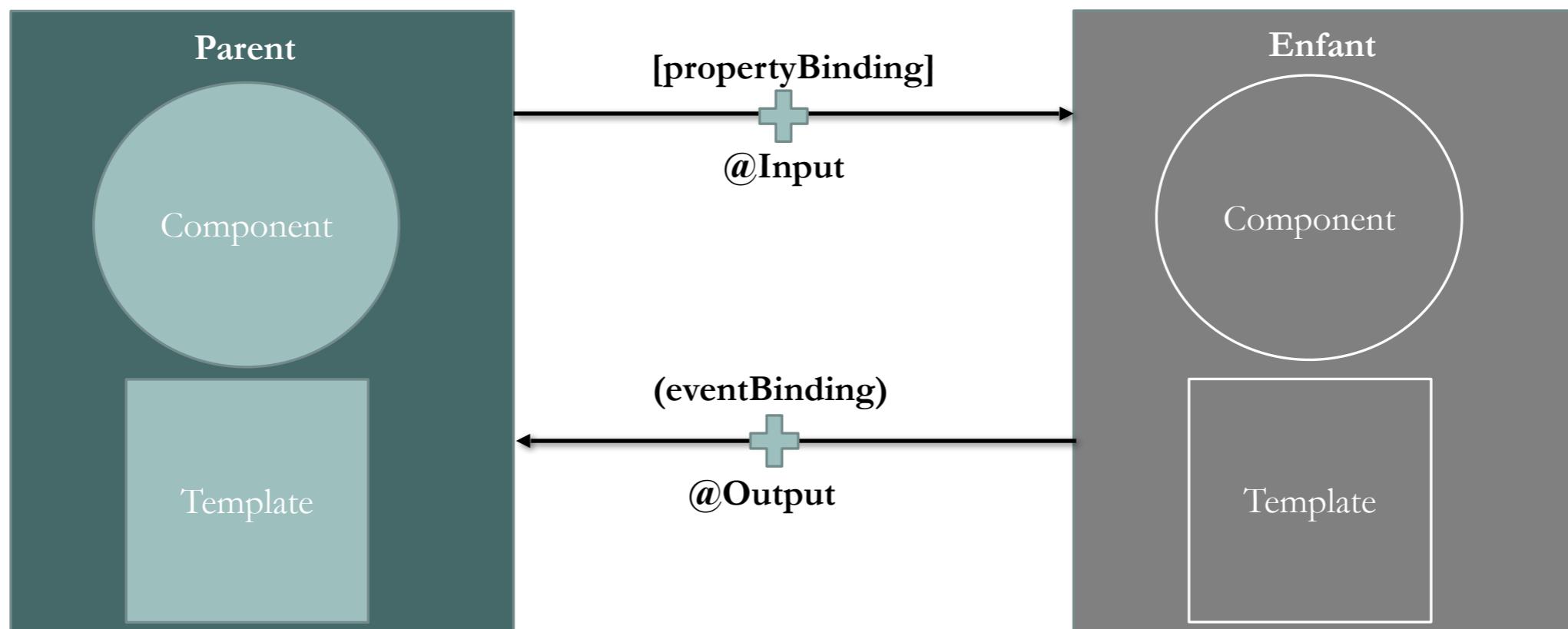
Les méthodes que vous utiliserez le plus seront certainement `ngOnInit` et `ngOnDestroy`, qui permettent d'initialiser un composant, et de le nettoyer proprement par la suite lorsqu'il est détruit.

CYCLE DE VIE D'UN COMPOSANT



- ▶ Il est important à noter que même si vous ne définissez pas explicitement des méthodes de cycle de vie dans votre composant, ces méthodes sont appelées en interne par Angular pour chaque composant.
- ▶ Lorsqu'on utilise ces méthodes, on vient donc juste surcharger tel ou tel événement du cycle de vie d'un composant.

INTERACTION ENTRE LES COMPOSANTS



INTERACTION ENTRE LES COMPOSANTS

- ▶ Relation entre un composant **parent** et un composant **enfant** :
- ▶ Le parent voit l'enfant mais ne peut pas voir ses propriétés.
- ▶ **Solution** : Faire du property binding avec @input, qui peut prendre un objet en paramètre (pour spécifier que l'envoie d'une valeur est required par exemple).

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

@Component({
  selector: 'app-child-first',
  templateUrl: './child-first.component.html',
  styleUrls: ['./child-first.component.css']
})
export class ChildFirstComponent implements OnInit {
  @Input() colour:string;
```

INTERACTION ENTRE LES COMPOSANTS

- ▶ Relation entre un composant **enfant** et un composant **parent** :
- ▶ L'enfant ne voit pas le parent car il ne sait simplement pas par quel composant il a été appelé.
- ▶ **Solution** : 1. Faire du event binding avec @output

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

@Component({
  selector: 'app-child-first',
  templateUrl: './child-first.component.html',
  styleUrls: ['./child-first.component.css']
})
export class ChildFirstComponent implements OnInit {
  @Output() sendRequest = new EventEmitter();
```

INTERACTION ENTRE LES COMPOSANTS

2. Configurer l'événement

```
sendEvent() {
    this.sendRequest.emit('Accuse la réception de la couleur '+ this.color);
}
```

3. Récupérer l'évènement dans le composant parent

```
<app-child-first (sendRequest)="ReceivedEvent($event)"></app-child-first>
```

4. Traiter le message reçu de la part du composant enfant

```
ReceivedEvent(msg : any)
{
    alert(msg);
}
```

ngOnChanges

- ▶ Comme déjà expliqué, cette méthode est appelée à chaque fois que Angular détecte un changement des informations traitées par le composant.

```
export class ReceiveComponent implements OnInit, OnChanges {
  @Input() msgSend : String;
```

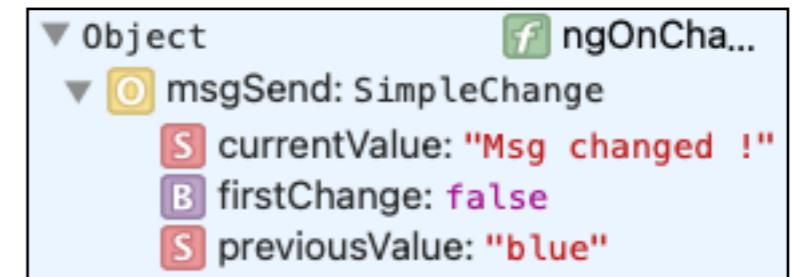
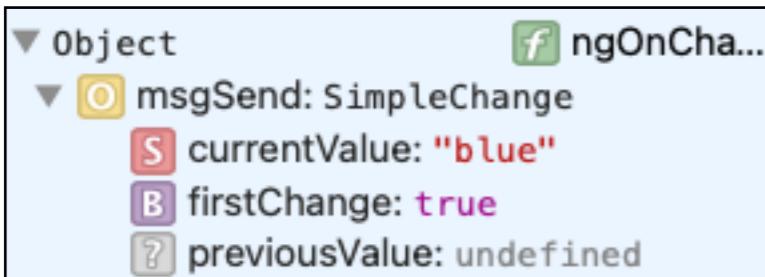
```
  ngOnChanges(changes: SimpleChanges): void {
    console.log("ngOnChanges");
    console.log(changes);
  }
}
```

receive.component.ts

```
onChangeMsg() {
  this.color = "Msg changed !"
}
```

app.component.ts

```
<app-receive [msgSend]="color"></app-receive>
app.component.html
```



ngOnChanges... alternative

- ▶ ngOnChanges est appelé quand n'importe quel input voit sa valeur modifiée, ce qui n'est pas recommandé.
- ▶ Comme alternative, Angular nous propose d'utiliser la méthode set.

```
export class ChildComponent {  
  @Input({ required: true }) nameToShow;  
  ...  
  
  ngOnChanges(change: SimpleChange) {  
    console.log(change['nameToShow'].currentValue[0]);  
    this.eMailGenerated = `${change['nameToShow'].currentValue[0]}${  
      change['nameToShow'].currentValue[  
        change['nameToShow'].currentValue.length - 1  
      ]  
    }@angular.com`;  
  }  
}
```



```
export class ChildComponent {  
  @Input({ required: true }) set nameToShow(val) {  
    this.eMailGenerated = `${val[0]}${val[val.length - 1]}@angular.com`;  
  }  
  ...  
  eMailGenerated;
```

```
<h4>Hello {{ nameToShow }},  
| you email is {{ eMailGenerated }}</h4>
```

Change Candidate Homer
Hello , you email is Hr@angular.com

ngOnInit VS Constructor

- ▶ Il est assez courant de confondre les deux. Il est donc fortement recommandé de comprendre à quoi sert chacun deux.
- ▶ *ngOnInit* est appelée quand le composant est initié. Plus précisément, quand les propriétés sont affichées et que les propriétés **@Input** sont initiés.
- ▶ La méthode **constructor** n'a absolument rien à voir avec Angular. Il s'agit d'une méthode liée aux classes TypeScript.
- ▶ Le framework Angular n'a absolument aucun contrôle sur ce constructor et son appel.

ngOnInit VS CONSTRUCTOR

Pour faire court donc :

- ▶ Le seul intérêt d'utiliser le **constructor** dans vos classes est lors de l'injection de dépendances avec les services.
- ▶ En effet, rappelons que le constructor est appelé par le moteur JavaScript. Par conséquent, il pourra dire au framework Angular de quelles dépendances, il va avoir besoin.
- ▶ Pour l'initialisation des attributs de vos composants, passez plutôt par ngOnInit.

<NG-CONTENT>

- ▶ La balise <ng-content> permet de définir un modèle de vue fixe qui est inséré quand on fait appel à un composant.
- ▶ Dans l'exemple ci-dessous, on ajoutant le composant ReceiveComponent, ce qu'on inséré entre <app-receive> et </app-receive> peut être récupéré dans la vue receive.component.html via la balise <ng-content>.

```
<app-receive> <p #paragraphe>Un paragraphe à passer à appReceive</p></app-receive>  
app.component.html
```

```
<h2><ng-content></ng-content></h2>
```

receive.component.html

- ▶ Notez qu'on référencé la balise p dans le contenu à passer à ReceiveComponent. Comment peut-on le récupérer ?

@CONTENTCHILD

- ▶ Pour récupérer un élément référencé dans la vue et envoyé à travers <ng-content>, Angular nous propose @ContentChild.

```
@Component({
  selector: 'app-receive',
  templateUrl: './receive.component.html',
  styleUrls: ['./receive.component.css']
})
export class ReceiveComponent implements OnInit, OnChanges, AfterContentInit {
  @ContentChild('paragraphe') pg;
```

- ▶ Il est important de noter que la récupération du contenu d'un @ContentChild ne se fait que lors de l'exécution de la méthode ngAfterContentInit().

```
ngAfterContentInit(): void {
  console.log(this.pg)
}
```

@VIEWCHILD

- ▶ Semblable à @ContentChild, @ViewChild permet de récupérer et contrôler dans notre classe Typescript un élément HTML qui a été référencé par #.

```
<input #in type="text" class="form-control">
```

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

  @ViewChild('in', {static : true}) inputValue;
```

- ▶ La priorité static de l'objet donné en argument à @ViewChild doit être égal à true si l'élément (dans l'exemple, inputValue) va être utilisé par ngOnInit(). Sinon, il prend la valeur false.

DÉBUTONS NOTRE PROJET

- ▶ L'objectif est de créer une mini plateforme de recrutement.
- ▶ La première étape est de créer la structure suivante avec une vue contenant deux parties :
 - ▶ Liste des Cvs inscrits
 - ▶ Détails du Cv qui apparaitra au clic sur un candidat
- ▶ Il vous est demandé juste d'afficher un seul Cv à la fois
- ▶ Il faudra suivre l'architecture du diapo précédent.

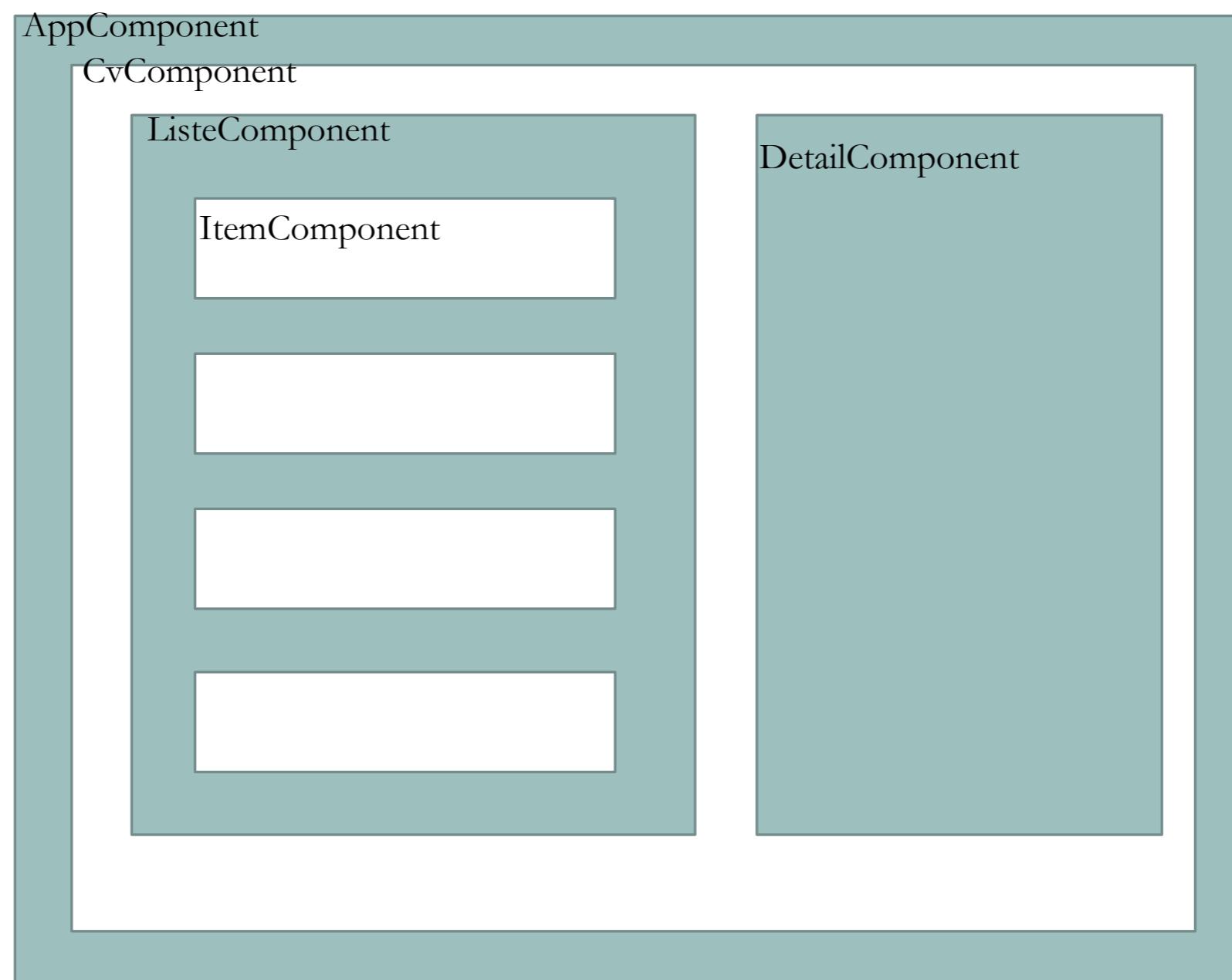
RÉSULTAT ATTENDU

- ▶ Un cv est caractérisé par : id, nom, prenom, age, profession et avatar.

The image shows a mobile application interface. On the left, there is a vertical list of three contacts: Nidhal JELASSI, Bart SIMPSON, and Homer SIMPSON. Each contact entry includes a small square thumbnail image on the left and the contact's name on the right. On the right, there is a larger, detailed view of the first contact, Nidhal Jelassi. This view features a large circular thumbnail of Nidhal Jelassi's face, a blue header bar, and the contact's name "Nidhal Jelassi" followed by "35 ans" and "enseignant". At the bottom of this view, there is a button labeled "Auto Rotation".

DÉBUTONS NOTRE PROJET

- ▶ Application de gestion de CVs.



Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

Chapitre 4 : Les directives
Supplément : Les Pipes



OBJECTIFS

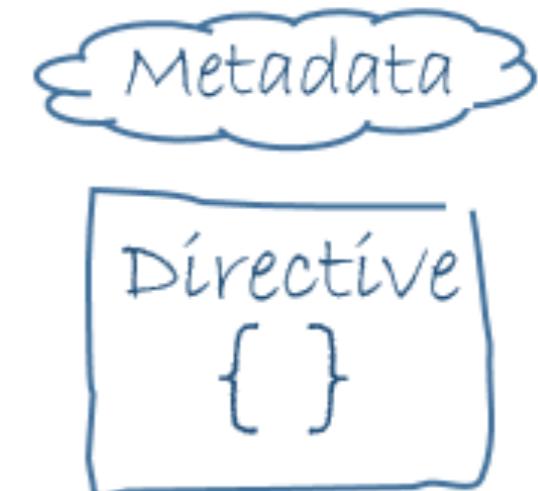
- ▶ Comprendre la définition et l'intérêt des directives.
- ▶ Voir quelques directives d'attributs offertes par Angular et savoir les utiliser
- ▶ Créer votre propre directive d'attributs
- ▶ Voir quelques directives structurelles offertes par Angular et savoir les utiliser
- ▶ Définir les pipes et l'intérêt de les utiliser
- ▶ Vue globale des pipes prédéfinies
- ▶ Créer un pipe personnalisé

QU'EST CE QU'UNE DIRECTIVE ?

- ▶ Jusqu'ici, nous avons déjà utilisé des directives... sans vraiment nous en rendre compte.
- ▶ En effet, une directive est une classe comme le composant sauf qu'elle n'a pas de template.
- ▶ C'est une classe qui permet d'attacher un comportement spécifique aux éléments du DOM. Elle est décorée avec l'annotation **@Directive**.
- ▶ Apparaît dans un élément comme un tag (comme le font les attributs).
- ▶ La commande pour créer une directive est : **ng g d nomDirective**

QU'EST CE QU'UNE DIRECTIVE ?

- ▶ Il est possible d'avoir plusieurs directives appliquées à un même élément.
- ▶ Il est possible d'avoir plusieurs directives appliquées à un même élément.
- ▶ Une directive possède un sélecteur CSS qui indique au framework où l'activer dans notre template.
- ▶ Lorsque Angular trouve une directive dans notre template HTML, il instancie la classe de la directive correspondante et donne à cette instance le contrôle sur cet élément du DOM.



TYPES DE DIRECTIVES

Angular distingue trois types de directives :

- ▶ Les composants qui sont des directives avec des templates. D'ailleurs la classe **Component** hérite de la classe **Directive** dans Angular.
- ▶ Les directives structurelles qui changent l'apparence du DOM en ajoutant, manipulant et supprimant des éléments. Les directives **@for** et **@if** font partie de cette catégorie.
- ▶ Les directives d'attributs (attribute directives) qui permettent de changer l'apparence ou le comportement d'un élément.

A blue circular icon containing the white number '1'.

Les directives structurelles

LES DIRECTIVES STRUCTURELLES

- ▶ Comme déjà dit, les directives structurelles d'Angular sont les directives qui manipulent les éléments du DOM, comme **@if**, **@for** et **@switch**.
- ▶ Avant la version 17, ces directives avaient une autre syntaxe avec ***ngIf**, ***ngFor** et **[ngSwitch]**.
- ▶ Elles sont appliquées sur **l'élément HOST** (l'élément sur lequel la directive est appliquée).
- ▶ Alors que l'*interpolation* et le *property binding* permettent de modifier l'affichage et le contenu, ils ne permettent pas de modifier la structure du DOM en ajoutant ou en retirant des éléments par exemple

LES DIRECTIVES STRUCTURELLES

- ▶ Pour remédier à cette limitation, Angular fournit des directives structurelles qui permettent de modifier la structure du DOM.
- ▶ Les directives de ce type fournies par Angular sont :
 - ▶ *@if (anciennement *ngIf)*
 - ▶ *@for (anciennement *ngFor)*
 - ▶ *@switch, @case et @default* (anciennement [ngSwitch],
**ngSwitchCase* and **ngSwitchDefault*)

@For

- ▶ La directive structurelle **@for** permet de boucler sur un array et d'injecter les éléments dans le DOM.

```
<ol>
  @for (candidat of listeCandidats; track $index) {
    <li>{{ candidat }}</li>
  }
</ol>
```



Bienvenue dans l'application de gestion de CV

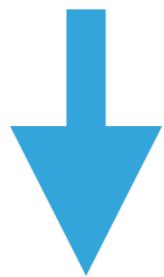
- 1. Bart
- 2. Homer
- 3. Marge

@For

- ▶ **@for** permet de répéter un élément plusieurs fois dans le DOM.
- ▶ Prend en paramètre les entités à reproduire.
- ▶ Fournit certaines valeurs :
 - ▶ `$index`: position de l'élément.
 - ▶ `$odd`: true si l'élément est à une position impaire.
 - ▶ `$even`: true si l'élément est à une position paire.
 - ▶ `$first`: true si l'élément est à la première position.
 - ▶ `$last`: true si l'élément est à la dernière position.

@For

```
<ol>
  @for (candidat of listeCandidats; let pair = $even; let impair = $odd; let premier = $first;
  let dernier = $last; track $index;) {
    <li>
      {{ $index }} : {{ candidat }} - Pair : {{ pair }} - Impair : {{ impair }} - Premier : {{ premier }}
      - Dernier : {{ dernier }}
    </li>
  }
</ol>
```



Bienvenue dans l'application de gestion de CV

0 : Bart - Pair : true - Impair : false - Premier : true - Dernier : false
1 : Homer - Pair : false - Impair : true - Premier : false - Dernier : false
2 : Marge - Pair : true - Impair : false - Premier : false - Dernier : true

@For

- Angular vous oblige désormais utiliser track (anciennement) trackBy afin d'identifier les éléments qui ont été modifiés

*ngFor trackBy

Reset items Change ids Clear counts

without trackBy

```
(0) Teapot  
(1) Lamp  
(2) Phone  
(3) Television  
(4) Fishbowl
```

with trackBy

```
(0) Teapot  
(1) Lamp  
(2) Phone  
(3) Television  
(4) Fishbowl
```

@if

- ▶ **@if** est associé à une expression booléenne. Si cette expression est fausse (false) alors l'élément et son contenu sont retirés du DOM, ou jamais ajoutés).
- ▶ **@if(condition)**
 - ▶ Si le booléen est true alors l'élément host est visible.
 - ▶ Si le booléen est false alors l'élément host est caché.

```
@if(listeCandidats[0] == 'Bart') {  
    <p>Ce candidat existe !</p>  
}
```

@if

- ▶ **@if** peut également être utilisé avec **@else if** et/ou **@else** selon les besoins.

```
listeCandidats = ['Bart', 'Homer', 'Marge'];
```

```
@if(!listeCandidats.length){  
    <p>Il n'y aucun candidat inscrit...</p>  
}  
@else if(listeCandidats[0] == 'Bart') {  
    <p>Ce candidat existe !</p>  
}  
@else {  
    <p>Ce candidat n'existe pas !</p>  
}
```

@switch

- ▶ La directive **@switch** est une structure conditionnelle de type switch qui s'utilise directement dans le template. Elle permet d'afficher un élément donné selon la valeur de l'expression qu'elle évalue.



Application

- ▶ Créez un composant affichant des comptes et donnant la possibilité à l'utilisateur d'ajouter un nouveau compte.
- ▶ Chaque compte est représenté par son nom et son statut (*active*, *inactive*, *unknown*). Bien évidemment, chaque nouveau compte ajouté doit apparaître dans la liste des comptes.

Account Name

Add Account

Master Account

This account is active

Set to 'active' Set to 'inactive' Set to 'unknown'

Testaccount

This account is inactive

Set to 'active' Set to 'inactive' Set to 'unknown'

Hidden Account

This account is unknown

Set to 'active' Set to 'inactive' Set to 'unknown'

A blue circular badge with a white number '2' in the center.

Les directives d'attributs

NgStyle

- ▶ Cette directive permet de modifier l'apparence de l'élément cible.
- ▶ Lorsque l'on utilise cette directive, il est nécessaire de la placer entre crochets comme ceci : [ngStyle] . Logique, puisqu'elle utilise le property Binding.
- ▶ Elle prend en paramètre un attribut représentant un objet décrivant le style à appliquer.

```
<div class="grand" [ngStyle]= "{  
    'backgroundColor' : bgColor,  
    'color' : color,  
    'fontSize' : size  
}">
```

NgStyle : Application

Créez un mini-simulateur de Microsoft Word pour gérer un paragraphe en utilisant ngStyle.



- ▶ Préparer un input de type *texte*, un input de type *number*, et un *selectbox*.
- ▶ Faites en sorte que lorsqu'on écrit une couleur dans le texte input, ça devienne la couleur du paragraphe. Et que lorsque on change le nombre dans le number input, la taille de l'écriture change également.
- ▶ Finalement ajouter une liste et mettez y un ensemble de polices. Lorsque le user sélectionne une police dans la liste, la police dans le paragraphe change.

NgClass

- ▶ Cette directive permet de modifier l'attribut class. Permet de choisir facilement parmi une liste de plusieurs classes possibles qui vous permettent de spécifier une liste d'objets incluant des conditions. Angular est capable d'utiliser la classe correcte en fonction de la véracité des conditions.
- ▶ Votre objet doit contenir des paires clé / valeur. La clé est un nom de classe qui sera appliqué lorsque la valeur (conditionnelle) est évaluée à true.
- ▶ Elle utilise le property Binding.

```
<div [ngClass]="{  
    'c1' : true,  
    'c2' : false,  
    'c3' : true  
}"> Exemple de NgClass  
</div>
```

NgClass : Autre Exemple

```
.style-1 {  
    color : green;  
    background-color: rgba(100, 100, 224, 0.596);  
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen, U  
}  
  
.style-2 {  
    color : blue;  
    background-color : rgb(217, 255, 127);  
    font-family: 'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif;  
}  
  
.style-3 {  
    color : black;  
    background-color: greenyellow;  
    font-family: 'Lucida Sans', 'Lucida Sans Regular', 'Lucida Grande', 'Lucida S  
}
```

CSS

```
<h1 [ngClass]="{  
    'style-1' : true,  
    'style-2' : false,  
    'style-3' : false  
}">  
    Bienvenue dans l'application de gestion de CV  
</h1>
```

HTML

APPLICATION

- ▶ Créer un composant qui affiche une liste de serveurs. Chaque serveur est caractérisé par un nom, un type, une date de démarrage et un statut.
- ▶ Utilisez la directive `ngClass` pour avoir le résultat ci-dessous.



Production Server medium Sat Jul 20 2019 00:00:00 GMT+0100 (CET) critical
Testing Development Server small Fri Apr 20 2018 00:00:00 GMT+0100 (CET) stable
Development Server large Wed Feb 22 2017 00:00:00 GMT+0100 (CET) offline
Nidhal Server medium Thu Apr 11 2019 00:00:00 GMT+0100 (CET) stable

CREER SA PROPRE DIRECTIVE

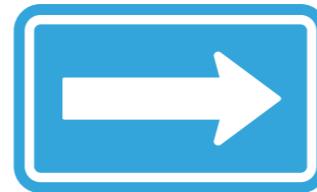
- ▶ Afin de créer sa propre « attribut directive », on a le choix entre injecter ElementRef ou utiliser un HostBinding sur la propriété que vous voulez binder.
- Exemple : `@HostBinding('style.fontFamily') family:string="Arial";`
- ▶ Si on veut associer un **événement** à notre directive on utilise un HostListener qu'on associe à un **événement** déclenchant une méthode.
- Exemple : `@HostListener('mouseenter') mouseenter() {
 this.family = "Garamond"; }`
- ▶ Afin d'utiliser le **HostBinding** et le **HostListener** il faut les importer du core d'angular

RETOUR AU PROJET

- ▶ Créez une directive qui modifie la couleur du background d'un candidat quand la souris survole son espace.



	Nidhal JELASSI
	Bart SIMPSON
	Homer SIMPSON

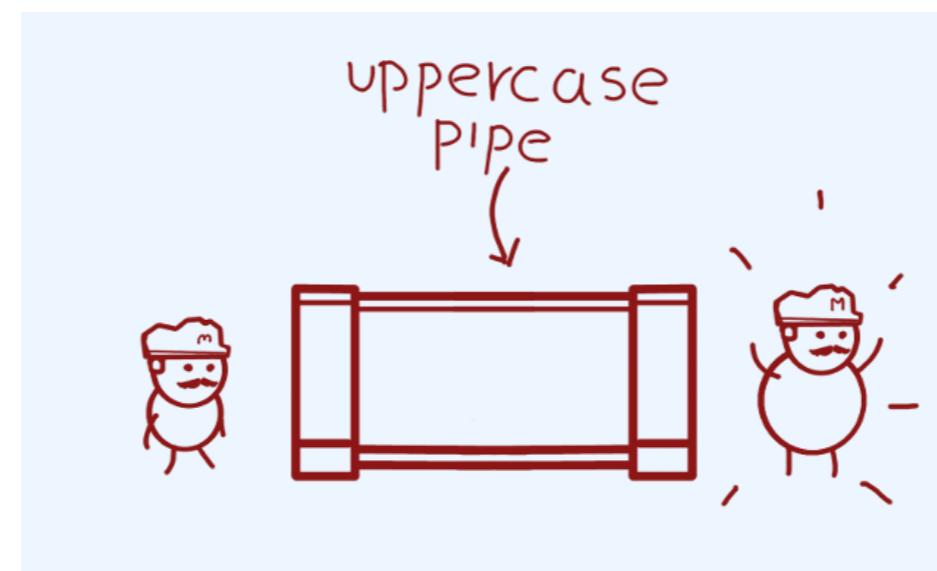


	Nidhal JELASSI
	Bart SIMPSON
	Homer SIMPSON

LES PIPES

PRÉSENTATION D'UN PIPE

- ▶ Un pipe est une fonctionnalité qui permet de formater et de transformer vos données avant de les afficher dans vos Templates.
- ▶ **Exemple** : l'affichage d'une date selon un certain format, Mettre une chaîne de caractères en majuscules, etc.
- ▶ Il existe des pipes offerts par Angular et prêt à l'emploi.
- ▶ Vous pouvez créer vos propres pipes personnalisés.



SYNTAXE

- ▶ Afin d'utiliser un pipe vous utilisez la syntaxe suivante :

`{{ variable | nomDuPipe }}`

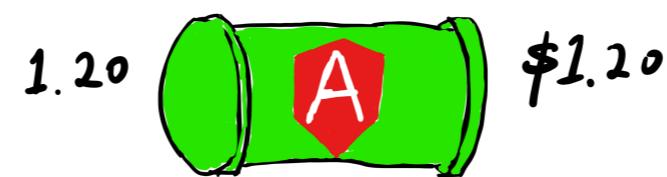
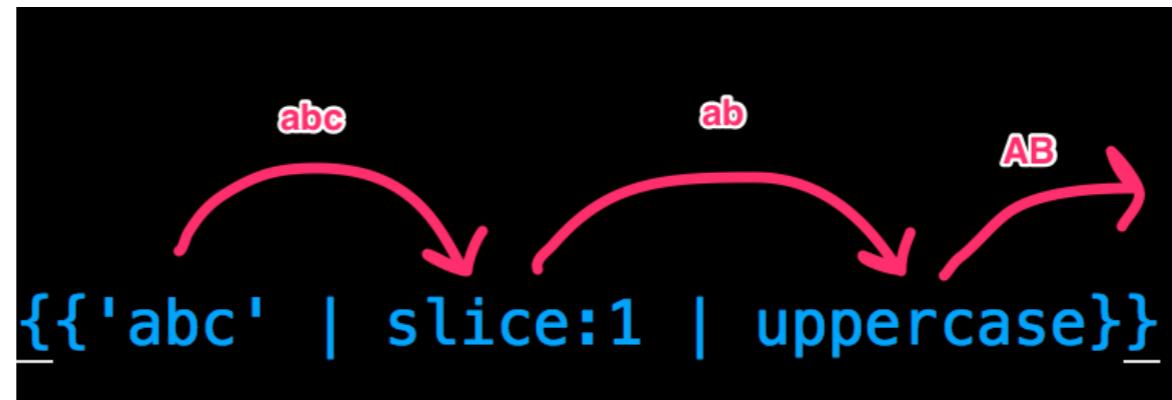
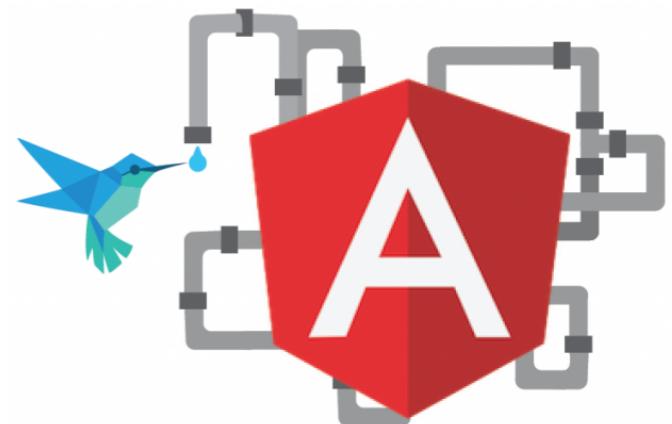
- ▶ *Exemple* : {{ maDate | date }}
- ▶ Afin d'utiliser plusieurs pipes combinés vous utilisez la syntaxe suivante :

`{{ variable | nomDuPipe1 | nomDuPipe2 | nomDuPipe3 }}`

- ▶ *Exemple* : {{ maDate | date | uppercase }}

LES PIPES DISPONIBLES PAR DÉFAUT (BUILT-IN PIPES)

- ▶ La documentation d'angular vous offre la liste des pipes prêt à l'emploi (<https://angular.io/api?type=pipe>).
 - uppercase
 - lowercase
 - titlecase
 - currency
 - date
 - json
 - percent
 - etc..



PARAMÉTRER UN PIPE

- ▶ Afin de paramétrer les pipes ajouter ':' après le pipe suivi de votre paramètre.

```
 {{ maDate | date:"MM/dd/yy" }}
```

- ▶ Si vous avez plusieurs paramètres c'est une suite de ':'

```
 {{ nom | slice:1:4 }}
```

PIPE PERSONNALISÉ

- ▶ Un pipe personnalisé est une classe décoré avec le décorateur **@Pipe**.
- ▶ Elle implémente l'interface **PipeTransform**.
- ▶ Elle doit implémenter la méthode **transform** qui prend en paramètre la valeur cible ainsi qu'un ensemble d'options.
- ▶ La méthode **transform** doit retourner la valeur transformée.
- ▶ Le pipe doit être déclaré au niveau de votre module de la même manière qu'une directive ou un composant.
- ▶ Pour créer un pipe avec le cli : **ng g p nomPipe**

EXEMPLE

```
<p class ="{{selectedPersonne.age | age}}">{{selectedPersonne.age}} </p>
```



```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'age'
})
export class AgePipe implements PipeTransform {

  transform(age: number): string {
    let color : string;
    if(age>0 && age<4)
      color = 'bg-primary text-white';
    else if(age>=4 && age <10)
      color = 'bg-warning text-white';
    else
      color = 'bg-danger text-white';
    return 'profession ' + color;
  }
}
```

APPLICATION : SUITE

- ▶ Reprenez le composant du diapo 19 et appliquez le pipe Date pour afficher la date de démarrage de chaque serveur avec ce format.
- ▶ Créez ensuite 2 pipes. Le 1er n'affichera que les 15 premiers caractères des noms des serveurs, concaténés à "...". Le 2ème, n'affichera que les serveurs dont le status est tapé dans la zone texte juste en-dessus.



stable

Testing Develop... | small | FRIDAY, APRIL 20, 2018 stable

Nidhal Server | medium | THURSDAY, APRIL 11, 2019 stable

A screenshot of a mobile application interface. At the top, there is a red-outlined button containing the word "stable". Below it, a green card displays the text "Testing Develop... | small | FRIDAY, APRIL 20, 2018" followed by a "stable" button. Another green card below it displays the text "Nidhal Server | medium | THURSDAY, APRIL 11, 2019" followed by a "stable" button. The entire interface has a light blue header bar.

RETOUR AU PROJET CV

- ▶ Créer un pipe appelé nolmage qui retourne le nom d'une image par défaut que vous stockerai dans vos assets au cas où la valeur fournie au pipe est une chaîne vide.

The screenshot shows a user interface for managing profiles. On the left, there is a list of four items, each with a small thumbnail and a name:

- Nidhal JELASSI (Thumbnail of a man)
- Bart SIMPSON (Thumbnail of Bart Simpson)
- Homer SIMPSON (Thumbnail of Homer Simpson)
- Marge SIMPSON (Thumbnail of Marge Simpson, highlighted with a red circle)

On the right, there is a detailed view of Marge Simpson's profile:

Marge Simpson
42 ans
Architecte

Auto Rotation

PIPES PURES ET IMPURES

- ▶ Par défaut, les Pipes sont purs. C'est à dire que leur méthode **transform** est une fonction pure, la valeur renvoyée ne dépend que des paramètres reçus et les appels n'ont aucun effet de bord. Autrement dit, le Pipe est "stateless".
- ▶ Les Pipes impurs sont évalués à chaque Change Detection même quand les paramètres ne changent pas. Ils sont déclarés en passant la valeur `false` au paramètre **pure** du décorateur **Pipe**.

```
@Pipe({  
  name: 'filter',  
  pure : false  
})
```

PIPES PURES ET IMPURES

Nativement, seuls les Pipes suivant sont impurs :

- ▶ **async** : Il est impur car les valeurs retournées par ce Pipe peuvent changer à n'importe quel moment étant donné qu'elles proviennent d'une source asynchrone (Observable ou Promise).
- ▶ **json** : Etant donné qu'il est principalement utilisé pour le "debug", ce Pipe est impur car il est préférable que la valeur retournée soit mise à jour même si l'immutabilité n'est pas respectée.
- ▶ **slice** : Bien que ce Pipe fonctionnerait parfaitement en étant pur, il est probablement impur pour faciliter l'adoption d'Angular par la communauté car malheureusement l'immutabilité des Arrays est rarement respectée par les développeurs Angular.

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

Chapitre 5 : Les Services et les
injections de dépendances



OBJECTIFS

- ▶ Définir un service
- ▶ Définir ce qu'est l'injection de dépendance
- ▶ Injecter un service
- ▶ Définir la portée d'un service
- ▶ Réordonner son code en utilisant les services

INTÉRÊT DES SERVICES

- ▶ Un service est une classe qui permet d'exécuter un traitement.
- ▶ Permet d'encapsuler des fonctionnalités redondantes permettant ainsi d'éviter la redondance de code.

Component 1

```
f();  
g();  
k();
```

Component 2

```
f();  
g();  
l();
```

Component 3

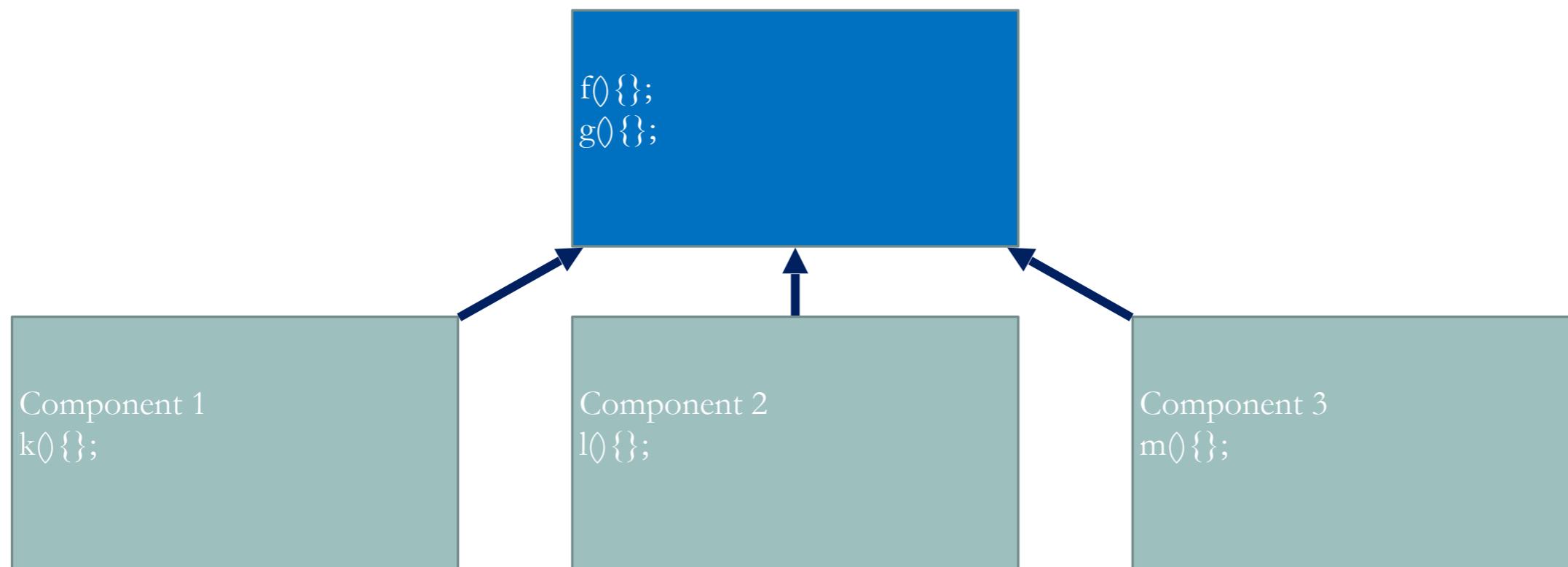
```
f();  
g();  
m();
```

Redondance de code

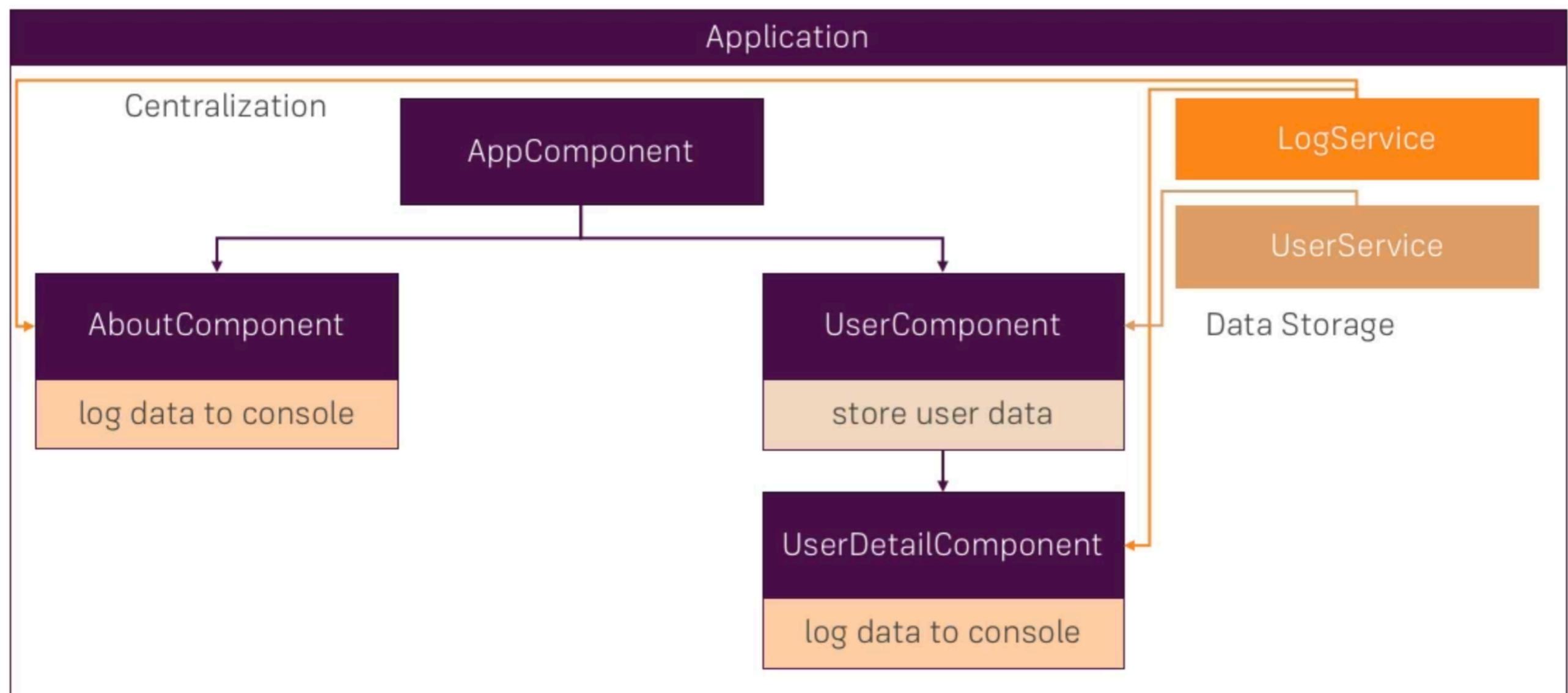
Maintenabilité difficile

Indisponibilité

INTÉRÊT DES SERVICES



EXEMPLE



QU'EST CE QU'UN SERVICE ?

- ▶ Un service est une sorte d'intermédiaire entre la vue et la logique
- ▶ Ce qui n'est pas trivial doit être écrit sous forme d'un composant
- ▶ Un service est associé à un composant en utilisant l'injection de dépendance (DI).
- ▶ Un service peut donc :
 - ▶ Interagir avec les données (fournit, supprime et modifie)
 - ▶ Interagir entre classes et composants
 - ▶ Effectuer tout traitement métier (calcul, tri, extraction ...)

CRÉATION D'UN SERVICE

- ▶ **Manuellement**
- ▶ A noter qu'un Service est comme un composant sauf qu'il n'a pas de template.

- ▶ **Via CLI**



- ▶ **ng generate service nomDuService**
- ▶ **ng g s nomDuService**

EXEMPLE DE CRÉATION D'UN SERVICE

```
import { Injectable } from '@angular/core';
@Injectable()
export class AuthService {
```



```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class AuthService {
```

```
//...Other import
import {AuthService} from "./
auth.service";
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [AuthService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

SYNTAXE

Marque une classe comme étant disponible pour l'Injector, au moment de sa création.

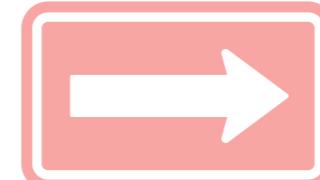
```
import { Injectable } from '@angular/core';
import { Personne } from './model/Personne';
```

```
@Injectable({
  providedIn: 'root'
})
export class PersonneService {
```

@Component, @Pipe, et @Directive sont des sous classes de @Injectable(), ceci explique le fait qu'on peut y injecter directement des dépendances.

Component *Service*
{constructor(service)}

providedIn : 'root' | 'any' | 'platform'



Détermine quels injecteurs fourniront l'injectable, en l'associant à :

- Le Module racine(root)
- Tous les Modules de l'application (any)
- A toute l'application si cette dernière est représenté par plusieurs modules racines (platform).

SERVICES / COMPOSANTS

“Do limit logic in a component to only that required for the view. All other logic should be delegated to services.”

L'INJECTION DE DÉPENDANCE

- ▶ L'injection de dépendance est un patron de conception (Pattern Design).

```
Classe A1{  
    ClasseB b;  
    ClasseC c;  
    ...  
}
```

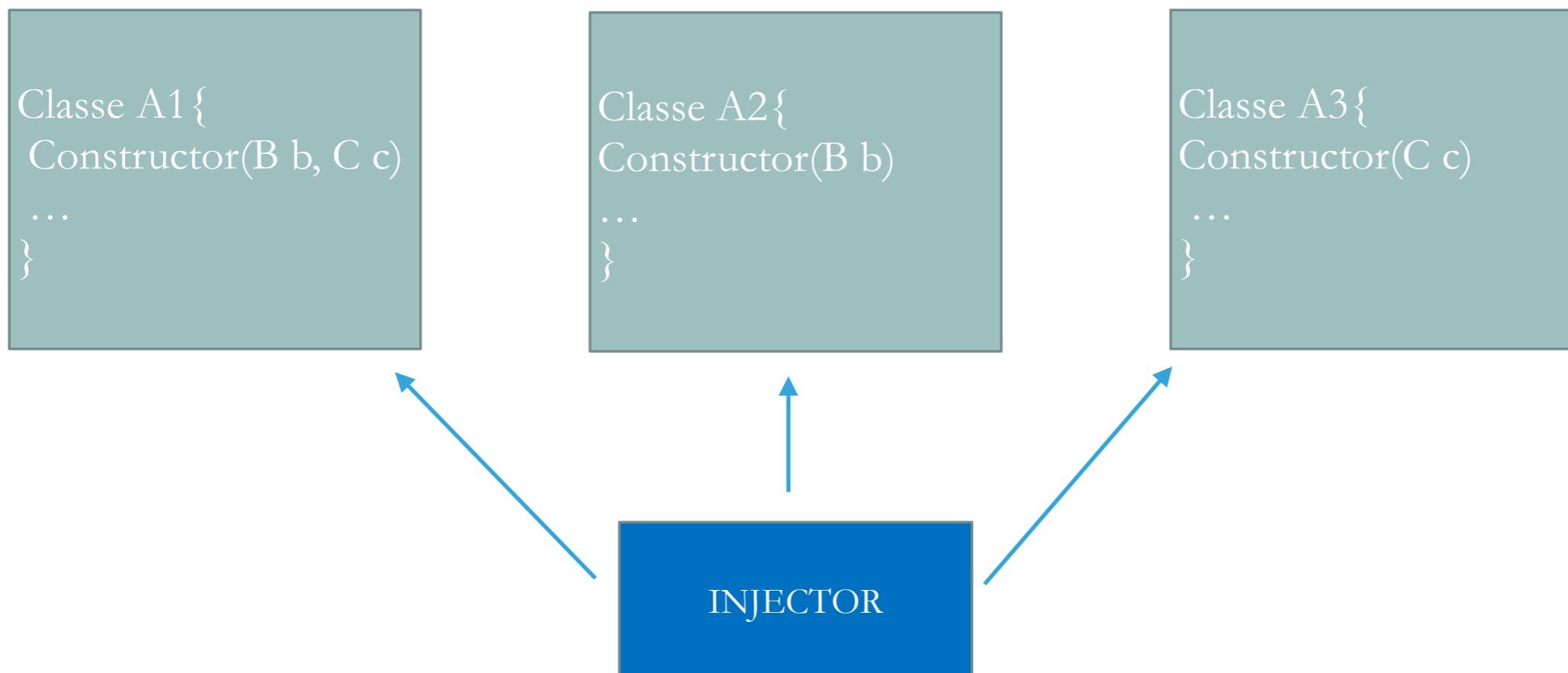
```
Classe A2{  
    ClasseB b;  
    ...  
}
```

```
Classe A3{  
    ClasseC c;  
    ...  
}
```

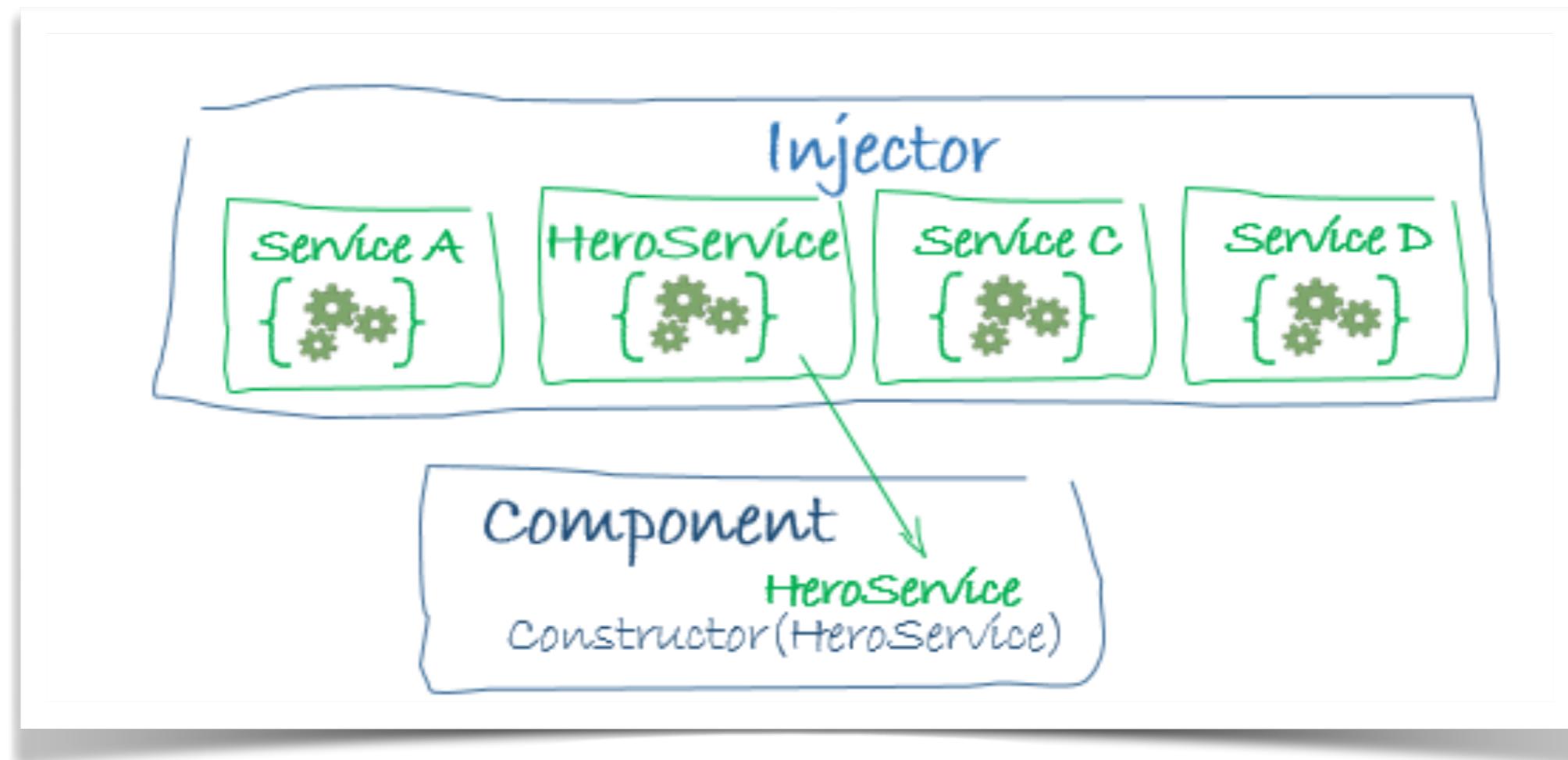
- ▶ *Que se passera t-il si on change quelque chose dans le constructeur de B ou C ?*
- ▶ *Qui va modifier linstanciation de ces classes dans les différentes classes qui en dépendent?*

SOLUTION

- ▶ Déléguer cette tache à une entité tierce.
- ▶ Il s'agit de l'**INJECTOR**



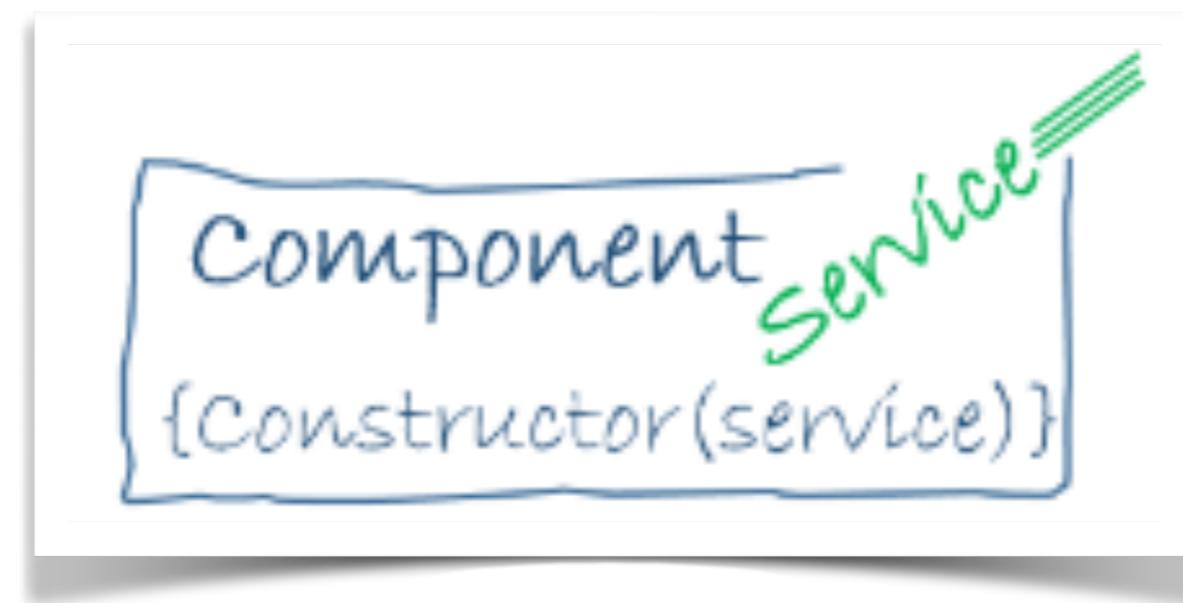
Injector



- ▶ Comment les injecter ?
- ▶ Comment spécifier à l'injecteur **quel** service injecter et **où** l'injecter ?

L'INJECTION DE DÉPENDANCE

- ▶ L'injection de dépendance utilise les étapes suivantes :
 1. **Provider un service** : Le déclarer dans le provider du module ou du composant
 2. **Passer le service à l'Injector** : Spécifier le service comme paramètre du constructeur de l'entité qui en a besoin.



L'INJECTION DE DÉPENDANCE

- ▶ Pour injecter un service dans un composant (ou une directive ou autre), on peut passer soit par :
 - ▶ Le constructeur de la classe en question.

```
export class CvComponent {  
    constructor(private firstSer: FirstService) {}
```

- ▶ La méthode inject.

```
export class CvComponent {  
    firstSer = inject(FirstService);
```

EXEMPLE

Delivering Services to Components



Provider

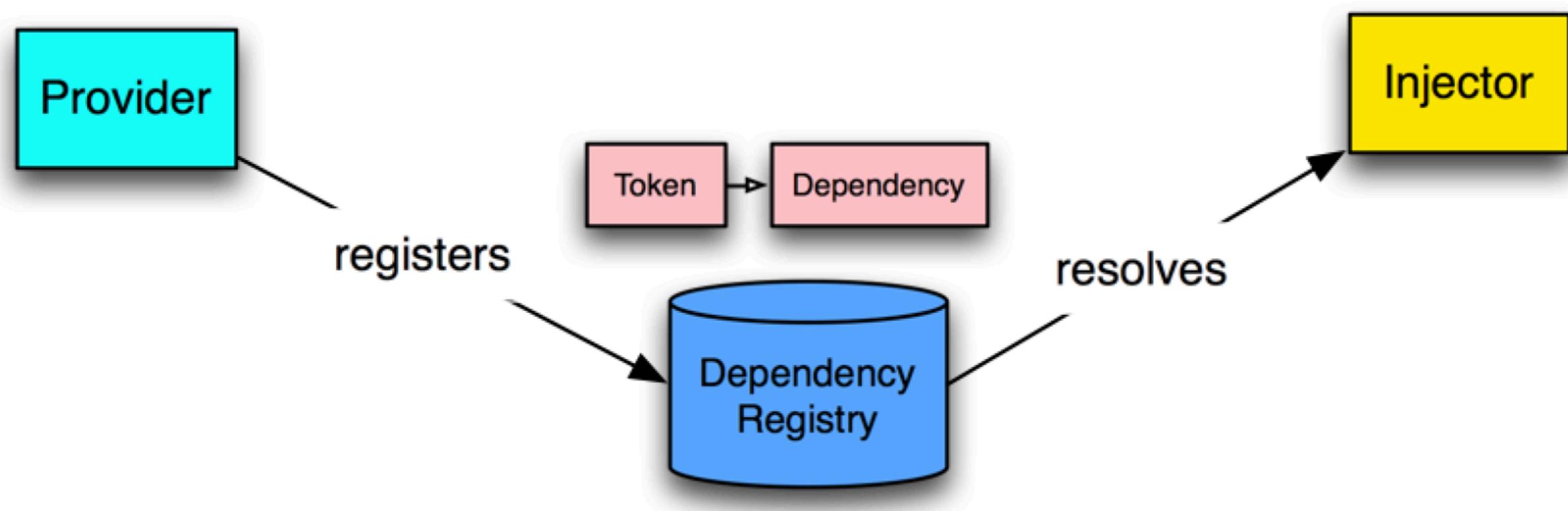


Injector

```
export class DashboardComponent  
  constructor(private dataService: DataService) { }  
}
```

An orange arrow points from the provider icon to the `DataService` parameter in the `constructor` of the `DashboardComponent` code block.

SCHÉMA RÉCAPITULATIF



EXEMPLE

```
import { Component, OnInit } from '@angular/core';
import { ListPersonneService } from '../list-personne.service';
import { Personne } from '../model/personne';

@Component({
  selector: 'app-cv',
  templateUrl: './cv.component.html',
  styleUrls: ['./cv.component.css'],
  providers : [ListPersonneService]
})
export class CvComponent implements OnInit {
  selectedPersonne : Personne;
  listPersonne : Personne[];
  constructor(private persServ : ListPersonneService) { }
```

CHARGEMENT AUTOMATIQUE DU SERVICE

- ▶ Que est le rôle du décorateur **@Injectable** ?
- ▶ A partir de Angular 6 vous pouvez ne plus utiliser le provider du module afin de charger votre service mais le faire directement au niveau du service à travers la propriété **providedIn** de l'annotation **@Injectable**. Vous pouvez charger le service dans toute l'application via le mot clé root.
- ▶ Si vous voulez charger le service dans un module particulier vous l'importer et vous le mettez à la place de 'root'.

@INJECTABLE

- ▶ C'est un décorateur permettant de rendre une classe injectable
- ▶ Une classe est dite injectable si on peut y injecter des dépendances
- ▶ @Component, @Pipe, et @Directive sont des sous classes de @Injectable, ceci explique le fait qu'on peut y injecter directement des dépendances.
- ▶ Si vous n'allez injecter aucun service dans votre service, cette annotation n'est plus nécessaire.
- ▶ Néanmoins, son utilisation est très recommandé par Angular.

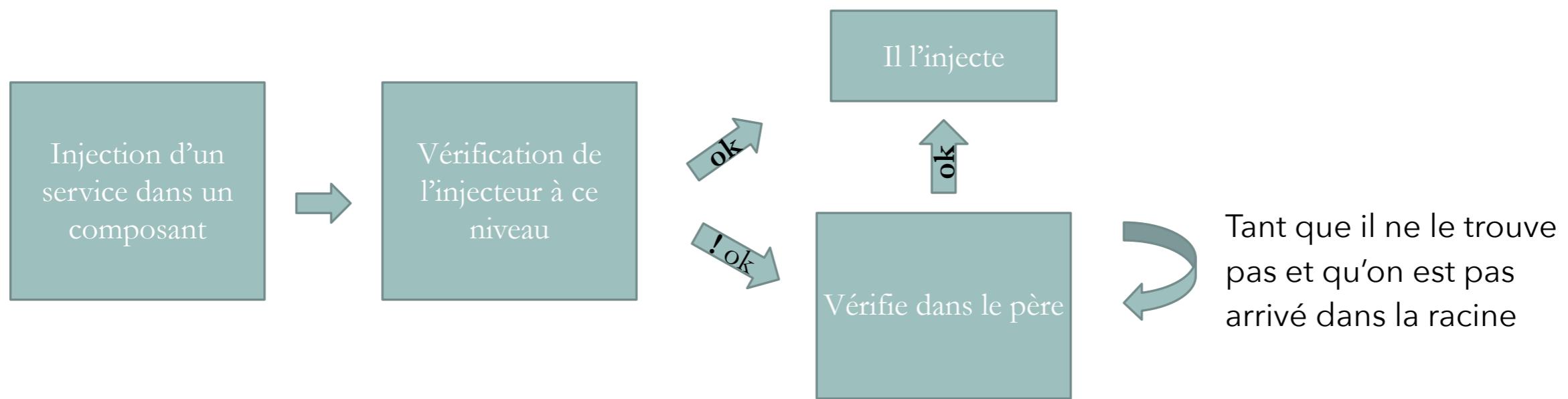
AVANTAGES DU providedIn

On peut citer deux avantages au providedIn :

- ▶ **Lazy loading** : Ne charger le code des services qu'à la première injection
- ▶ Permettre le **Tree-Shaking** des services non utilisés : Si le service n'est jamais utilisé, son code ne sera entièrement retiré du build final.

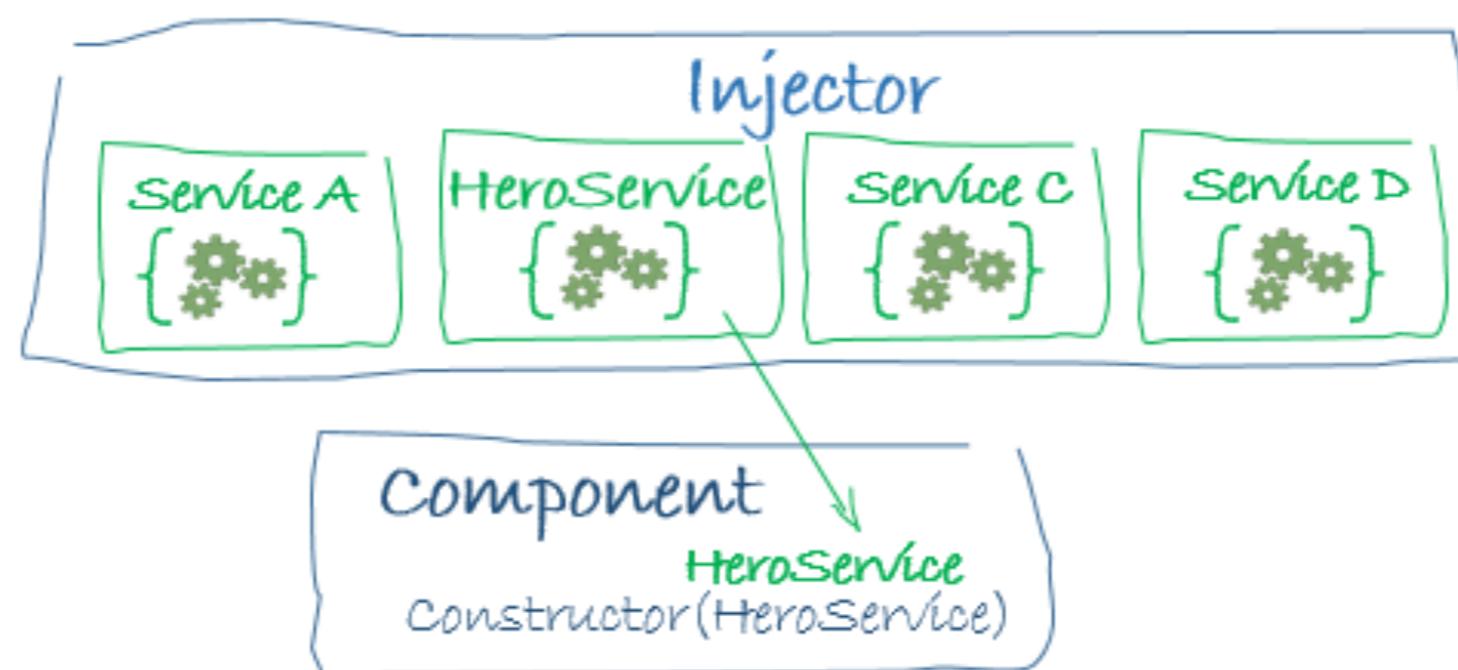
DI HIÉRARCHIQUE

- ▶ Le système d'injection de dépendance d'Angular est **hiérarchique**.
- ▶ Un arbre d'injecteur est créé. Cet arbre est parallèle à l'arbre de composant.
- ▶ L'algorithme suivant permet la détection de l'injecteur adéquat selon le diagramme suivant :



L'INJECTION DE DÉPENDANCE

- ▶ Si un service est déclaré au niveau du Module et qu'il est déclaré dans le provider d'un composant c'est la déclaration la plus spécifique qui l'emporte.



- ▶ ***L'injection d'un service dans un autre est la même que pour un composant.***

RETOUR AU PROJET CV...

- ▶ Ajouter aussi un composant pour afficher la liste des cvs recrutés ainsi qu'un service **RecruteService** qui gérer les personnes recrutés.
- ▶ Au click sur le bouton **Recruter** d'un Cv, la personne est ajouté à la liste des personnes embauchées et **une liste des recrutés** apparaît.
- ▶ Dans cette liste, chaque personne recrutée est représentée par son image, son nom et son prénom.

Les deux captures d'écrans des prochains diapos vous donneront un aperçu du résultat attendu...

CAPTURE 1

The screenshot shows a profile page for 'bart simpson'. At the top, there is a small cartoon image of Bart Simpson. Below it, the name 'bart simpson' is displayed. To the right, a quote reads: "To be or not to be, this is my awesome motto!" Below this section, there are three more entries: 'homer simpson' with a cartoon image of Homer Simpson, and 'marge simpson' with a cartoon image of Marge Simpson. To the right of these entries, there is a 'Job Description' section with the text: 'Web design, Adobe Photoshop, HTML5, CSS3, Corel and many others...'. Below this, there are three statistics: '235 Followers', '114 Following', and '35 Projects'. A large red arrow points downwards from the 'Following' statistic towards a green button labeled 'Recruter'.

"To be or not to be, this is my awesome motto!"

Job Description

Web design, Adobe Photoshop, HTML5, CSS3, Corel and many others...

235
Followers

114
Following

35
Projects

Recruter + Details

CAPTURE 2

The screenshot displays a user interface for a character profile. On the left, there is a sidebar with three items: "bart simpson" (with an icon of Bart Simpson), "homer simpson" (with an icon of Homer Simpson), and "marge simpson" (with an icon of Marge Simpson). On the right, a detailed profile for "Bart Simpson" is shown. The profile includes a large circular portrait of Bart, his name "Bart Simpson", his age "23 ans", his occupation "Ingénieur", and a "Auto Rotation" button at the bottom.

bart simpson

homer simpson

marge simpson

Bart Simpson
23 ans
Ingénieur
Auto Rotation

Liste des recrues

The screenshot shows a list of recruits. It includes two items: "bart simpson" (with an icon of Bart Simpson) and "marge simpson" (with an icon of Marge Simpson). The background of this section is pink.

bart simpson

marge simpson

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

Chapitre 6 : Le routing



OBJECTIFS

- ▶ Définir le routeur d'Angular
- ▶ Définir une route
- ▶ Déclencher une route à partir d'un composant
- ▶ Ajouter des paramètres à une route
- ▶ Récupérer les paramètres d'une route à partir du composant.
- ▶ Manipuler les QuerParameters
- ▶ Gérer les routes inexistantes

UN SYSTÈME DE ROUTING... POURQUOI ?

- ▶ Un système de routing permet d'associer une route (url - path) à un traitement particulier.
- ▶ Angular produit une **SPA**. Pourquoi alors parle-on de route ?
- ▶ L'objectif est donc de :
 - ▶ Séparer différentes fonctionnalités du système
 - ▶ Maintenir l'état de l'application
 - ▶ Ajouter des règles de protection

UTILITÉ

- ▶ *Que risque t-on d'avoir si on n'utilise pas un système de routing ?*

- ▶ On ne peut plus rafraîchir notre page
- ▶ Perdre tout son contexte à chaque changement de page
- ▶ Impossibilité de partager nos pages

CRÉATION D'UN SYSTÈME DE ROUTING

1. Indiquer au routeur comment composer les urls en ajoutant dans le head la balise suivante : <**base href="/"**>
2. Créer un fichier 'app.routing.ts'
3. Importer le service de routing d'Angular : import { **RouterModule**, **Routes** } from '@angular/router';
 - ▶ Le **RouterModule** va permettre de configurer les routes dans notre projet.
 - ▶ Le **Routes** va permettre de créer les routes

CRÉATION D'UN SYSTÈME DE ROUTING

4. Créer une **constante** qui est un **tableau d'objets** de type **Routes** représentant chacun la route à décrire.
5. Intégrer les routes à notre application dans le **AppModule** à travers le **RouterModule** et sa méthode **forRoot**.

```
import { HomeComponent } from './home/home.component';
import { ServersComponent } from './servers/servers.component';
import { UsersComponent } from './users/users.component';
import { Routes, RouterModule } from '@angular/router';

const routes : Routes = [
  {path: '' ,component: HomeComponent},
  {path: 'servers', component: ServersComponent},
  {path: 'users' ,component: UsersComponent},
];

export const ROUTING = RouterModule.forRoot(routes);
```

```
@NgModule({
  declarations: [
    AppComponent,
    ServersComponent,
    ServerComponent,
    EditServerComponent,
    UsersComponent,
    UserComponent,
    HomeComponent,
    NavbarComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ROUTING
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

UTILISATION DU SYS DE ROUTING

- ▶ Pour indiquer à Angular où est-ce qu'il doit charger les vues spécifiques aux routes nous utilisons le **router-outlet**.
- ▶ Router-outlet est une directive qui permet de spécifier l'endroit où la vue va être chargée.
- ▶ Sa syntaxe est **<router-outlet></router-outlet>**

```
<div class="container">
  <div class="row">
    <div class="col-md-10">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

SYNTAXE MINIMALISTE D'UNE ROUTE

- ▶ Une route est un objet.
- ▶ Les deux propriétés essentielles sont **path** et **component**.
- ▶ **path** permet de spécifier l'URI. Cette url ne doit pas commencer par un **/**.
- ▶ **component** permet de spécifier le composant à exécuter.

```
{path: '', component: CvComponent},  
{path: 'onlyHeader', component: HeaderComponent}
```

RETOUR AU PROJET

- ▶ Configurer votre routing
- ▶ Vérifier le fonctionnement de votre routing en modifiant l'url suivant les paths que vous avez spécifié dans votre système de routing.



DÉCLENCHER UNE ROUTE (ROUTERLINK)

- ▶ L'idée intuitive pour déclencher une route est d'utiliser la balise **a** et son attribut href. Quel problème cette idée risque de poser ?
- ▶ L'utilisation de **<a href >** va déclencher le chargement de la page ce qui est **inconcevable pour une SPA**.
- ▶ La solution proposée par le **router** d'Angular est l'utilisation de la propriété **routerLink** qui comme son nom l'indique liera cette dernière à la route que nous souhaitons déclencher sans recharger la page.
- ▶ Exemple :

```
<li><a [routerLink]= "[ 'cv' ]" routerLinkActive="active">Gérer les  
cvs</a></li>
```

ROUTERLINK : DIRECTIVE VS PROPERTY

- ▶ RouterLink a deux variantes : une directive et une property.
 1. On opte pour la directive quand la route cible est statique (sans paramètres).
 2. On fait du property binding quand la route cible est dynamique.

```
<li class="nav-item">
  <a class="nav-link" [routerLink]=["'servers', id"] 2
    routerLinkActive="active" >
      Serveurs</a>
    </li>

<li class="nav-item">
  <a class="nav-link" routerLink="users" 1
    routerLinkActive="active">
      Utilisateurs</a>
    </li>
```

DÉCLENCHER UNE ROUTE (ROUTERLINK)

- ▶ **routerLinkActiveOptions** + **exact** permet de définir que la classe CSS active est ajoutée si le chemin correspond exactement. Sans cela par exemple pour la route « /users » ayant des childrens, le lien serait toujours actif que l'on sur « /users/3 » ou « users/about ».

```
<ul class="nav nav-tabs">
  <li class="nav-item">
    <a class="nav-link" [routerLink]="['']"
      routerLinkActive="active"
      [routerLinkActiveOptions]={`${exact: true}`}>
      Accueil</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" [routerLink]="['servers']"
        routerLinkActive="active" >
        Serveurs</a>
      </li>

      <li class="nav-item">
        <a class="nav-link" [routerLink]="['users']"
          routerLinkActive="active">
          Utilisateurs</a>
        </li>
      </ul>
```

DÉCLENCHER UNE ROUTE À PARTIR DU COMPOSANT

- ▶ Afin de déclencher une route à travers le composant on utilise l'objet Router et sa méthode **navigate** ou **navigateByUrl**.
- ▶ Cette méthode prend le même paramètre que le **routerLink**, à savoir un tableau contenant la description de la route.
- ▶ Afin d'utiliser le Router, il faut l'importer de **@angular/router** et l'injecter dans votre composant.

```
onLoadServer(id: number) {
    // Traitement à faire
    this.router.navigate(['/servers', id]);
    // ou
    this.router.navigateByUrl(`/servers/${id}`);
}
```

APPLICATION



- ▶ Créer un composant appelé RouterSimulator
- ▶ Dans ce composant créer une liste déroulante contenant le nom des différentes routes de votre application.
- ▶ Ajouter ce composant au même niveau que le header et que le <router-outlet>.
- ▶ En sélectionnant le nom d'un composant, il doit apparaître dans le <router-outlet> simulant ainsi le fonctionnement d'un routeur.

LES PARAMÈTRES D'UNE ROUTE

- ▶ Afin de spécifier à notre router qu'un segment d'une route est un paramètre, il suffit d'y ajouter ':' devant le nom de ce segment.
- ▶ Exemple : **/user/:id** permet de dire que la route contient au début **user** ensuite un paramètre de route appelé **id**.

```
const routes : Routes = [
  {path: '' ,component: HomeComponent},
  {path: 'servers' , component: ServersComponent},
  {path: 'servers/:id' , component: ServerComponent},
  {path: 'users' ,component: UsersComponent},
];
```

RÉCUPÉRATION DES PARAMÈTRES

- ▶ Afin de récupérer les paramètres d'une route au niveau d'un composant on doit procéder comme suit :
 1. Importer **ActivatedRoute** qui nous permettra de récupérer les paramètres de la route.
 2. Injecter **ActivatedRoute** au niveau du composant (c'est un service).
 3. Affecter le paramètre à une variable du composant en s'inscrivant avec la méthode **subscribe** à **l'observable** retourné par **paramMap** de notre **ActivatedRoute**. Cette variable retourne un tableau de l'ensemble des paramètres.
- ▶ Syntaxe : `activatedRoute.paramMap.subscribe(params=>{this.monParam=params.get('id')});`

RÉCUPÉRATION DES PARAMÈTRES



QUELLE EST LA DIFFÉRENCE
ENTRE LES 2 TECHNIQUES ?

```
export class UserComponent implements OnInit, OnDestroy {  
  user: {id: number, name: string};  
  paramsSubscription: Subscription;  
  
  constructor(private route: ActivatedRoute) {}  
  
  ngOnInit() {  
    this.user = {  
      id: this.route.snapshot.params['id'],  
      name: this.route.snapshot.params['name']  
    };  
  
    this.paramsSubscription = this.route.params  
      .subscribe(  
        (params: Params) => {  
          this.user.id = params['id'];  
          this.user.name = params['name'];  
        }  
      );  
  }  
  
  ngOnDestroy() {  
    this.paramsSubscription.unsubscribe();  
  }  
}
```

1ère technique (avec snapshot)

2ème technique (avec un observable)

SUBSCRIBE / UNSUBSCRIBE

La méthode **subscribe** permet de s'inscrire à un observable.

- ▶ **Problème** : Cette souscription reste (parfois) valide même après la disparition de la variable ce qui sature la mémoire pour rien.
- ▶ **Solution** : Se Désinscrire à la mort du composant donc dans le `ngOnDestroy()`.

A importer
depuis rxjs

```
export class ServerComponent implements OnInit, OnDestroy {  
  subscription : Subscription;  
  
  constructor(private activatedRoute : ActivatedRoute) { }  
  
  ngOnDestroy() {  
    this.subscription.unsubscribe();  
  }  
  
  ngOnInit() {  
    this.subscription = this.activatedRoute.params.subscribe(  
      (p: Params) => {  
        // ...  
      }  
    );  
  }  
}
```

ROUTE RELATIVE OU ABSOLUE

- ▶ Si on préfixe la valeur passé au routerLink avec :
 - '/', la route sera absolue.
 - Rien ou '.', le Router regardera dans les enfants du ActivatedRoute.
 - '../', le Router regardera dans le niveau au-dessus dans l'arbre de routes.
- ▶ Quand on désire passer un path relativement au route dans laquelle on est, il est possible de passer à la méthode navigate, en argument, un objet dont la clé est relativeTo et la valeur est une instance de ActivatedRoute.

```
this.router.navigate(['edit'], {relativeTo: this.activatedRoute});
```

LES QUERY PARAMETERS

- ▶ Les **queryParameters** sont les paramètres envoyé à travers une requête **GET**.
- ▶ Identifié avec le **?**. S'il y en a plusieurs, ils sont séparer par un **&**.



- ▶ Afin d'insérer un **queryParameters** on dispose de deux méthodes :

1ère méthode :

- ▶ On ajoute dans la méthode navigate du Router un second paramètre de type objet.

LES QUERY PARAMETERS

- ▶ L'une des propriétés de cet objet est aussi un objet dont la clé est queryParams dont le contenu est aussi un objet content les identifiants des queryParams et leurs valeurs.

```
this.router.navigate(['/about', this.id], {queryParams: { 'qpVar': 'je suis un qp' } });
```

2ème méthode :

- ▶ On intègre à notre routerLink de la manière suivante :

```
<a [routerLink]="/about/10" [queryParams]="{qpVar: 'je suis un qp bindé avec le routerLink'}">About</a>
```

RÉCUPÉRER LES QUERYPARAMETERS

- ▶ Les **queryParameters** sont récupérable de la même façon que les paramètres.
- ▶ Afin de récupérer les paramètres d'une root au niveau d'un composant on doit procéder comme suit :
- ▶ Importer **ActivatedRoute** qui nous permettra de récupérer les paramètres de la root.
- ▶ Injecter **ActivatedRoute** au niveau du composant.
- ▶ Affecter le paramètre à une variable du composant en utilisant la méthode **subscribe** pour se souscrire à **params** de notre **ActivatedRoute**.
- ▶ Syntaxe : activatedRouter.**queryParams**.**subscribe**(
 (queryParam:**any**)=>(**this.monQp**=queryParam['**qpVar**'])
);

CHILD ROUTES

- ▶ Certains composants ne sont visibles qu'à l'intérieur d'autres composants.
- ▶ Prenons l'exemple d'un objet Personne. En accédant à la route **/personne/:id** nous avons l'affichage de la personne et nous aimerais avoir deux boutons. Un pour éditer la Personne (route **/personne/:id/editer**). L'autre pour afficher ces détails (route **/personne/:id/aperçu**).
- ▶ L'idée est de préfixer nos routes.
- ▶ Afin de mettre en place ce processus nous procédonss comme suit :
 - ▶ Nous définissons le préfixe avec la propriété **path**.
 - ▶ Nous y ajoutons la propriété **children** qui contiendra le tableau des routes. Chaque route de ce tableau sera préfixé avec la route définie dans path.

CHILD ROUTES + TEMPLATE

```
const routes : Routes = [
  {path: '', component: HomeComponent},
  {path: 'servers', component: ServersComponent, children: [
    {path: ':id', component: ServerComponent},
    {path: ':id/edit', component: EditServerComponent}
  ]},
];
```

- ▶ Supposons que nous voulons avoir un Template central avec des données fixes et des parties variables dans le même template.
- ▶ En changeant les routes, le contenu principal doit rester le même et la partie variable doit changer selon la route.

CHILD ROUTES

Afin de mettre en place ce processus nous procédons comme suit :

- ▶ Nous définissons le préfixe avec la propriété **path**. On lui associe le composant parent.
- ▶ Nous y ajoutons la propriété **children** qui contiendra le tableau des routes. Chaque route de ce tableau sera préfixé avec la route définie dans path.
- ▶ Nous ajoutons la balise `<router-outlet></router-outlet>` dans le Template père.

APPLICATION

- ▶ Créez une application qui permet gérer l'affichage de serveurs et d'utilisateurs. En cliquant sur un serveur, on peut consulter ses informations. De même pour les utilisateurs. Un seul serveur peut être édité, c'est celui qui a pour id 3.

Serveur de production
Serveur de test
Serveur de développement

Serveur de production
Le statut du serveur est offline
Editer
Impossible d'éditer ce serveur

RedirectTo

- ▶ Afin de rediriger une route il suffit d'ajouter une propriété dans l'objet route qui est **redirectTo**. Cette propriété permet d'indiquer vers quelle route le **path** doit être redirigé. Si la route n'a pas encore été matché, alors les routes commençant par ce path seront redirigées.
- ▶ Une autre propriété peut être utilisé qui est la propriété **pathMatch**. Cette propriété permet de définir comment le matching des path est exécuté. Avec la valeur '**full**', elle spécifie au routeur de ne faire la redirection que si le path exact est matché.

RedirectTo

```
const APP_Routes:Routes = [  
  {path: '', component: HomeComponent},  
  {path: 'about', redirectTo: '/', pathMatch: 'full'},  
  {path: 'about/:param', component: AboutComponent},  
  {path: 'about/:param', component: AboutComponent,  
   children: FILS_ROUTE},  
]
```

ERROR 404

- ▶ Afin de rediriger une route inexistante vers une page d'erreur, il suffit de garder la même syntaxe de redirection et de mettre dans la propriété path '**'.
 - ▶

```
const APP_ROUTE: Routes = [  
    {path: "", redirectTo: 'cv', pathMatch: 'full'} ,  
    {path: 'lampe', component: ColorComponent} ,  
    {path: 'login', component: LoginComponent} ,  
    {path: 'error', component: ErrorPageComponent} ,  
    {path: '**', component: ErrorPageComponent }  
];
```

RETOUR AU PROJET...

- ▶ Créer un composant pour ajouter un navbar à notre application.
- ▶ Ajouter les fonctionnalités suivantes à votre CV:
 - ▶ Une page détail qui va afficher les détails d'un CV.
 - ▶ Un bouton dans chaque CV qui au clic vous renvoie vers la page détails.
 - ▶ Dans la page détail, un bouton delete qui au click supprime ce cv et vous renvoi à la liste des cvs.

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

Chapitre 7 : Les Forms



OBJECTIFS

- ▶ Créer un formulaire
- ▶ Ajouter des validateurs
- ▶ Appréhender les classes CSS générées par le ngForm
- ▶ Manipuler l'objet ngForm
- ▶ Manipuler les contrôles du formulaire

APPROCHES

- ▶ Etant un framework front-end complet, Angular possède son propre ensemble de bibliothèques pour la création de formulaires complexes.
- ▶ Angular permet de :
 - ▶ Modéliser le formulaire.
 - ▶ Permettre la gestion et la validation du formulaire.
- ▶ La dernière version d'Angular dispose de deux stratégies puissantes de création de formulaires:
 - ▶ **Approche basée Template** (Template-driven forms)
 - ▶ **Approche réactive** (Reactive forms)

2 APPROCHES

D'après la documentation officielle d'Angular :

- ▶ Les **Reactive forms** sont plus robustes, plus évolutifs, réutilisables et testables. Si les formulaires constituent un élément clé de notre application, c'est le choix idéal.
- ▶ Les **Template-driven forms** sont utiles pour ajouter un formulaire simple à une application, tel qu'un formulaire d'inscription. Ils sont faciles à ajouter à une application, mais ils ne sont pas aussi évolutifs que les formulaires réactifs. C'est un choix idéal si nous avons des exigences de base et une logique pouvant être gérées uniquement dans le template.

TEMPLATE- DRIVEN FORMS

APPROCHE BASÉE TEMPLATE

1. Importer le module **FormsModule** dans le AppModule.
2. Angular détecte automatiquement un objet **form** à l'aide de la balise **FORM**. Cependant, il ne détecte aucun des éléments (inputs).
3. Spécifier à Angular quel sont les éléments (contrôles) à gérer. Pour chaque élément (i) ajouter la **directive** angular **ngModel**, (ii) identifier l'élément avec un nom permettant de le détecter et de l'identifier dans le composant.
4. Associer l'objet représentant le formulaire à une variable en utilisant le référencement interne **#** et la directive **ngForm**

EXEMPLE

```
<form  
  (ngSubmit)="Show(f)"  
  #f="ngForm">
```

Template

```
export class TestComponent {  
  Show(f: NgForm) {  
    console.log(f);  
  }  
}
```

Component.ts

@ViewChild

- ▶ L'autre manière d'accéder à notre formulaire depuis la classe Typescript est de faire appel à @ViewChild, déjà vu dans le chapitre dédié aux composants.

```
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
```

```
export class LoginComponent implements OnInit {
  name = {first: 'Nidhal', last: 'Jelassi'};

  @ViewChild('f') myForm : NgForm;

  onSubmit(f: NgForm) {
    console.log(this.myForm.value);
    // {name: {first: 'Nidhal', last: 'Jelassi'}, email: ''}
  }
}
```

PROPRIÉTÉS DES ÉLÉMENTS

- ▶ Afin de valider les propriétés des différents contrôles, Angular utilise des attributs et des directives tel que :
 - ▶ required
 - ▶ email
 - ▶ etc...
- ▶ La propriété **valid** de **ngForm** permet de vérifier si le formulaire est valide ou non en se basant sur les validateurs qu'ils contient.

```
<form #f="ngForm">
  <label for="nom">Pseudo</label>
  <input ngModel required type="text" class="form-control" name="name">

  <label for="prenom">Email</label>
  <input ngModel email type="text" class="form-control" name="email">
```

CLASSES CSS

- ▶ En détectant le formulaire, Angular décore les différents éléments du formulaire avec des classes (booléennes) qui informe sur leur état :
- ▶ **ng-dirty** : informe sur le fait que l'une des propriétés du formulaire a été modifié ou non
- ▶ **ng-valid / ng-invalid** : informe si le formulaire est valide ou non
- ▶ **ng-untouched / ng-touched** : informe si le formulaire est touché ou non
- ▶ **ng-pristine** : le formulaire n'a pas été modifié, c'est l'opposé du dirty

Dr IQ

TODO: remove this: form-control ng-untouched ng-pristine ng-valid

APPLICATION

- ▶ Créer un formulaire d'authentification contenant les champs suivants :
 - ▶ Email
 - ▶ Password
 - ▶ Envoyer (Un bouton)
- ▶ Si un champ est invalide alors il devra avoir une bordure rouge.
- ▶ Un champ vide et non encore modifié ne peut avoir de bordure rouge que s'il a été touché. Le bouton « envoyer » ne doit être cliquable que si le formulaire est valide. Utiliser le binding sur la propriété disable.



Contrôle du formulaire et ses propriétés

- ▶ Pour accéder à l'objet form et ces propriétés, nous avons utilisé **#notreForm= "ngForm"**
- ▶ Pour les champs du formulaire, pour les récupérer **au sein du formulaire**, c'est la même chose sauf qu'au lieu du ngForm c'est un ngModel : **#notreChamp='ngModel'**

```
<form #f="ngForm">
  <label for="name">Login</label>
    <input #in="ngModel" ngModel email required type="text" class="form-control" name="email">
```

Contrôle du formulaire et ses propriétés

- ▶ Ceci nous donne accès à des propriétés qui portent (quasiment) le même nom que les classes CSS, évoquées dans un diapo précédent, sans le préfixe 'ng-' :
 - Pristine, dirty, valid, invalid, touched, intouched.
 - Toutes ces propriétés ont une valeur booléennes.

```
<input #loginInput="ngModel" required email type="text" ngModel name="email" class="form-control">

<div *ngIf="loginInput.invalid && loginInput.touched" class="alert alert-danger">
  Veuillez saisir une adresse mail valide
</div>
```

SUITE DE L'EXERCICE PRÉCÉDENT

- ▶ Ajouter un petit message d'erreur qui devra s'afficher sous le champ de l'email s'il est invalide.
- ▶ Ce champ ne devra apparaître que si l'utilisateur accède ou modifie le champ email.
- ▶ Le password devra avoir au moins 6 caractères. Si l'utilisateur en entre moins, un message d'erreur va s'afficher également.



Login

Mot de passe

Se connecter

Login

Votre adresse mail est invalide

Mot de passe

Le mot de passe doit contenir au minimum 6 caractères

Se connecter

ERRORS

- ▶ Toutes les erreurs de validations générées par un élément du formulaire sont accessibles via l'objet errors et qui retournent des valeurs booléennes.

```
<label for="pseudo"></label>
<input #pseudo="ngModel" ngModel
required minlength="6" pattern="Nidhal Jelassi"
type="text" class="form-control" id="pseudo" name="pseudo">
<div class="alert alert-danger" *ngIf="pseudo.invalid && pseudo.touched">
  <p *ngIf="pseudo.errors.required">This Field is required</p>
  <p *ngIf="pseudo.errors.minLength">This Field must be at least 6 characters</p>
  <p *ngIf="pseudo.errors.pattern">This Field must obly be Nidhal Jelassi</p>
</div>
```

ASSOCIER DES VALEURS PAR DÉFAUT AUX CHAMPS

- ▶ Pour associer des valeurs par défaut au champs d'un formulaire associé à Angular il faut le faire à partir du composant.
- ▶ Afin de gérer les valeurs du formulaire à partir du composant il faut faire du binding.
- ▶ Au lieu d'avoir juste la directive `ngModel` associé à l'élément du formulaire, on utilisera le property binding avec `[ngModel]`
- ▶ Il est même possible de faire du two-way binding avec `[(ngModel)]` en cas de besoin.

GROUPING FORM

- ▶ Afin de grouper l'ensemble des contrôles (propriétés/champs) d'un formulaire, on peut utiliser la technique du « *grouping form controls* ».
- ▶ Il suffit d'ajouter la directive `ngModelGroup` dans la div qui englobe les propriétés à grouper.
- ▶ Afin d'accéder à cet objet vous pouvez le référence localement en utilisant le mot clé `ngModelGroup`

```
<div  
    ngModelGroup= "user"  
    #userData= "ngModelGroup"  
>
```

EXEMPLE : ngModelGroup

```
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
  <p *ngIf="nameCtrl.invalid">Name is invalid.</p>

  <div ngModelGroup="name" #nameCtrl="ngModelGroup">
    <input name="first" [ngModel]="name.first" minlength="2">
    <input name="last" [ngModel]="name.last" required>
  </div>

  <input name="email" ngModel>
  <button>Submit</button>
</form>
```

Nidhal

Jelassi

jelassi.nidhal@gmail.com

Submit

Set value

```
E {name: {first: "Nidhal", last: "Jelassi"}, email: "jelassi.nidhal@gmail.com"} E onSubmit — login.component.ts:19
```

RESET ET MODIFICATION

- ▶ Pour faire un reset de votre formulaire, ca se passe dans la classe Typescript de votre composant. Ainsi, il suffit de passer l'objet NgForm à la méthode concernée et d'appeler la méthode reset().
- ▶ Pour modifier le formulaire depuis la classe Typescript, vous avez le choix entre 2 méthodes de NgForm : setValue() et form.patchValue()

```
randomPwd(f : NgForm) {  
  f.setValue({  
    login:'',  
    password:'pwd1234566'  
  })  
}
```

```
randomPwdTwo(f : NgForm) {  
  f.form.patchValue({  
    password:'blablabla1234566'  
  })  
}
```

REACTIVE FORMS

ReactiveFormsModuleModule

- ▶ En adoptant l'approche Reactive Forms, le formulaire va être codé. C'est à dire qu'on va programmer depuis la classe Typescript comment le formulaire va se comporter dans le HTML.
- Importer le module `ReactiveFormsModuleModule`
- ▶ Le formulaire est créé en Typescript en tant qu'instance de `FormGroup` du package `@angular/forms`
- ▶ Au constructeur de `FormGroup`, on passe un objet avec différents `FormControl`, du package `@angular/forms`.
- ▶ Le constructeur de `FormControl` prend, au plus, 2 arguments : Valeur par défaut et un ensemble de validators.

EXEMPLE

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-react-form',
  templateUrl: './react-form.component.html',
  styleUrls: ['./react-form.component.css']
})
export class ReactFormComponent implements OnInit {
  genders = ['male', 'female'];
  signupForm : FormGroup;

  constructor() { }

  ngOnInit(): void {
    this.signupForm = new FormGroup({
      'username' : new FormControl(null, Validators.required),
      'email' : new FormControl(null, [Validators.required, Validators.email])
      'gender' : new FormControl('male')
    })
  }
}
```

LIAISON AVEC LA VUE

- ▶ Maintenant il faut lier le formulaire de la vue (HTML) avec celui créé dans le model (Typescript).
- ▶ On va binder la propriété `formGroup` avec l'instance créé dans le `.ts`
- ▶ Pour chaque élément du formulaire, on va ajouter la propriété `formControlName`.

```
<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="username">Username</label>
    <input
      type="text"
      id="username"
      formControlName="username"
      class="form-control">
  </div>
  <div class="form-group">
    <label for="email">email</label>
    <input
      type="text"
      id="email"
      formControlName="email"
      class="form-control">
  </div>
```

VALIDATORS

- ▶ Comme pour les NgForm, les Reactive Forms ont les propriétés valid, touched, pristine, etc...
- ▶ Pour les utiliser, on les récupérera à partir de notre formulaire via la méthode get() et en spécifiant le nom de la propriété.

```
<div class="form-group">
    <label for="username">Username</label>
    <input
        type="text"
        id="username"
        formControlName="username"
        class="form-control">
    <span *ngIf="!signupForm.get('username').valid &&
        signupForm.get('username').touched"
        class="help-block">Please enter a valid username !</span>
```

GROUP CONTROLS

- ▶ Pour regrouper des controls, à l'instar du `ngModelGroup` dans les formulaires template-driven, les reactive-forms nous donnent la possibilité de passer des `FormControl`s à un autre `FormGroup`.

```
this.signupForm = new FormGroup({
  'userData' : new FormGroup({
    'username' : new FormControl(null, Validators.required),
    'email' : new FormControl(null, [Validators.required, Validators.email])
  }),
  'gender' : new FormControl('male')
})
```

GROUP CONTROLS

- ▶ Que remarquez dans votre application à présent ? Une erreur ? Evidemment.
- ▶ L'utilisation d'un FormGroup nous oblige à faire des modifications dans le formulaire, côté vue, car les FormControl ne sont plus désormais sous notre formulaire directement.
- ▶ De même pour les validators dans la vue, faut prendre soin de spécifier pour chaque formControl, le fromGroup auquel il appartient.

```
<div formGroupName="userData">
  <div class="form-group">
    <label for="username">Username</label>
    <input
      type="text"
      id="username"
      formControlName="username"
      class="form-control">
    <span *ngIf="!signupForm.get('userData.username').valid &&
      signupForm.get('userData.username').touched"
      class="help-block">Please enter a valid username !</span>
  </div>
  <div class="form-group">
    <label for="email">email</label>
    <input
      type="text"
      id="email"
      formControlName="email"
      class="form-control">
  </div>
```

FormArray

- ▶ Un Reactive Form peut également renfermer des **FormArray**, composé de plusieurs **FormControl**.
- ▶ Ainsi, on peut ajouter au **FormArray** des éléments à partir de la vue.
- ▶ **FormArray** doit être importé depuis `@angular/forms`

```
this.signupForm = new FormGroup({
  'userData' : new FormGroup({
    'username' : new FormControl(null, Validators.required),
    'email' : new FormControl(null, [Validators.required, Validators.email])
  }),
  'gender' : new FormControl('male'),
  'skills' : new FormArray([])
})
```

FormArray

- ▶ Il est nécessaire de "caster" le FormArray pour pouvoir récupérer les FormControl qu'il contient et pouvoir y ajouter de nouveaux.

```
get formSkills() {  
  return <FormArray>this.signupForm.get('skills');  
}
```

- ▶ En ajoutant un nouveau FormControl, on utilisera la méthode push() comme pour les tableaux simples.

```
addSkill() {  
  const control = new FormControl(null, Validators.required);  
  (<FormArray>this.signupForm.get('skills')).push(control);  
}
```

- ▶ Notre vue bouclera ainsi sur le FormArray avec un binding sur FormControl.

```
<div formArrayName = "skills">  
  <h3>Your skills</h3>  
  <button type="button" class="btn btn-warning" (click)="addSkill()">Add Skill</button>  
  <div class="form-group"  
    *ngFor="let skill of formSkills.controls; let i = index">  
    <input #in type="text" class="form-control" [formControl]="formSkills.controls[i]">  
  </div>  
</div>
```

CUSTOM VALIDATORS

- ▶ Les Reactive Forms permettent de créer facilement des validateurs personnalisés.
- ▶ C'est une méthode qui prend en argument le FormControl qu'on désire tester par le validateur. Elle retourne un objet dont la valeur de la propriété est booléenne.

```
forbiddenUsernames = ['nidhal', 'yasmine'];
```

```
forbiddenNames(ctrl : FormControl) : {[s : string] : boolean} {  
  if(this.forbiddenUsernames.indexOf(ctrl.value) != -1) {  
    return {'nameIsForbidden' : true}  
  }  
  return null;  
}
```

- ▶ Que se passe-t-il si on retourne `{nameIsForbidden : false}` à la place de `null` ?

ERRORS

- ▶ Pour accéder aux errors générés par le(s) validator(s) d'un FormControl, il suffit d'accéder à l'objet errors, comme pour les template-driven form.

```
<div class="form-group">
    <label for="username">Username</label>
    <input
        type="text"
        id="username"
        formControlName="username"
        class="form-control">
    <span class="help-block" *ngIf="!signupForm.get('username').valid
        && signupForm.get('username').touched">
        <span *ngIf="signupForm.get('username').errors['nameIsForbidden']"
            class="help-block">This username is invalid !</span>
        <span *ngIf="signupForm.get('username').errors['required']"
            class="help-block">This field is required !</span>
    </span>
```

STATUS ET VALUE D'UN FORMULAIRE

- ▶ A partir de notre classe typescript, nous avons la possibilité de suivre la valeur du formulaire ainsi que son statut (*valid*, *pending*, *invalid*).
- ▶ Pour cela, à partir de notre `FormGroup`, on peut faire appeler à `valueChanges` ou/et `statusChanges`, qui retournent un observable.

```
this.signupForm.statusChanges.subscribe(  
  (value) => {  
    console.log(value);  
  }  
)
```

```
this.signupForm.valueChanges.subscribe(  
  (value) => {  
    console.log(value);  
  }  
)  
}
```

RETOUR AU PROJET

- ▶ Ajouter dans notre projet un composant contenant un formulaire permettant d'ajouter une nouvelle personne.
- ▶ N'oubliez pas rediriger l'utilisateur vers la liste des CVs, après chaque ajout.
- ▶ Ajouter un composant permettant d'éditer une personne pour changer un ou plusieurs de ses propriétés.

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

Chapitre 8 : HTTP Client



OBJECTIFS

- ▶ Comprendre le design pattern Observable et son implémentation avec RxJs
- ▶ Appréhender le Module HttpClientModule d'Angular
- ▶ Utiliser les différents services du module HttpClientModule
- ▶ Comprendre le principe d'authentification via les tokens
- ▶ Utiliser les protecteurs de routes (les guards)
- ▶ Utiliser les Interceptors afin d'intercepter les requêtes Http
- ▶ Déployer votre application en production (AWS - Ghpages)

HTTPCLIENT

- ▶ Angular est un Framework FrontEnd
 - Pas d'accès à la BD
 - Pas de possibilité de persistance des données
 - Pas de puissance permettant des traitements lourds.
- ▶ Tout ceci nous pose un grand problème...

Solution :

- ▶ Le Module HttpClient

PROGRAMMATION ASYNCHRONE : LES PROMESSES

- ▶ Les promesses (`promise`) sont des objets qui représente une complétion ou l'échec d'une opération asynchrone.
- ▶ Le fonctionnement des promesses est le suivant :
 - ▶ On crée notre fonction qui retourne une promesse
 - ▶ La promesse va toujours retourner deux résultats :
 - **resolve** en cas de succès
 - **reject** en cas d'erreur
 - ▶ Vous devrez donc gérer les deux cas afin de créer votre traitement

EXEMPLE

```
var promise2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(3);
    }, 5000);
} );

promise2.then(
    function (x) {
        console.log('resolved with value : ', x);
    }
).catch( console.log('Erreur avec la Promesse' );
)
```

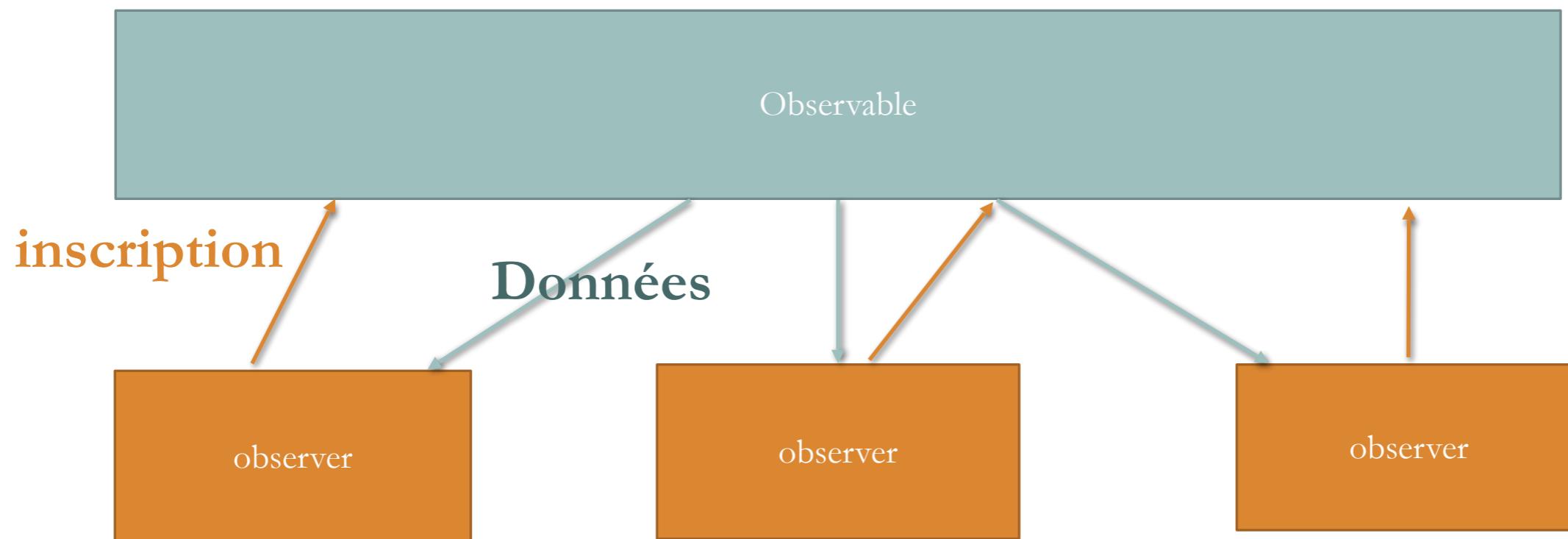
PROGRAMMATION RÉACTIVE

- ▶ Semblables aux promesses, les observables représentent une nouvelle manière d'appréhender les appels asynchrones
- ▶ Programmation avec des flux de données asynchrones

Programmation reactive =

Flux de données (*observable*) + écouteurs d'événements (*observer*).

OBSERVABLES ET OBSERVERS



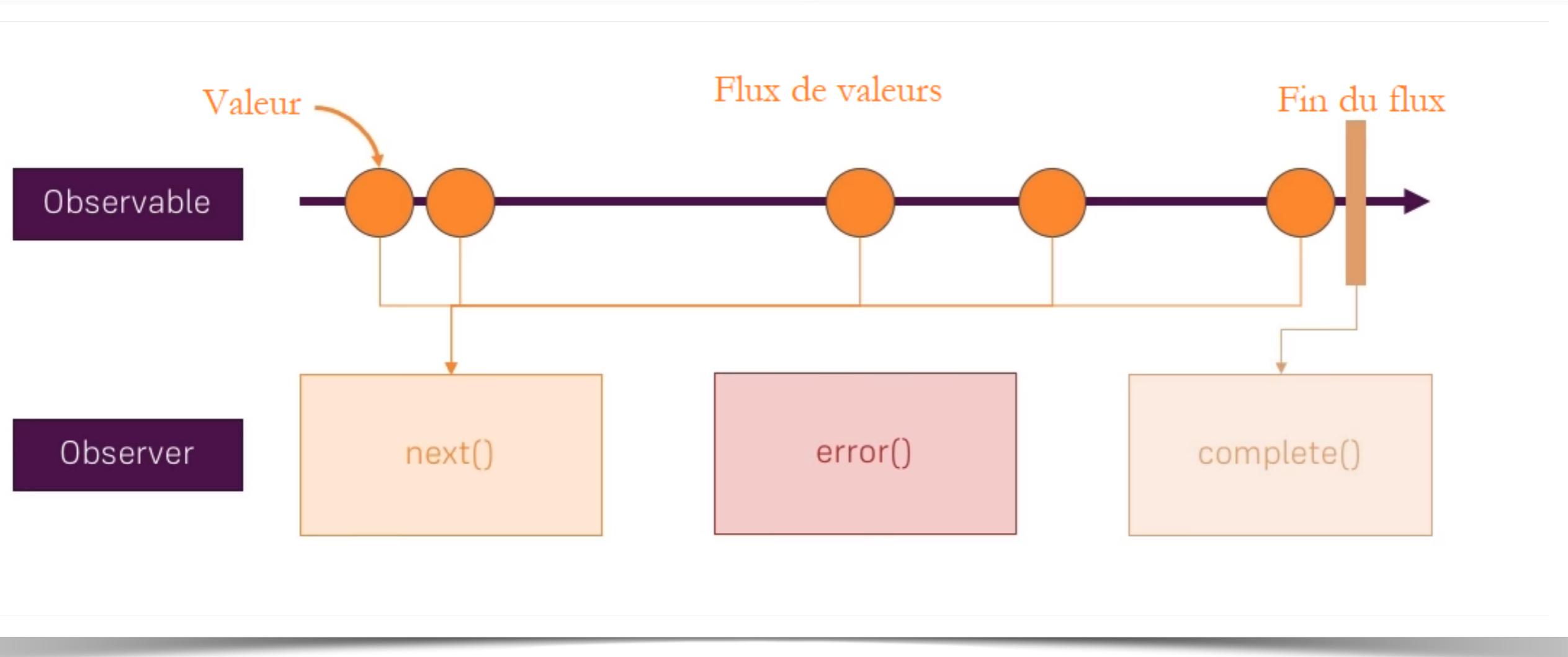
traitement

OBSERVABLES

- ▶ Le pattern design (patron de conception) Observable permet à un objet de garder la trace d'autres objets, intéressés par l'état de ce dernier.
- ▶ Il définit une relation entre objets de type un-à-plusieurs.
- ▶ Lorsque l'état de cet objet change, il notifie ces observateurs.

○ Observable

FONCTIONNEMENT

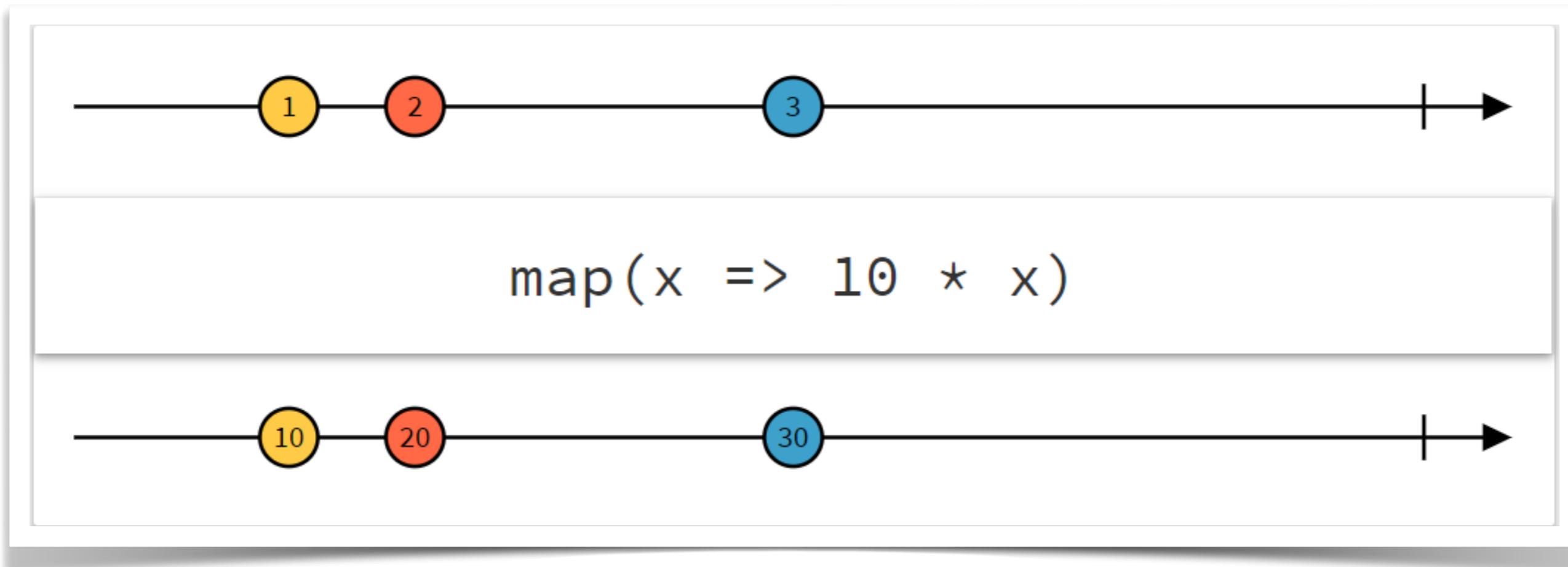


EXEMPLE OBSERVABLE

```
const observable = new Observable(  
  (observer) => {  
    let i = 5;  
    setInterval(() => {  
      if (!i) {  
        observer.complete();  
      }  
      observer.next(i--);  
    }, 1000);  
  } );  
  
observable.subscribe(  
  (val) => {  
    console.log(val);  
  }  
);
```

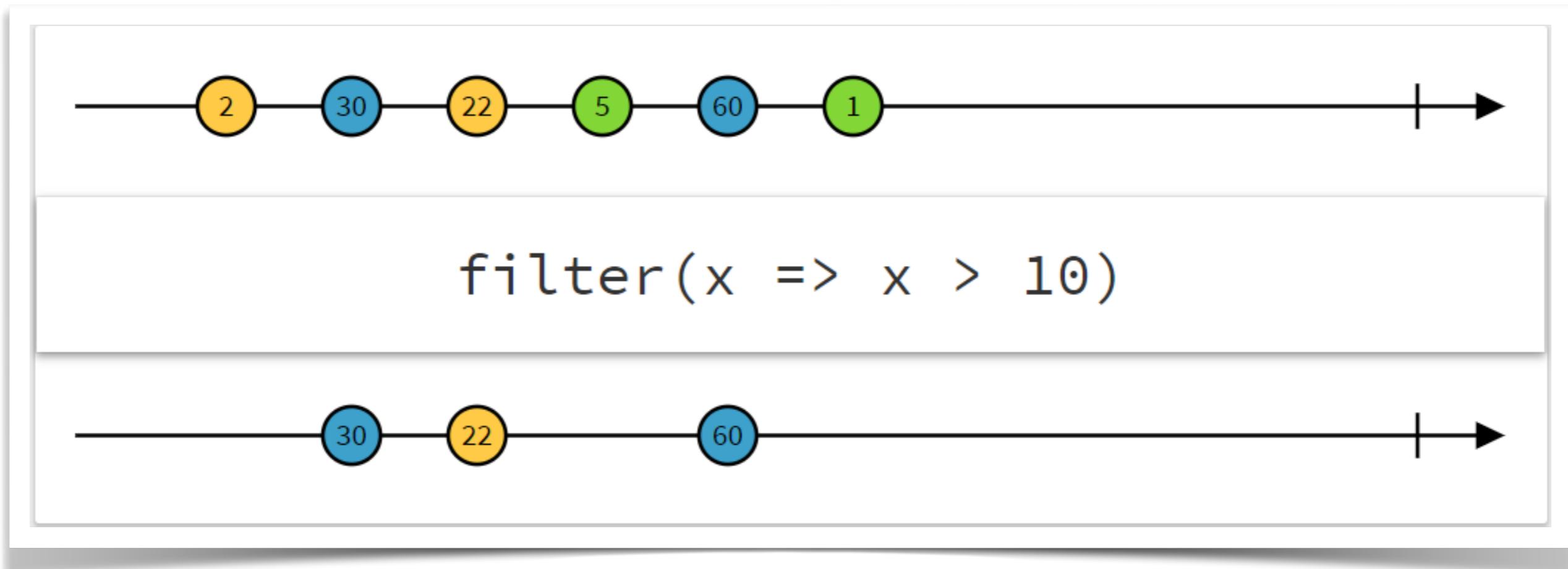
OPÉRATEURS UTILES AVEC LES OBSERVABLES

► Map



OPÉRATEURS UTILES AVEC LES OBSERVABLES

▶ Filter



OPÉRATEURS UTILES AVEC LES OBSERVABLES

▶ throttleTime



throttleTime(25)



AUTRES OPÉRATEURS UTILES

- <https://angular.io/guide/rx-library>
- <http://reactivex.io/rxjs/manual/overview.html#operators>
- <http://rxmarbles.com/>

SUBJECTS

- ▶ C'est un type d'Observable qui permet non seulement de réagir à de nouvelles informations, mais également d'en émettre.
- ▶ En gros donc, un Subject est à la fois un observable et un observer. On peut donc faire subscribe dessus, mais également lui envoyer des valeurs
- ▶ **Exemple de cas d'utilisation** : Imaginez une variable dans un service, par exemple, qui peut être modifié depuis plusieurs components et qui fera réagir tous les components qui y sont liés en même temps.

INSTALLATION DU MODULE HTTPCLIENTMODULE

- ▶ Le module permettant la consommation d'API externe s'appelle le HttpClientModule.
- ▶ Afin d'utiliser le module HTTP, il faut l'importer de @angular/common/http (@angular/http dans les anciennes versions)
- ▶ Il faudra aussi l'ajouter dans le fichier app.module.ts dans le tableau d'imports.

```
import {HttpClientModule} from "@angular/common/http";

imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule,
],
```

INJECTION DE HTTPCLIENT

- ▶ Afin d'utiliser le module `HttpClientModule`, il faut injecter le service `HttpClient` dans le composant ou le service dans lequel vous voulez l'utiliser.



```
constructor(private http:HttpClient) { }
```

INTERAGIR AVEC UNE API - GET REQUEST

- ▶ Afin d'exécuter une requête **get**, le service HttpClient nous offre une méthode **get**.
- ▶ Cette méthode retourne un Observable.
- ▶ Cette observable a 3 callback functions comme paramètres.
 - ▶ Une en cas de réponse
 - ▶ Une en cas d'erreur
 - ▶ La troisième en cas de fin du flux de réponse.

INTERAGIR AVEC UNE API - GET REQUEST

```
this.http.get(API_URL).subscribe(  
  (response:Response)=>{  
    //ToDo with DATA  
  },  
  (err:Error)=>{  
    //ToDo with error  
  },  
  () => {  
    console.log('Data transmission complete');  
  }  
) ;
```

POST REQUEST

- ▶ Afin d'exécuter une requête **POST** le service `HttpClient` nous offre une méthode `post`.
- ▶ Cette méthode retourne un `Observable`. Elle diffère de la méthode `get` avec un attribut supplémentaire : ce qu'on va poster !
- ▶ Cette observable a 3 **callback functions** comme paramètres.
 - ▶ Une en cas de réponse
 - ▶ Une en cas d'erreur
 - ▶ La troisième en cas de fin du flux de réponse.

POST REQUEST

```
this.http.post(API_URL,dataToSend).subscribe(  
  (response:Response)=>{  
    //ToDo with response  
  },  
  (err:Error)=>{  
    //ToDo with error  
  },  
  () => {  
    console.log('complete');  
  }  
);
```

PLUS D'INFOS

- ▶ Pour plus d'informations concernant ce module, vous pouvez vous référer à la documentation officielle d'Angular :

<https://angular.io/guide/http>

APPLICATION



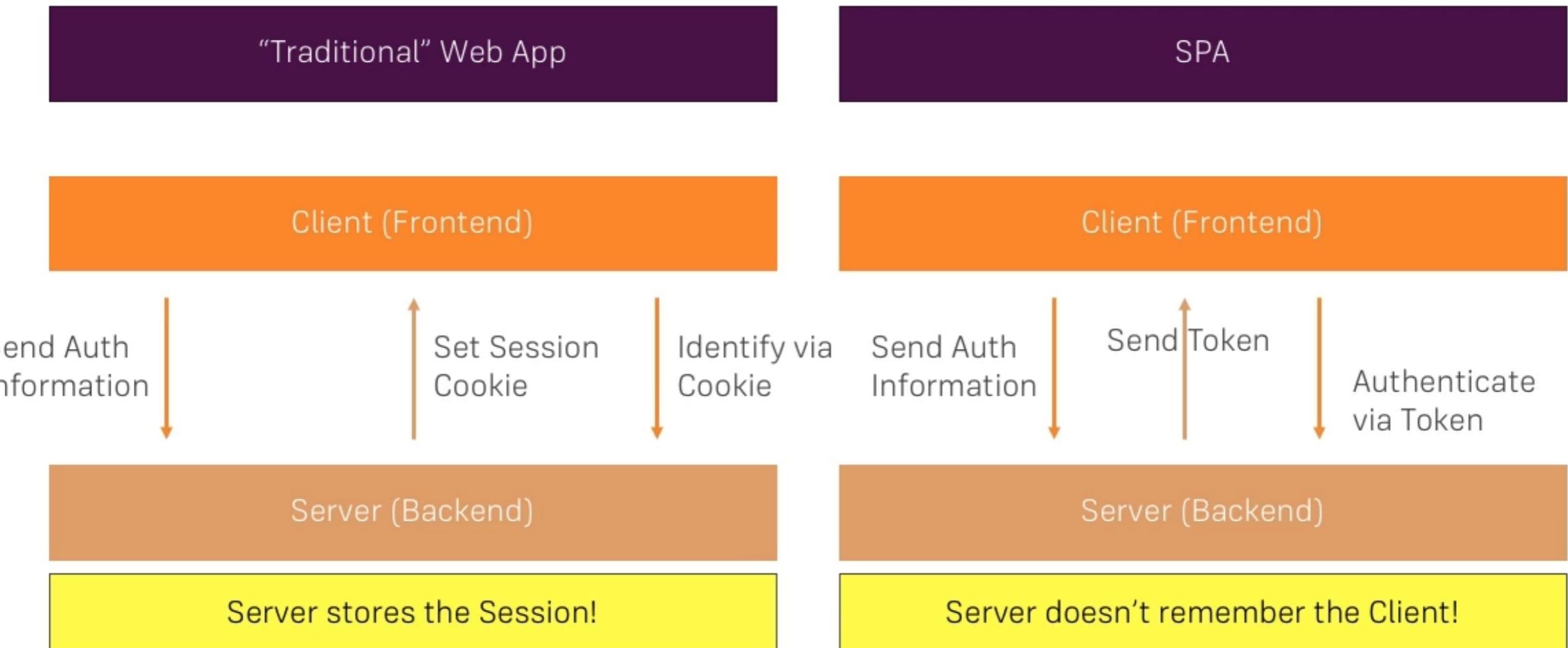
- ▶ Accéder au site <https://jsonplaceholder.typicode.com/>
- ▶ Utiliser l'API des photos pour afficher la liste des posts.
- ▶ En attendant le chargement des données afficher un message « loading... ».

HTTPPARAMS / HTTPHEADERS

- ▶ Afin d'ajouter des paramètres à vos requêtes, HttpClient et vous offre la classe `HttpParams` et `HttpHeaders`, en fonction de la manière avec laquelle vous souhaitez envoyer des informations avec votre requête.
- ▶ Ces classes sont des classes **immutables** (read Only).
- ▶ Elle propose une panoplie de méthode helpers permettant de la manipuler.
 - ▶ `set(clé,valeur)` permet d'ajouter des headers. Elle écrase les anciennes valeurs.
 - ▶ `append(clé,valeur)` concatène de nouveaux headers.
- ▶ Toutes les méthodes de modification retourne un objet de même type, permettant un chainage d'appel.

AUTHENTIFICATION

Comment ceci va se passer ?



AJOUTER LE TOKEN DANS LA REQUÊTE (V1)

- ▶ Si la ressource demandé est contrôlé avec un token, vous devez y insérer le token afin d'être authentifié au niveau du serveur.
- ▶ Pour ajouter un token vous pouvez le faire via un objet `HttpParams`. Cet objet possède une méthode `set` à laquelle on passe le nom du token '`access_token`' suivi du token.
- ▶ Vous devez ensuite l'ajouter comme paramètre à votre requête.

```
const params = new HttpParams()
  .set('access_token', localStorage.getItem('token'));
return this.http.post(this.apiUrl, personne, {params});
```

AJOUTER LE TOKEN DANS LA REQUÊTE (V2)

- ▶ Une seconde méthode consiste à ajouter dans le header de la requête avec comme name 'Authorization' et comme valeur 'bearer' à laquelle on concatène le Token.
- ▶ Pour se faire, créer un objet de type HttpHeaders.
- ▶ Utiliser sa méthode append afin d'y ajouter ses paramètres.
- ▶ Ajouter la à la requête.

```
const headers = new HttpHeaders();
headers.append('Authorization', 'Bearer ${token}');
return this.http.post(this.apiUrl, personne,
{headers});
```

GUARDS

- ▶ Dans vos applications, certaines routes ne doivent être accessibles que si vous êtes authentifié ou selon le rôle de l'utilisateur.
- ▶ Ce cas d'utilisation se répète souvent et s'est un sous cas de la sécurisation de vos routes.
- ▶ Angular a pris en considération ce cas en fournissant un mécanisme via l'utilisation des Guards.



GUARDS

- ▶ Ce sont des fonctions (depuis v17) qui permettent de gérer l'accès à vos routes.
- ▶ Un guard informe sur la validité ou non de la continuation du process de navigation en retournant un booléen, une promesse d'un booléen ou un observable d'un booléen.
- ▶ Le routeur supporte plusieurs types de guards, par exemple :
 - ▶ CanActivate
 - ▶ CanActivateChild
 - ▶ CanDeactivate
 - ▶ CanMatch (depuis La version 14)

GUARDS

- ▶ `canActivate` : lorsqu'un tel guard est appliqué à une route, il peut empêcher l'accès à la route.
- ▶ `canActivateChild` : peut empêcher les activations des enfants de la route sur lequel il est appliqué.
- ▶ `canDeactivate` : est utilisé pour empêcher de quitter la route actuelle.
- ▶ `canMatch` : indique au routeur si la route peut être utilisée ou non. Un usage typique de ce type de guard est d'avoir plusieurs routes pour le même chemin avec des composants différents, et d'utiliser `canMatch` pour savoir quel composant afficher en fonction du profil de l'utilisateur.

CANACTIVATE

- ▶ Afin d'utiliser le guard `canActivate` (de même pour les autres), vous devez :
 1. Créer une fonction de type `CanActivateFn` qui prend deux paramètres, **route** (`ActivatedRouteSnapshot`) et **state** (`RouterStateSnapshot`) et qui retourne un **booléen** permettant ainsi l'accès ou non à la route cible.
 2. Finalement pour l'appliquer à une route, ajouter la dans la propriété `canActivate`. Cette propriété prend un tableau de guard. Elle ne laissera l'accès à la route que si la totalité des guard retourne true.

ETAPES

```
export const allowGuard: CanActivateFn = (route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean => {
  const router = inject(Router);
  let token = localStorage.getItem('token');
  if (token) return true;
  else {
    router.navigateByUrl('/login');
    return false;
  }
};
```

Ceci est l'un des cas où la fonction **inject()** doit être utilisé : comme l'intercepteur est une fonction, il n'y a pas de constructeur qui permette d'injecter les dépendances, mais la fonction **inject()** permet néanmoins de les obtenir.

```
const myRoutes : Routes = [
  {path : '', component : AccueilComponent},
  {path : 'cv', children : [
    {path : '', component : CvComponent},
    {path : 'add', component : AddComponent, canActivate : [AllowGuard]},
    {path : ':id', component : InfosComponent},
    {path : ':id/edit', component : EditComponent, canActivate : [AllowGuard]},
  ]},
  {path : 'servers', component : ManageServersComponent},
```

RETOUR AU PROJET...

- ▶ Ajouter les guards nécessaire afin de sécuriser vos routes.
- ▶ Faites en sorte qu'une personne déjà connectée ne peut pas accéder au composant de login.

LES INTERCEPTEURS

- ▶ A chaque fois que nous avons une requête à laquelle nous devons ajouter le token, nous devons refaire toujours le même travail.
- ▶ Pourquoi ne pas intercepter les requêtes HTTP et leur associer le token s'il est là à chaque fois ?
- ▶ Un interceptor Angular (fournit par le client HTTP) va nous permettre d'intercepter une requête à l'entrée et à la sortie de l'application.
- ▶ Un intercepteur est une classe qui implémente l'interface `HttpInterceptor`.
- ▶ En implémentant cette interface, chaque intercepteur va devoir implémenter la méthode `intercept`.

LES INTERCEPTEURS

- ▶ A chaque fois que nous avons une requête à laquelle nous devons ajouter le token, nous devons refaire toujours le même travail.
- ▶ *Pourquoi ne pas intercepter les requêtes HTTP et leur associer le token s'il est là à chaque fois ?*
- ▶ Un interceptor Angular (fournit par le client HTTP) va nous permettre d'intercepter une requête à l'entrée et à la sortie de l'application.
- ▶ Un intercepteur est une fonction (depuis v17) de type `HttpInterceptorFn`, qui prend deux paramètres : un objet `HttpRequest` et un objet `HttpHandlerFn`.
- ▶ Elle retourne un Observable <`HttpEvent<unknown>`>

MODIFIER LA REQUÊTE

- ▶ Par défaut la requête est immutable, on ne peut pas la changer.
- **Solution** : la cloner, changer les headers du clone et le renvoyer. Dans cette exemple, on désire exempter les requêtes GET de tout ajout de Headers.

```
export const addTokenInterceptor: HttpInterceptorFn = (req, next) => {
  if(req.method == "GET")
    return next(req);

  else {
    let token = localStorage.getItem('myToken');
    if(token){
      let cloneReq = req.clone({
        headers : new HttpHeaders().set('Authorization', `bearer ${token}`)
      })
      return next(cloneReq);
    }
    return next(req);
  }
};
```

V2

```
let cloneReq = req.clone({
  setHeaders: { Authorization: `bearer ${token}` },
});
```

GÉRER LES ERREURS AVEC LES INTERCEPTEURS

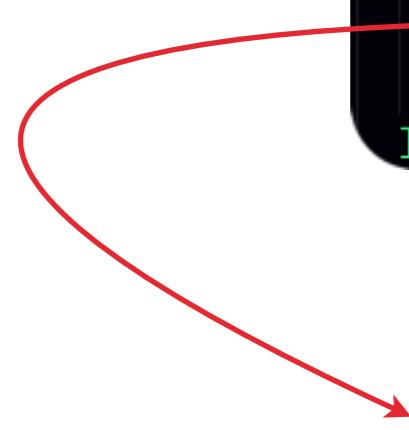
- ▶ Avec les intercepteurs, il est également possible d'intercepter la réponse, ce qui peut être pratique pour gérer les erreurs.
- ▶ On peut ainsi utiliser le service ErrorHandler de @angular/core.

```
export const errorHandlerInterceptor: HttpInterceptorFn = (
  req: HttpRequest<unknown>,
  next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  const errorHandler = inject(ErrorHandler);
  const router = inject(Router);

  return next(req).pipe(
    tap({
      error: (errorResponse: HttpErrorResponse) => {
        if (errorResponse.status === HttpStatusCode.Unauthorized) {
          router.navigateByUrl('/');
        } else {
          errorHandler.handleError(errorResponse);
        }
      },
    })
  );
};
```

PROVIDER UN INTERCEPTEUR

- ▶ Il faut ensuite configurer le provider du client HTTP afin qu'il utilise l'intercepteur
- ▶ A partir de la version 17, Angular nous propose une nouvelle manière de provider les intercepteurs.
- ▶ Ca se passe au niveau des modules (ou des standalones components) comme le montre la capture ci-dessous :



```
providers: [
  provideHttpClient(
    withInterceptors([addTokenInterceptor, errorHandlerInterceptor])
  ),
]
```

Importés depuis `@angular/common/http`

RETOUR AU PROJET

- ▶ Créer un intercepteur qui injecte un token à chaque requête.
- ▶ Si le token n'existe pas, l'utilisateur ne peut pas exécuter des opérations d'ajout, de suppression et de modification des candidats, déjà développés.
- ▶ Créer un second intercepteur qui traitera les erreurs que pourraient renvoyer l'API qu'on va consommer.

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

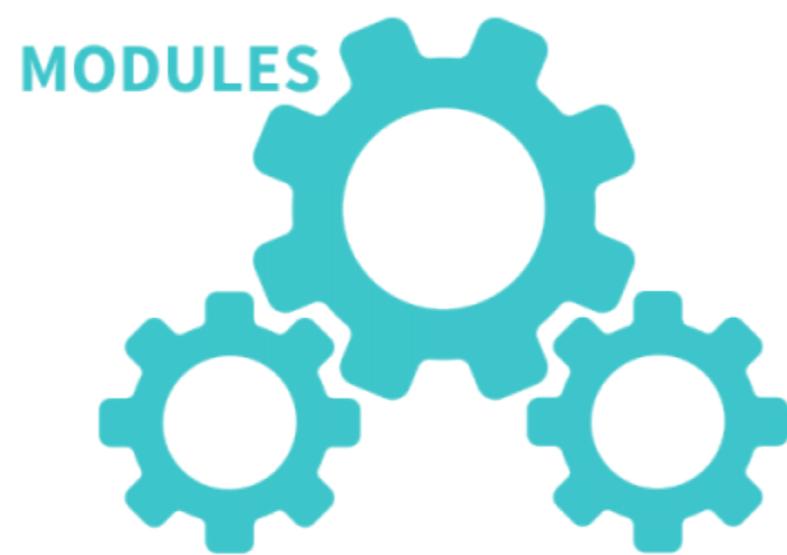
ANGULAR

Chapitre 9 : Les Modules

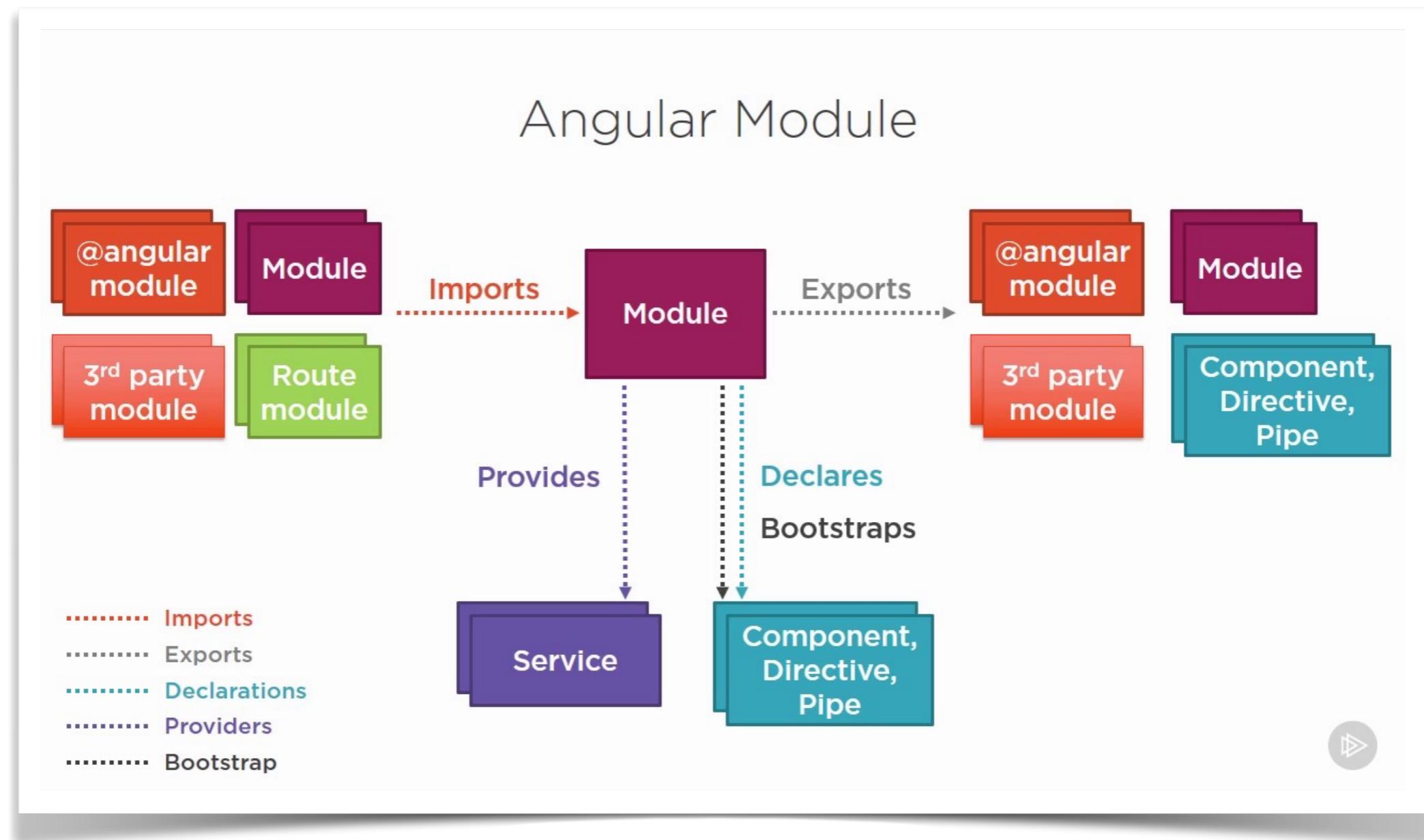


QU'EST CE QU'UN MODULE ?

- ▶ Un module Angular est un mécanisme pour regrouper des composants, des directives, des pipes et des services qui peuvent être combinés avec d'autres modules pour créer une application.
- ▶ Une application Angular peut être considérée comme un puzzle où chaque pièce (chaque module) est nécessaire pour pouvoir voir l'image complète.



QU'EST CE QU'UN MODULE ?



QU'EST CE QU'UN MODULE ?

- ▶ Un module, décoré avec `@NgModule` peut exporter ou non des composants, des directives, des pipes et des services.
- ▶ Les éléments exportés sont destinés à être utilisés par d'autres modules.
- ▶ Cependant, ceux qui ne sont pas exportés (qui sont masqués donc) sont juste utilisés à l'intérieur du module lui-même et ne sont pas directement accessibles par d'autres modules de notre application.
- ▶ Il y a 2 types de modules : Les modules racines (root) et les modules de fonctionnalité (feature). On y reviendra.

CLI

- ▶ Pour créer un nouveau sous-module via le CLI, on execute la commande suivante :
 - `ng generate module nom_du_module`
- ▶ Ou plus simplement
 - `ng g m nom_du_module`
- ▶ Pour créer un sous-module stock (ce dernier ne sera pas enregistré dans votre module racine `app.module.ts`)
 - `ng g m stock`

CLI

- ▶ Pour créer un sous-module stock et l'enregistrer dans app.module.ts, on modifiera la commande du diapo précédent comme ceci :
 - `ng g m stock --module=app`
- ▶ Pour créer un sous-module stock, l'enregistrer dans app.module.ts et créer son propre module de routage (l'ordre des options n'a pas d'importance) :
 - `ng g m stock --module=app --routing`

```
nidhal@N2SY test % ng g module stock --module=app --routing
CREATE src/app/stock/stock-routing.module.ts (248 bytes)
CREATE src/app/stock/stock.module.ts (276 bytes)
UPDATE src/app/app.module.ts (1103 bytes)
```

IMPORT

- `ng g m modules/stock --module=app –routing`

- ▶ Pour bien structurer l'application, il est recommandé de regrouper les modules dans un dossier seul dossier :
- ▶ Si l'option `--module=app` n'a pas été précisée, il faut déclarer `stock.module.ts` dans

```
import { StockModule } from './stock/stock.module';

@NgModule({
  declarations: [
    AppComponent,
    LoginComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    StockModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

MODULE ROOT

- ▶ Nous avons déjà vu que dans un module, il y a un composant racine (root) et éventuellement d'autres composants (secondaires).
- ▶ Dans une application, nous avons un seul module racine (root) et éventuellement d'autres modules. Ces derniers sont appelés module de fonctionnalité (ou feature module).
- ▶ Pour démarrer une application, Angular a besoin de connaître lequel des modules et le module racine. Pour l'identifier, c'est très simple. Le module racine est celui qui importe `BrowserModule` alors que les autres importent `CommonModule`.

CommonModule – BrowserModule

- ▶ En inspectant notre module racine (`app.module.ts`) et le module qu'on vient de créer (`stock.module.ts`), on peut se poser deux questions :
- ▶ *Pourquoi le module stock importe CommonModule ?*
- Parce que c'est le module contenant les pipes et les directives Angular
- ▶ *Pour alors CommonModule n'est pas importé dans le module App ?*
- Parce que dans `app.module.ts`, le module App importe `BrowserModule` et ce dernier importe `CommonModule`

ROUTING

- ▶ Puisque chaque module peut/doit avoir son propre système de routing, il faut faire attention à deux détails :
- ▶ Dans le module App, on utilise la propriété `loadChildren` pour associer un module tierce à un `path`. `LoadChildren` peut prendre comme valeur soit une fonction fléchée, soit une chaîne de caractères, comme sur la capture du diapo suivant.
- ▶ Dans un module qui n'est pas le module racine, on crée notre module de routing en faisant appel au `RouterModule` via la méthode `forChild` et non `forRoot`.

EXEMPLE

```
const myRoutes : Routes = [
  {path : 'add', component: AddComponent},
  {path : 'receive', component: ReceiveComponent },
  {path : 'to-stock-v2', loadChildren : () => import('./auth/auth.module').then( m => m.AuthModule)},
  {path : 'to-stock', loadChildren : './auth/auth.module#AuthModule'}
]

export const APP_ROUTING = RouterModule.forRoot(myRoutes);
```

app.routing.ts

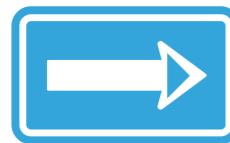
```
const stockRoutes : Routes = [
  { path : 'connect', component: ConnecterComponent},
  { path : '', component: HomeComponent}
]

export const STOCK_ROUTING = RouterModule.forChild(stockRoutes);
```

stock.routing.ts

LAZY LOADING

- ▶ En configurant l'intégralité du Routing de l'application dans le module `AppRoutingModule`, on serait amené à importer tous les composants / modules de l'application dès son démarrage.
- ▶ Ainsi, plus l'application sera riche, plus la page d'accueil sera lente à charger par le navigateur.
- ▶ Pour éviter ces problèmes de scalabilité, Angular permet de charger les modules à la demande afin de ne pas gêner le chargement initial de l'application.



C'est le principe du **lazy loading**.

Par NIDHAL JELASSI
jelassi.nidhal@gmail.com

ANGULAR

Chapitre 10 : Déploiement et tests
unitaires



OBJECTIFS

- ▶ Déploiement d'une application Angular sur :
 - GitHub pages
 - AWS S3
- ▶ Découverte des tests unitaires avec Karma et Jasmine.
 - Tester des composants et services
 - Tester des appels asynchrones
 - Tester des pipes.

TESTS UNITAIRES

INTRODUCTION

- ▶ Angular fournit en standard tous les outils permettant d'implémenter les tests unitaires.
- ▶ Ces tests vont assurer un code de qualité, sans traitement inutile ou complexe.
- ▶ Ils permettent de documenter nos fonctionnalités, et nous orienter vers une simplicité de programmation qui permet d'assurer la maintenabilité de nos applications.
- ▶ Un test unitaire couvre une petite fonctionnalité et ne s'occupe pas de la manière dont les différents unités testées interagissent entre elles. Il est rapide, fiable et pointe directement sur le bug en question.

JASMINE ET KARMA

- ▶ En installant Angular via le Cli, vous trouverez prêt à l'emploi deux outils de tests : Jasmine et Karma.



- ▶ Karma va être notre moteur de tests. Notre projet, créé avec l'Angular-Cli, comporte un fichier de configuration standard, karma.conf.js. Les tests sont par défaut lancés dans Chrome, on peut cependant choisir d'autres navigateurs.
- ▶ Plusieurs plugins sont également ajoutés.

PACKAGE.JSON

- ▶ En créant un nouveau projet avec Angular CLI, vous pouvez trouver dans package.json que nous disposons déjà de deux outils de tests que sont Jasmin et Karma.

```
"@types/jasmine": "~2.8.8",
"@types/jasminewd2": "~2.0.3",
"codelyzer": "~4.5.0",
"jasmine-core": "~2.99.1",
"jasmine-spec-reporter": "~4.2.1",
"karma": "~3.1.1",
"karma-chrome-launcher": "~2.2.0",
"karma-coverage-istanbul-reporter": "~2.0.1",
"karma-jasmine": "~1.1.2",
"karma-jasmine-html-reporter": "^0.2.2",
```

- ▶ Pour ajouter le fichier de configuration du Karma, il suffit d'exécuter la commande :

```
ng generate config karma
```

CONFIGURATION (KARMA.CONFIG.JS)

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular-devkit/build-angular'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
      require('@angular-devkit/build-angular/plugins/karma')
    ],
    client: {
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    coverageIstanbulReporter: {
      dir: require('path').join(__dirname, '../coverage/DMWMM-A'),
      reports: ['html', 'lcovonly', 'text-summary'],
      fixWebpackSourcePaths: true
    },
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    LogLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false
  });
};
```

The diagram illustrates the structure of a Karma configuration file (karma.config.js) with annotations explaining various configuration options:

- An arrow points from the line `frameworks: ['jasmine', '@angular-devkit/build-angular'],` to the text "Framework à lancer".
- An arrow points from the line `plugins: [` to the text "Plugins à charger".
- An arrow points from the line `dir: require('path').join(__dirname, '../coverage/DMWMM-A'),` to the text "Format du rapport".
- An arrow points from the line `autoWatch: true,` to the text "Lancer le test en mode watch".
- An arrow points from the line `browsers: ['Chrome'],` to the text "Navigateur".

JASMINE

- ▶ Jasmine sera notre langage d'assertion. Il permet de décrire les objets à tester, d'exécuter des commandes avant et après chaque test, et d'évaluer le bon fonctionnement de notre code.
- ▶ Jasmine est un framework permettant de faciliter la création de tests. Il contient un ensemble de fonctionnalités permettant d'écrire plusieurs types de test.



Jasmine

NG TEST

- ▶ Afin de lancer un test, vous avez juste besoin d'une seule commande et Karma fait le reste. Il exécutera les tests, ouvrira le navigateur, et affichera un rapport sur l'ensemble des tests.
 - ▶ Angular-Cli fournit la ligne de commande de lancement de nos tests :

ng test

Karma v3.1.4 - connected

Chrome 81.0.4044 (Mac OS X 10.14.6) is idle

Jasmine 2.99.0

• • • • • • • • •

12 specs, 0 failures

```
AppComponent
  AppComponent
    should be 0 when negative

    should create the app
    should render title in a h1 tag

ReversedPipe
  create an instance

UserService
  should be created

UserComponent
  should create the app
  should user user name from the service
  should display the user name if user is Logged in
  should not display the user name if user is not Logged in
  should not fetch data successfully if not called asynchronously
  should fetch data successfully if called asynchronously
  should fetch data successfully if called asynchronously
```

JASMINE : CONCEPTS DE BASE

- ▶ Les fichiers dans lesquels nous allons écrire nos tests sont au format : `nomfichier.spec.ts`
- ▶ Par convention, nous créons nos fichiers **`spec.ts`** dans le même répertoire que l'objet que nous souhaitons tester. Karma ira analyser tous les fichiers `spec.ts` qu'il trouvera dans notre projet.
- ▶ Jasmine a les principales fonctions et méthodes suivantes :
- ▶ **describe** : description de l'objet à tester
- ▶ **beforeEach / afterEach /beforeAll / afterEach** : bout de code exécuté avant / après chaque test
- ▶ **it** : bloc de description de la fonctionnalité à tester
- ▶ **expect** : évaluation du cas à tester

JASMINE : CONCEPTS DE BASE

- ▶ **expect** comporte des **matchers** de comparaison avec la variable ou la méthode que l'on souhaite évaluer. Les principaux sont :
- ▶ `toBe` : égalité stricte (équivalent au `==` javascript)
- ▶ `toEqual` : égalité non stricte (comparaison entre objets)
- ▶ `toContain` : un Array contient un élément donné, ou une string contient une chaîne donnée
- ▶ `toBeDefined` : l'objet doit être défini
- ▶ `toBeNull` : la valeur doit être nulle
- ▶ `toBeTruthy` / `toBeFalsy` : la valeur est vraie / fausse (truthy / falsy)
- ▶ `toHaveBeenCalled` : une méthode doit avoir été appelée

MOCKS

- ▶ Jasmine nous offre également la possibilité de mocker des objets ou des méthodes avec :
 - ▶ `spyOn` : mock de la méthode d'un objet
 - ▶ `createSpyObj` : mock d'un objet dans son intégralité
- ▶ Ces mocks peuvent retourner une valeur que nous définissons afin de satisfaire les besoins de nos tests.

NOTRE PREMIER TEST

- ▶ Commençons par présenter une classe, prenons un service très simple. Il aura pour rôle d'authentifier un user en vérifiant qu'il est une variable stocké dans le localStorage.

```
@Injectable({
  providedIn: 'root'
})
export class AuthService {
  logged = false;
  isLoggedIn() {
    let myToken = localStorage.getItem('token');
    return (!!myToken);
  }
}
```

NOTRE PREMIER TEST

- ▶ La seul fonctionnalité de cette classe est de vérifier s'il existe une variable **user** dans le localStorage.
- ▶ Si oui elle doit retourner true, Sinon elle retourne false
- ▶ Tester cette classe revient donc à vérifier ces deux fonctionnalités.

```
describe('AuthService', () => {
  it('should be created', () => {
    const service = new AuthService();
    expect(service).toBeTruthy();
  });

  it('it should return true if there is a user in the localStorage ', () => {
    const service = new AuthService();
    localStorage.setItem('user', 'nidhal');
    expect(service.isLoggedIn()).toBeTruthy();
  });
});
```

NOTRE PREMIER TEST - V.2

```
let service;
describe('AuthService', () => {
  beforeEach(() => {service = new AuthService();
    });
  afterEach(() => {
    localStorage.removeItem('user');
    });
  it('should be created', () => {
    expect(service).toBeTruthy();
    });
  it('it should return true if there is a user in the localStorage ', () => {
    localStorage.setItem('user', 'nidhal');
    expect(service.isAuthenticated()).toBeTruthy();
    });
  it('it should return false if there is not a user in the localStorage ', () => {
    expect(service.isAuthenticated()).toBeFalsy();
    });
})
```

TESTBED

- ▶ La librairie standard de tests Angular se nomme `@angular/core/testing`. Elle comprend des objets et fonctions, dont **TestBed** et **async**. Ces éléments vont nous permettre de définir un module de test pour nos directives et services Angular.
- ▶ **TestBed** est l'outil le plus important des utilitaires de test Angular.
- ▶ Il permet de construire dynamiquement un module de teste afin de simuler un Module Angular.
- ▶ En utilisant la méthode **configureTestingModule** de **TestBed** vous avez accès à une grande partie des propriété de `@NgModule`.
- ▶ Prenant l'exemple de test d'un service, il vous faudra l'ajouter dans le provider avant de le tester.

TESTBED

- ▶ La méthode `createComponent` de **TestBed** prend en paramètre un composant et en crée un objet de type `ComponentFixture`.
- ▶ Exemple : `fixture = TestBed.createComponent(AppComponent);`
- ▶ Cette classe permet d'encapsuler un composant et son template et offre une multitudes d'attributs et de méthodes très utiles pour tester un composant.

```
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));
})
```

DEBUGELEMENT / NATEVEELEMENT

- ▶ `debugElement` est une classe Angular qui contient un ensemble de références et de méthodes permettant de gérer les éléments du composant.
- ▶ `nativeElement` est une propriété de `debugElement` qui permet de récupéré une référence sur l'élément DOM du template.
- ▶ A partir des fixtures on peut utiliser l'attribut `componentInstance` afin de récupérer une instance de notre composant.
- ▶ Nous pouvons aussi accéder à `debugElement` et accéder à la même propriété.

EXEMPLE

- ▶ Ci-dessous, deux tests qui ont pour objectifs de vérifier que le composant AppComponent a été démarré et que le contenu de la première balise H1 est égal à "My App"

```
it('should create the app', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  expect(app).toBeTruthy();
});

it('should render title in a h1 tag', () => {
  let fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  let compiled = fixture.debugElement.nativeElement;
  expect(compiled.querySelector('h1').textContent).toContain('My App');
})
```

COMPOSANT - SERVICE

- ▶ Un composant est une combinaison entre un Template (votre code HTML) et le métier qui le gère (cotre classe TypeScript).
- ▶ En testant votre composant, on doit tester l'adéquation entre le Template et sa classe TypeScript et la bonne communication entre les deux.
- ▶ Pour cela, on doit avoir accès à l'instance de la classe TS et l'élément qui héberge votre composant dans le DOM. TestBed facilite l'accès à ces deux éléments en offrant des méthodes pour le faire.
- ▶ Afin de récupérer un service injecté, TestBed offre la méthode get qui prend en paramètre la classe du service injecté et retourne une instance.

TESTER INJECTION DE DÉPENDANCES

```
it('should user user name from the service', () => {
  let fixture = TestBed.createComponent(UserComponent);
  let app= fixture.debugElement.componentInstance;
  let userService = TestBed.get(UserService);
  fixture.detectChanges();
  expect(userService.user.name).toEqual(app.user.name);
});
```

```
it('should not display the user name if user is not Logged in', () => {
  let fixture = TestBed.createComponent(UserComponent);
  let app= fixture.debugElement.componentInstance;
  fixture.detectChanges();
  let compiled = fixture.debugElement.nativeElement;
  expect(compiled.querySelector('p').textContent).not.toContain(app.user.name);
});
```

TESTS ASYNCHRONE AVEC `async`

- ▶ Async est un outil offert par Angular afin de gérer les tests asynchrones.
- ▶ Il suffit d'englober votre test avec `async` et d'ajouter une méthode `fixture.whenStable` qui retourne une promesse.
- ▶ Ensuite, c'est dans la méthode `then` de cette promesse, qu'on insère le code asynchrone à tester.

```
it('should fetch data successfully if called asynchronously', async(() => {
  let fixture = TestBed.createComponent(UserComponent);
  let app= fixture.debugElement.componentInstance;
  let dataService = fixture.debugElement.injector.get(DataService);
  let spy = spyOn(dataService, 'getDetails').and.returnValue(Promise.resolve('Data'));
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    expect(app.data).toBe('Data');
  })
}));
```

TESTS ASYNCHRONE AVEC fakeAsync

- ▶ fakeAsync permet de définir une zone asynchrone (à tester), et la méthode tick(n) va faire écouler le temps du timeout de manière synchrone au bout de n milliseconde.
- ▶ C'est donc comme si on simulait le traitement asynchrone de la méthode à tester. Très utilisé pour tester les appels Http.

```
it('should fetch data successfully if called asynchronously', fakeAsync(() => {
  let fixture = TestBed.createComponent(UserComponent);
  let app= fixture.debugElement.componentInstance;
  let dataService = fixture.debugElement.injector.get(DataService);
  let spy = spyOn(dataService, 'getDetails').and.returnValue(Promise.resolve('Data'));
  fixture.detectChanges();
  tick();
  expect(app.data).toBe('Data');
}););
```

APPLICATION : TEST D'UN PIPE

- ▶ Un pipe contient uniquement la méthode transform. Tester un pipe revient donc à tester la méthode transform.
- ▶ Soit un pipe qui, appliqué à une chaîne de caractères, renvoie son inverse. Exemple, pour "angular", le pipe renverra "ralugna".
- ▶ Donner le test unitaire qui permet de tester si ce pipe fonctionne correctement.

AWS S3

HTTP SERVER

- ▶ Avant de passer au déploiement de votre application, vous pouvez voir ce qu'elle donne en mode production grâce au package http-server.
- ▶ Pour l'installer : \$(sudo) nom Install http-server -g
- ▶ On génère le build de notre application avec : ng build --prod
- ▶ Il ne reste maintenant qu'à lancer la version prod de votre application :
- ▶ http-server dist/notreProjet

S3

- ▶ Pour déployer votre application sur S3, il vous faut un compte AWS.
- ▶ Pour toute nouvelle inscription, AWS vous offre 12 mois de gratuité, l'occasion donc de tester ce que nous allons faire.
- ▶ La première étape à faire est de se connecter à votre compte S3 et de créer un nouveau compartiment (Bucket).

Amazon S3

Compartiments (1)

Copier l'ARN Vider Supprimer **Créer un compartiment**

Les compartiments constituent le conteneur fondamental d'Amazon S3 pour le stockage de données. Pour que d'autres personnes puissent accéder aux objets de vos compartiments, vous devrez leur accorder explicitement des autorisations. [En savoir plus](#)

Rechercher le compartiment par nom

Nom	Région	Accéder	Compartiment créé
cv-test	USA Est (Virginie du Nord) us-east-1	Les objets peuvent être publics	2020-05-01T21:54:27.000Z

PROPRIÉTÉS

- Après avoir spécifié le nom du compartiment et de la région adéquate, décochez la case "Bloquer tout l'accès public" (Block all public access) dans la section des permissions.
- En cliquant sur le nom du nouveau compartiment créé, vous devez avoir cette interface.

Amazon S3 > cv-test

cv-test

Présentation Propriétés Autorisations Gestion Points d'accès

Saisir un préfixe et appuyer sur Entrée pour lancer la recherche. Pour annuler, appuyer sur ESC.

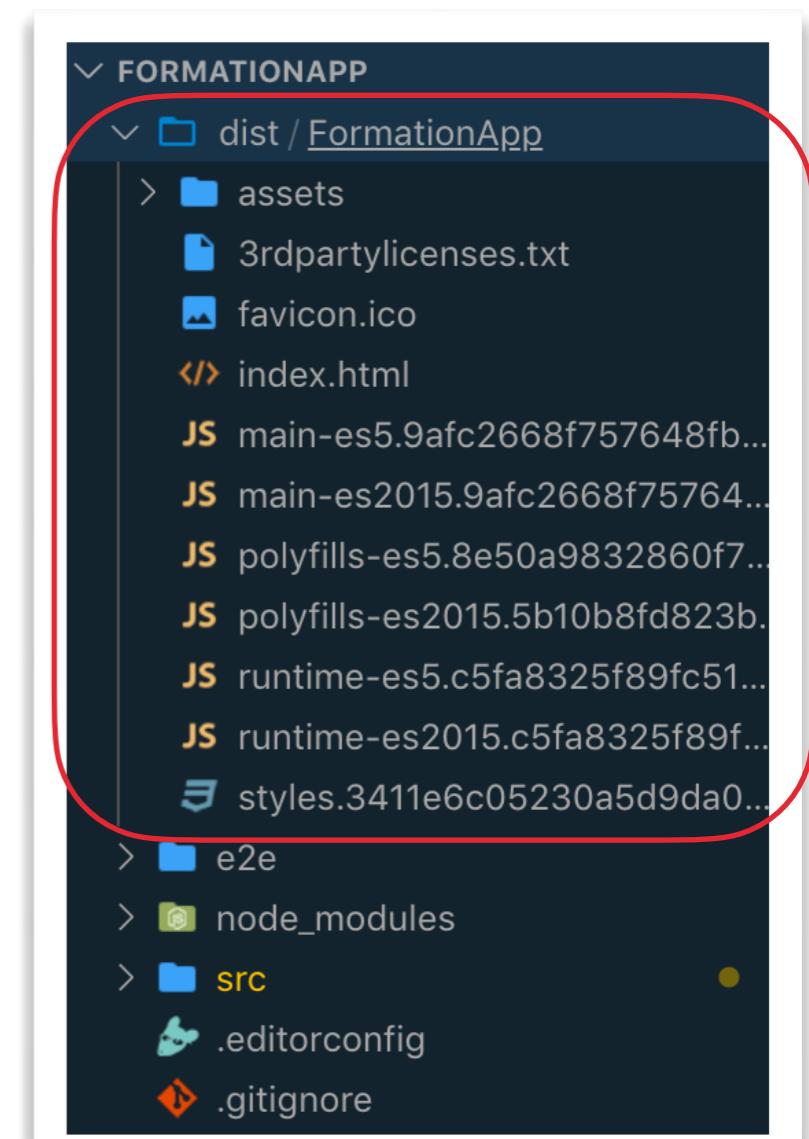
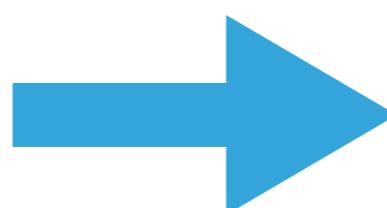
Charger + Créer un dossier Télécharger Actions ▾ USA Est (Virginie du Nord)

Affichage 1 à 11 < >

<input type="checkbox"/> Nom ▾	Dernière modification ▾	Taille ▾	Classe de stockage ▾
--------------------------------	-------------------------	----------	----------------------

BUILD

- ▶ Retour à présent à notre application Angular pour générer notre build.
- ▶ C'est ce build qu'on va charger dans notre compartiment dans S3.
- ▶ Génération du build : `ng build --prod`
- ▶ Création d'un dossier `dist` dans l'arborescence de votre application Angular.



ACCÈS PUBLIC

- ▶ Tout est désormais prêt pour notre déploiement.
- ▶ On revient vers notre compartiment sur S3 et nous allons uploader le contenu de notre dossier dist.
- ▶ N'oubliez de sélectionner "Octroyer un accès en lecture public à ces objets" (Grant public read access to his object(s)) dans "Definir des autorisations" puis "Aucun chiffrement" dans "Définir les propriétés".
- ▶ Ainsi, à l'étape "Vérification", votre interface devrait ressembler à celle du diapo suivant.

Charger X

Sélectionner les fichiers Définir des autorisations Définir les propriétés 4 Vérification

Fichiers Modifier

1 Fichiers Taille : 30.5 Ko

Autorisations Modifier

2 bénéficiaires

Propriétés Modifier

Chiffrement
Non

Classe de stockage
Standard

Metadonnées

Balise

[Précédent](#) Charger

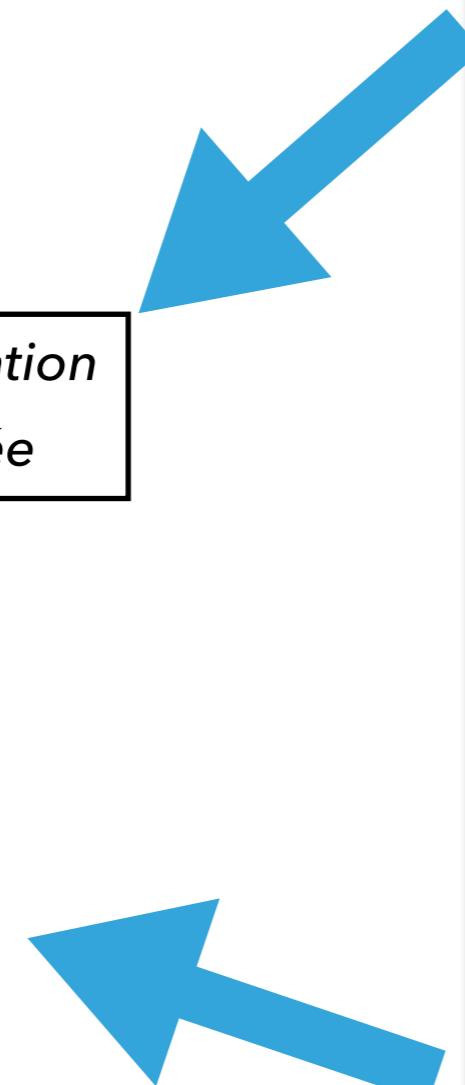
ETAPE FINALE

- ▶ La dernière étape consiste à aller aux propriétés de votre compartiment.
- ▶ Cliquez sur "Herbergement de site web statique" (qui ne nécessite pas de technologies côté serveur).
- ▶ Cliquez sur "Utiliser ce compartiment pour héberger un site web et donnez le nom du document d'index qui sera évidemment index.html

URL

- ▶ On termine par cliquer sur Enregister

**URL de votre application
fraîchement déployée**



Hébergement de site Web statique X

Point de terminaison : `http://cv-test.s3-website-us-east-1.amazonaws.com`

Utiliser ce compartiment pour héberger un site Web En savoir plus

Document d'index i

`index.html`

Document d'erreur i

`error.html`

Règles de redirection (optionnel) i

Rediriger les demandes En savoir plus

Désactiver l'hébergement de site Web

✓ Hébergement du compartiment Annuler Enregister

GITHUB PAGES

GHPAGES

- ▶ Créer un repository sur GitHub et uploadez y votre projet
- ▶ installer angular-cli-ghpages : `ng add angular-cli-ghpages`
- ▶ Vérifier que vous avez déjà effectuer le build de votre application avec:

`ng build --prod`

- ▶ Lancer ensuite la commande :

```
ng build --prod --base-href https://USERNAME.github.io/  
REPOSITORY_NAME/
```

GHPAGES

- ▶ Avant de lancer le déploiement, assurons-nous d'avoir cette confirmation dans le fichier **angular.json** :

```
"deploy": {  
    "builder": "angular-cli-ghpages:deploy",  
    "options": {}  
}
```

- ▶ Lancer la commande : `ng deploy --base-href=/the-repositoryname/`
- ▶ Accéder à votre page https://USERNAME.github.io/REPOSITORY_NAME
- ▶ En cas de mise à jour, relancer la même procédure.