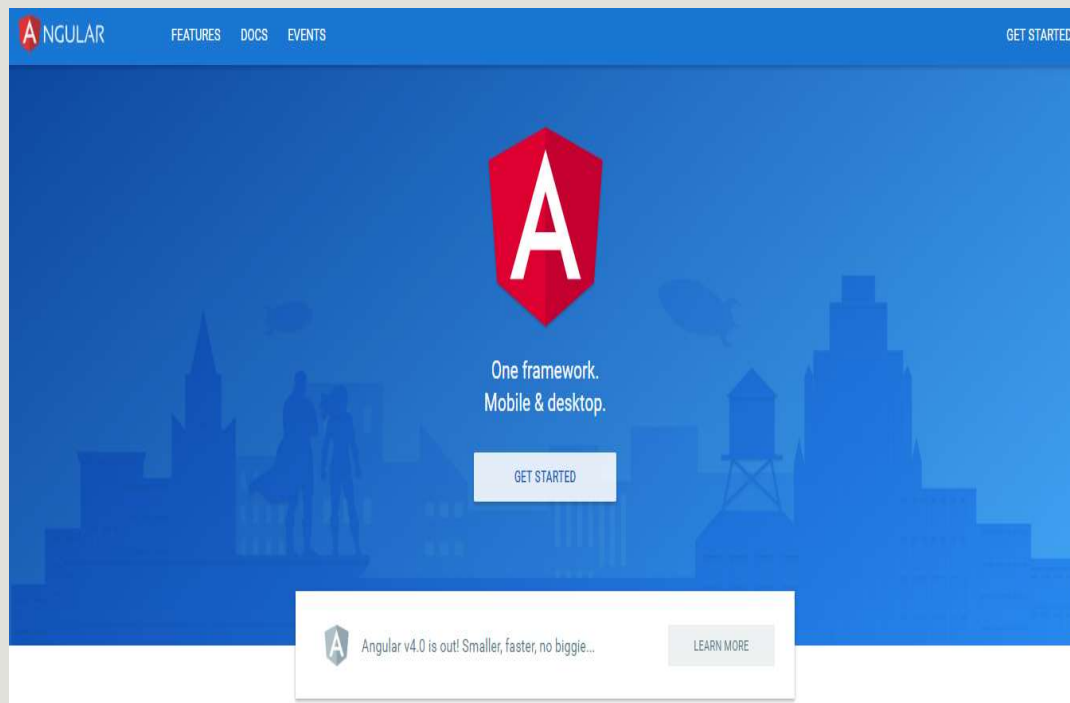


# Angular Introduction

---

AYMEN SELLAOUTI

# Références



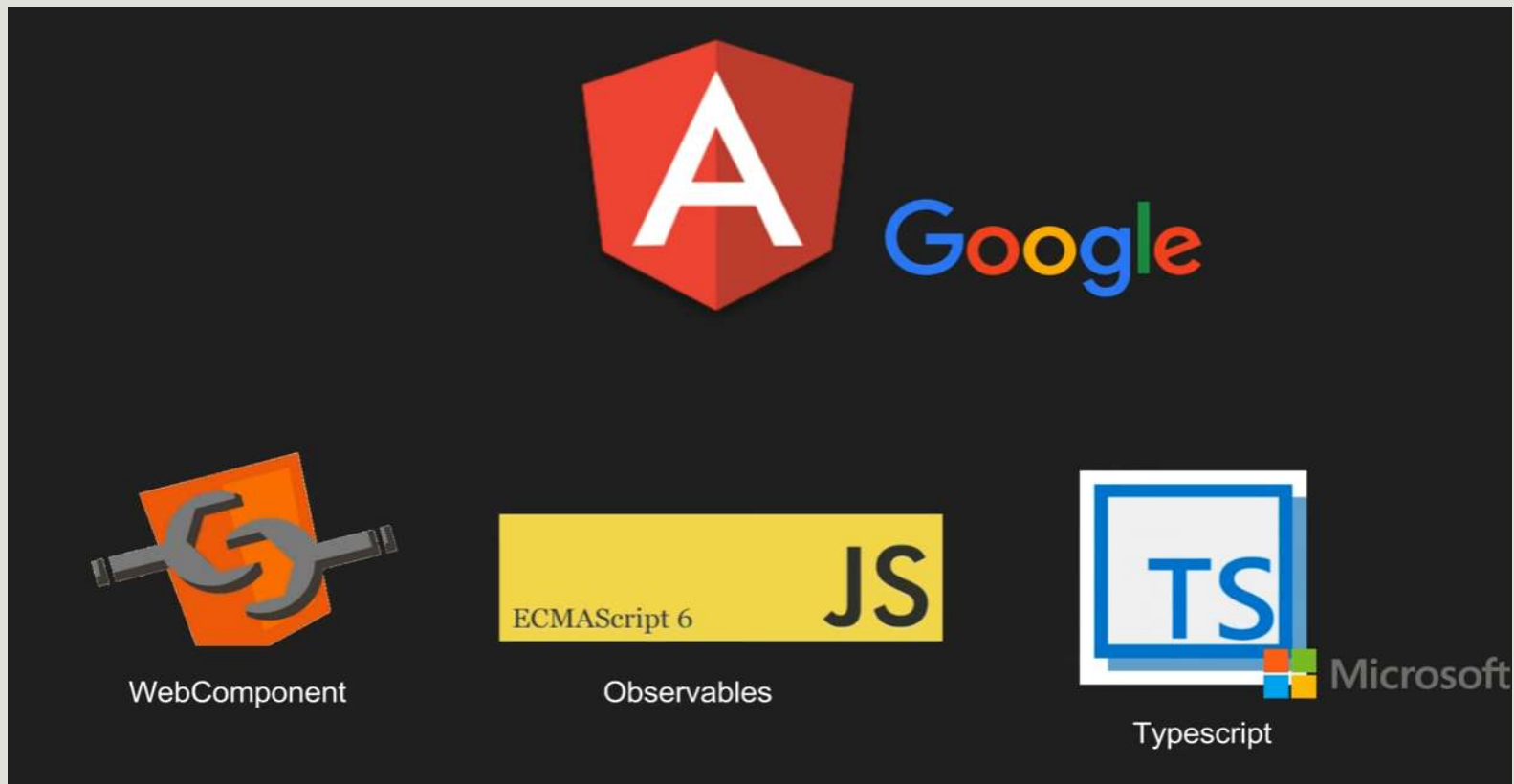
# Plan du Cours

---

1. Introduction
2. Les composants
3. Les directives
- 3 Bis. Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP

# C'est quoi Angular?

---

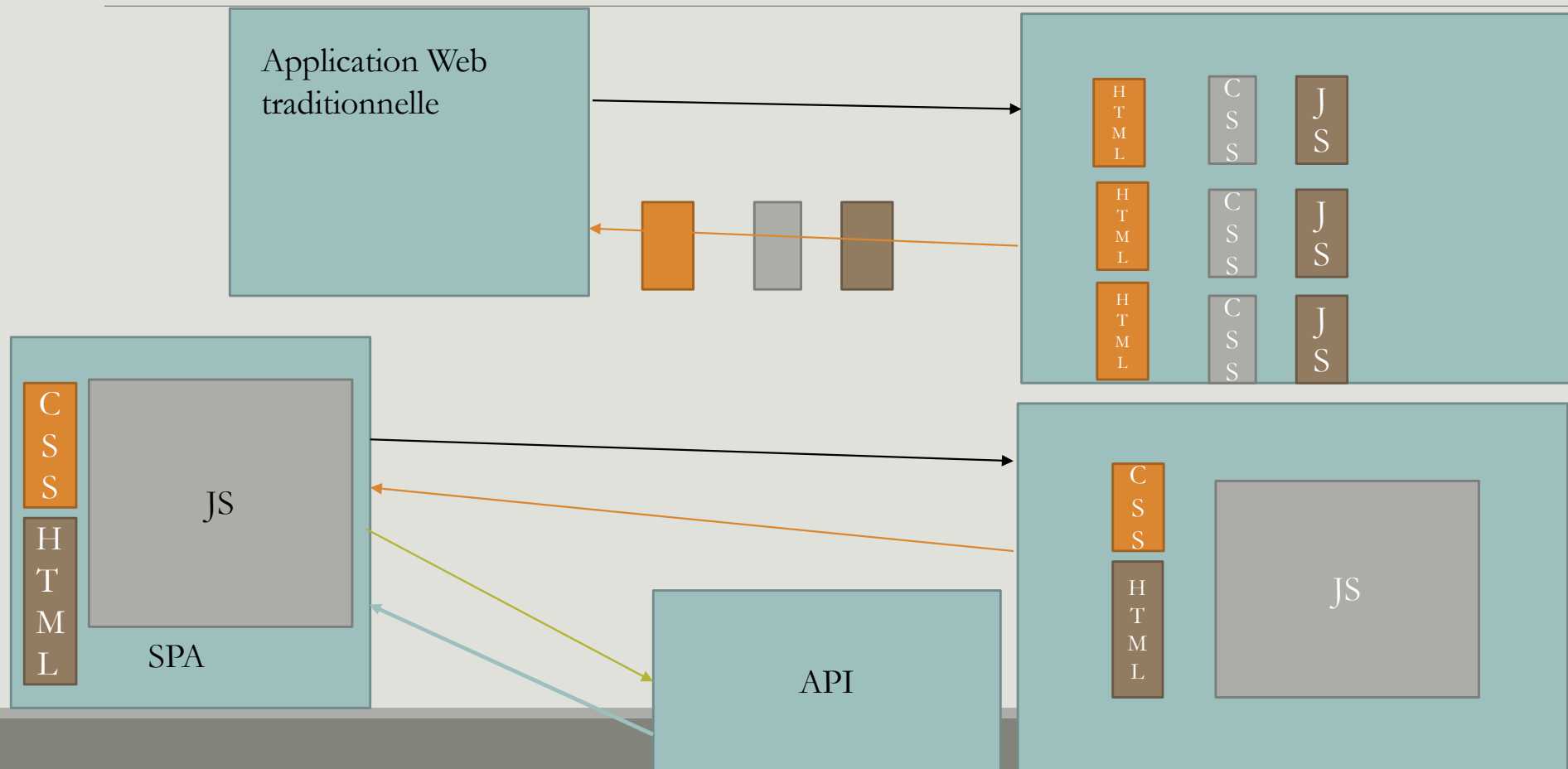


# C'est quoi Angular?

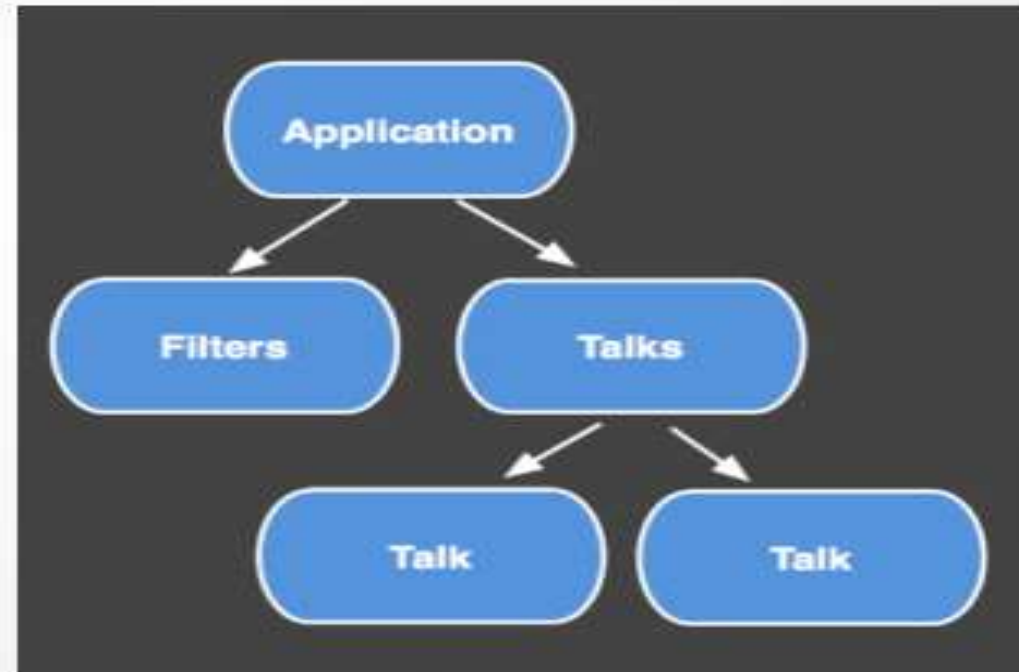
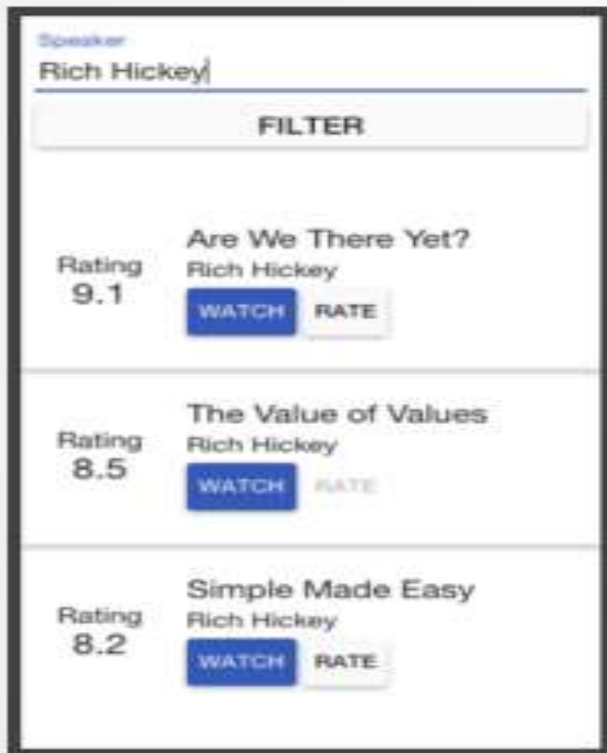
---

- Framework JS
- SPA
- Supporte plusieurs langages : ES5, EC6, TypeScript, Dart
- Modulaire (organisé en composants et modules)
- Rapide
- Orienté Composant

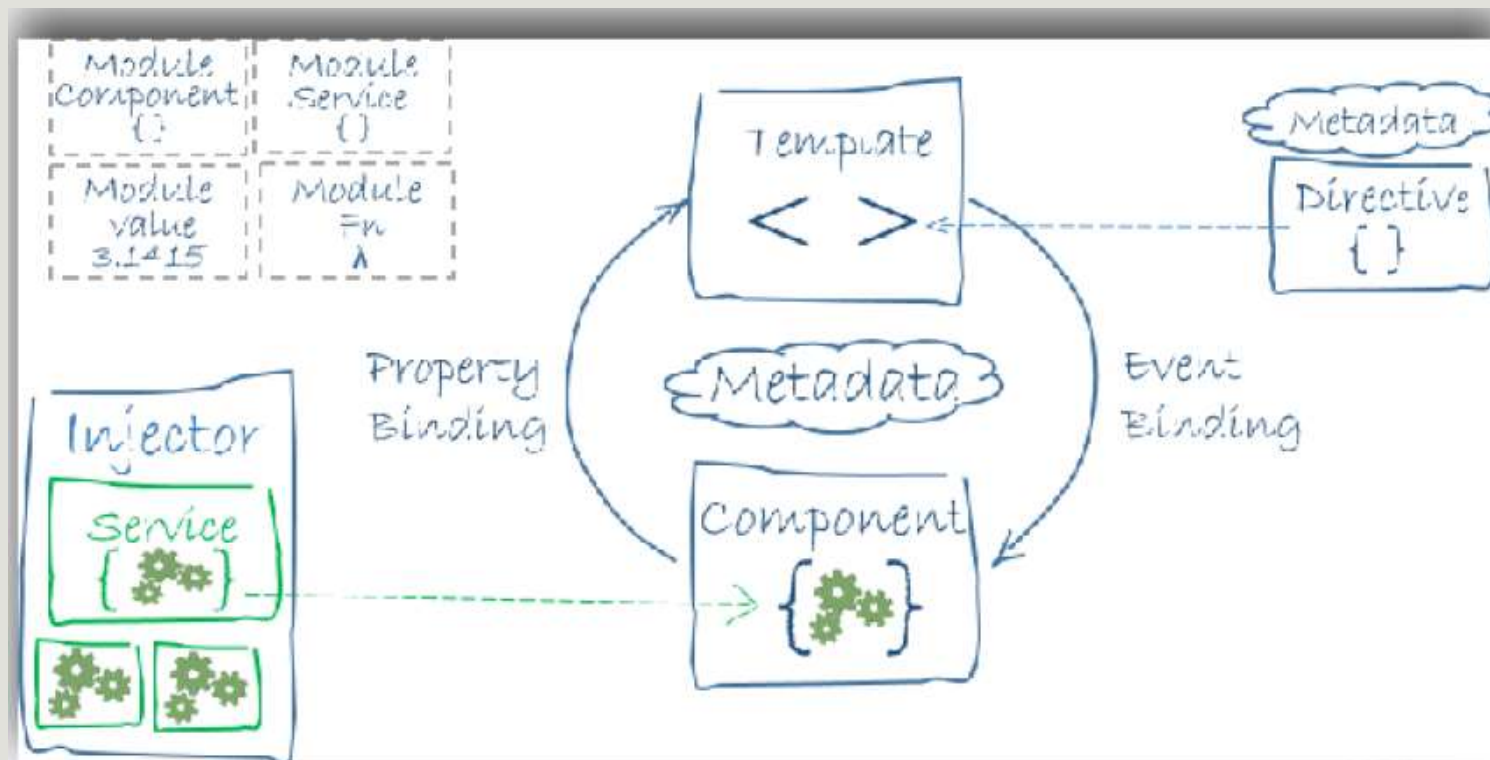
# SPA



# Angular : Arbre de composants



# Architecture Angular





# Principaux concepts et notions

---

Component

Template

DataBinding

Méta données

Service

Route

Injection de  
dépendance

# Module

---

Angular est modulaire (Ca a changé depuis la version 14)

- Chaque application va définir Angular Modules or NgModules
- Chaque module Angular est une classe avec une annotation `@NgModule`
- Chaque application a au moins un module, c'est le module principale.

# AppModule : le module principal

---

➤ le module principal est le module qui permet de lancer l'application de la bootstraper.

➤ Le nom par convention est AppModule.

➤ 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

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

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

# Les librairies d'Angular

---

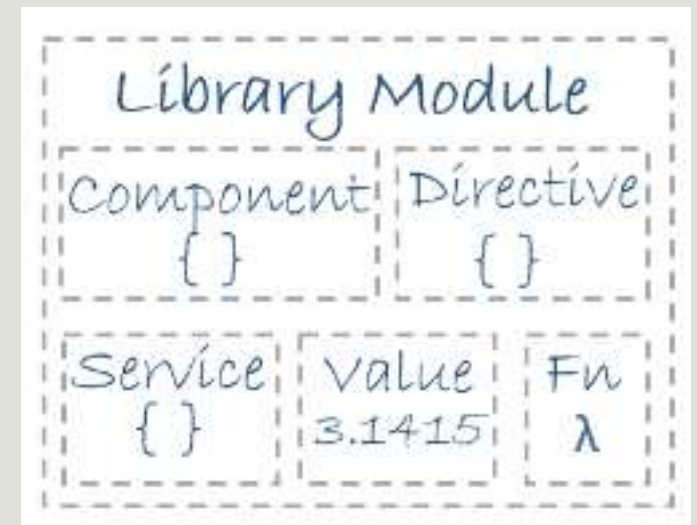
➤ Ensemble de modules JS

➤ Des librairies qui contiennent un ensemble de fonctionnalités.

➤ Toutes les librairies d'Angular sont préfixées par [@angular](#)

➤ Récupérable à travers un import JavaScript.

➤ Exemple pour récupérer l'annotation component : `import { Component } from '@angular/core';`



# Les composants

---

- Le **composant** est la partie principale d'Angular.
- Un composant s'occupe d'une partie de la vue.
- L'interaction entre le composant et la vue se fait à travers une API.

# Template

---

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

# Les méta data

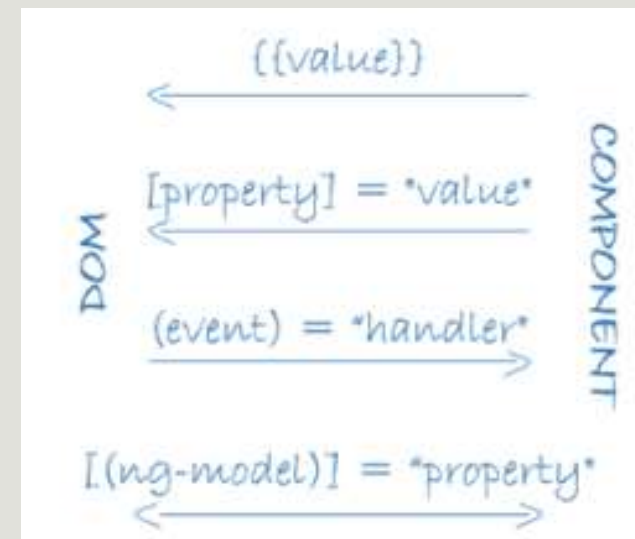
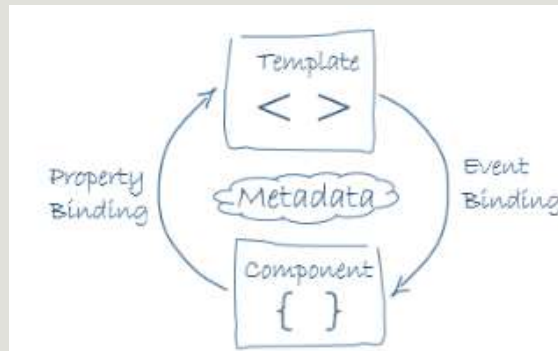
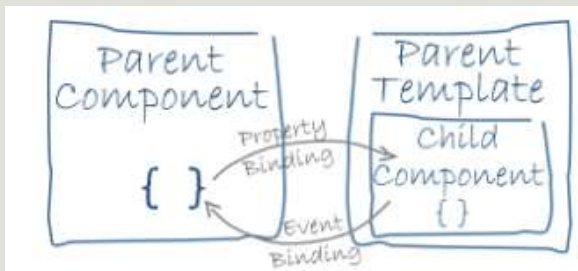
---



- Appelé aussi « decorator », ce sont des informations permettant de décrire les classes.
- `@Component` permet d'identifier la classe comme étant un composant angular.

# Le Data Binding

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





# Les directives

---

➤ Les directives Angular sont des classes avec la métadonnée `@Directive`. Elle permettent de modifier le DOM et de rendre les Template dynamiques.

➤ Apparaissent dans des éléments HTML comme les attributs.



➤ Un composant est une directive à laquelle Angular a associé un Template.

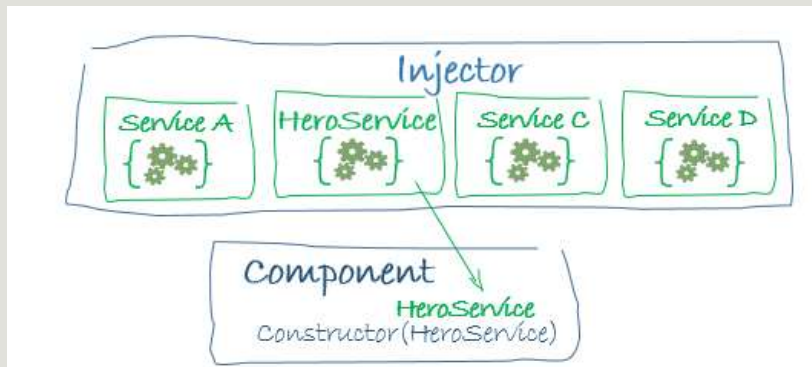
➤ Ils existe deux autres types de directives :

➤ Directives structurelles

➤ Directive d'attributs

# Les services

- Classes permettant d'encapsuler des traitements métiers.
- Doivent être légers.
- Associées aux composants et autres classes par injection de dépendances.



# Installation d'Angular

---

Deux méthodes pour installer un projet Angular.

- Cloner ou télécharger le QuickStart seed proposé par Angular.
- Utiliser le Angular-cli afin d'installer un nouveau projet (conseillé).
- Remarque : L'installation de NodeJs est obligatoire afin de pouvoir utiliser son npm (Node Package Manager).

# Installation d'Angular QuickStart

---

Deux méthodes

- Télécharger directement le projet du dépôt Git

`https://github.com/angular/quickstart`

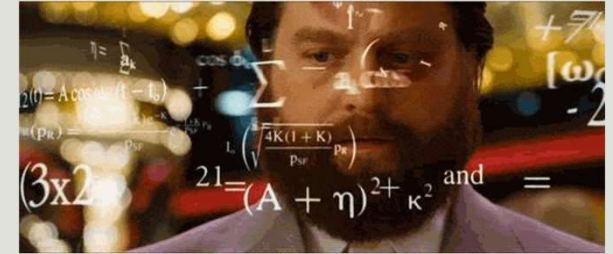
- Ou bien le cloner à l'aide de la commande suivante :

`git clone https://github.com/angular/quickstart.git quickstart`

- Se positionner sur le projet
- Installer les dépendance à l'aide de npm : `npm install`
- lancer le projet à l'aide de npm : `npm start`

# Installation d'Angular

## Angular Cli

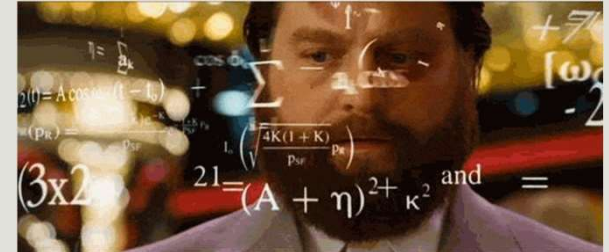


- Nous allons installer notre première application en utilisant **angular Cli**.
- Si vous avez Node c'est bon, sinon, installer **NodeJs** sur votre machine. Vous devez avoir une version de **node nécessaire pour la version Angular que vous installez**.
- Une fois installé vous disposez de npm qui est le **Node Package Manager**. Afin de vérifier si vous avez NodeJs installé, tapez **npm -v**.
- Installer maintenant le Cli en tapant la : **npm install -g @angular/cli**
  - **npm install -g @angular/cli@16.2.16** installe la version **16.2.16**
  - **npm view @angular/cli** affiche la liste des versions de la cli
- Installer un nouveau projet à l'aide de la commande **ng new nomProjet**
- **npx @angular/cli@16.2.16** new projectName
- Afin d'avoir du help pour le cli tapez **ng help**
- Lancer le projet en utilisant la commande **ng serve**

# Angular dépendances

Angular CLI version	Angular version	Node.js version	TypeScript version	RxJS version
29 ~10.1.7	~10.1.6	^10.13.0    ^12.11.1	>= 3.9.4 <= 4.0.8	^6.5.5
30 ~10.2.4	~10.2.5	^10.13.0    ^12.11.1	>= 3.9.4 <= 4.0.8	^6.5.5
31 ~11.0.7	~11.0.9	^10.13.0    ^12.11.1	~4.0.8	^6.5.5
32 ~11.1.4	~11.1.2	^10.13.0    ^12.11.1	>= 4.0.8 <= 4.1.6	^6.5.5
33 ~11.2.19	~11.2.14	^10.13.0    ^12.11.1	>= 4.0.8 <= 4.1.6	^6.5.5
34 ~12.0.5	~12.0.5	^12.14.1    ^14.15.0	~4.2.4	^6.5.5
35 ~12.1.4	~12.1.5	^12.14.1    ^14.15.0	>= 4.2.4 <= 4.3.5	^6.5.5
36 ~12.2.0	~12.2.0	^12.14.1    ^14.15.0	>= 4.2.4 <= 4.3.5	^6.5.5    ^7.0.1
37 ~13.0.4	~13.0.3	^12.20.2    ^14.15.0    ^16.10.0	~4.4.4	^6.5.5    ^7.4.0
38 ~13.1.4	~13.1.3	^12.20.2    ^14.15.0    ^16.10.0	>= 4.4.4 <= 4.5.5	^6.5.5    ^7.4.0
39 ~13.2.6	~13.2.7	^12.20.2    ^14.15.0    ^16.10.0	>= 4.4.4 <= 4.5.5	^6.5.5    ^7.4.0
40 ~13.3.0	~13.3.0	^12.20.2    ^14.15.0    ^16.10.0	>= 4.4.4 < 4.7.0	^6.5.5    ^7.4.0
41 ~14.0.7	~14.0.7	^14.15.0    ^16.10.0	>= 4.6.4 < 4.8.0	^6.5.5    ^7.4.0
42 ~14.1.3	~14.1.3	^14.15.0    ^16.10.0	>= 4.6.4 < 4.8.0	^6.5.5    ^7.4.0
43 ~14.2.0	~14.2.0	^14.15.0    ^16.10.0	>= 4.6.4 < 4.9.0	^6.5.5    ^7.4.0
44 ~15.0.0	~15.0.0	^14.20.0    ^16.13.0    ^18.10.0	~4.8.4	^6.5.5    ^7.4.0

# Installation d'Angular Angular Cli



- Positionnez vous maintenant dans le dossier
- Tapez la commande suivante : `ng new nomNewProject`
- lancez le projet à l'aide de npm : `ng serve`
- Naviguez vers l'adresse mentionnée.
- Vous pouvez configurer le Host ainsi que le port avec la commande suivante : `ng serve --host leHost --port lePort`
- Pour plus de détails sur le cli visitez <https://cli.angular.io/>

# Quelques commandes du Cli

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



# Ajouter Bootstrap

---

On peut ajouter Bootstrap de plusieurs façons :

- 1- Via le CDN à ajouter dans le fichier index.html
- 2- En le téléchargeant du site officiel
- 3- Via npm
  - `npm install bootstrap --save`

# Ajouter Bootstrap

---

Pour ajouter les dossiers téléchargés on peut le faire de deux façons :

1- En l'ajoutant dans index.html

2- En ajoutant 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",  
],  
"scripts": [  
  "../node_modules/jquery/dist/jquery.min.js",  
  "../node_modules/popper.js/dist/umd/popper.min.js",  
  "../node_modules/bootstrap/dist/js/bootstrap.min.js"  
],
```

# Ajouter Bootstrap

---

Ajouter dans le fichier `src/style.css` un import de vos bibliothèques.

```
@import "~bootstrap/dist/css/bootstrap.css";
```

Essayer la même chose avec font-awesome.

# Angular

# Les composants

---

AYMEN SELLAOUTI

# Objectifs

---

1. Comprendre la définition du composant
2. Assimiler et pratiquer la notion de Binding
3. Gérer les interactions entre composants.

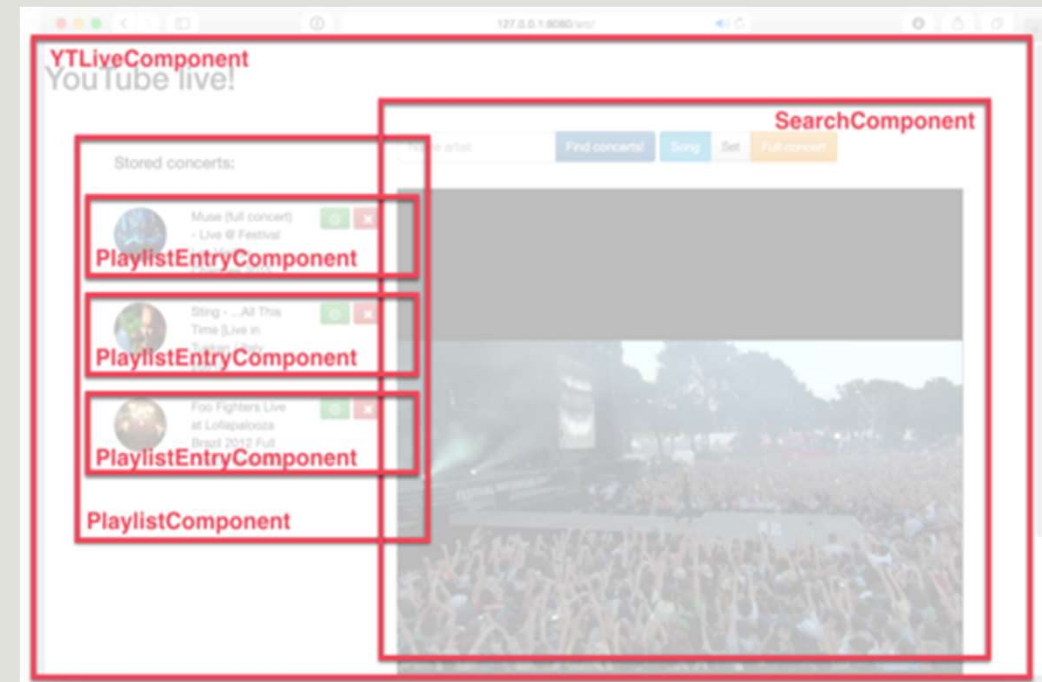
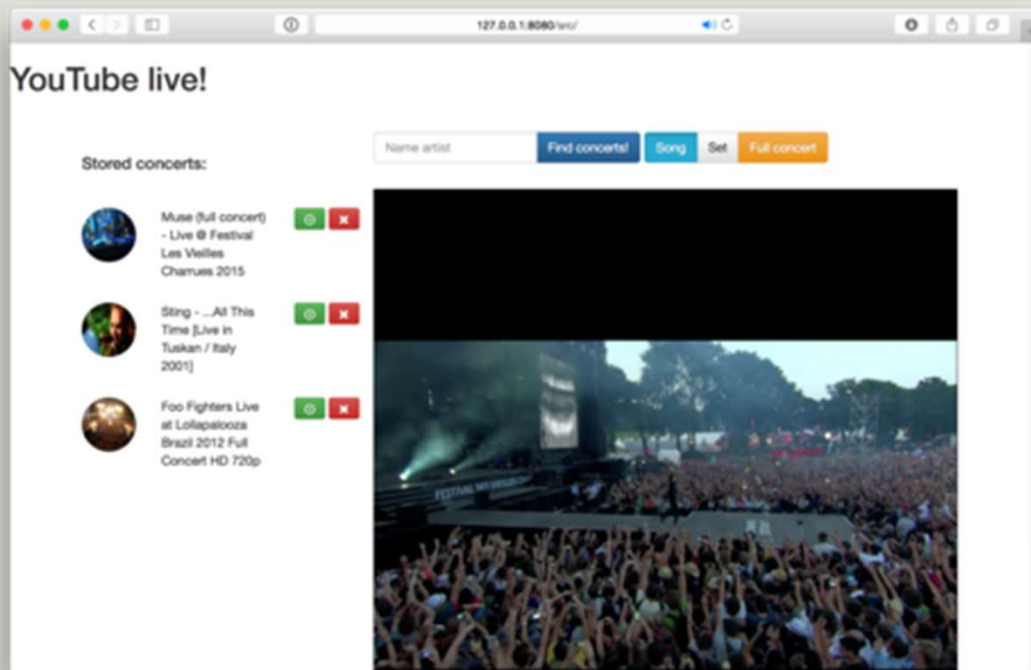
# Qu'est ce qu'un composant (Component)

---

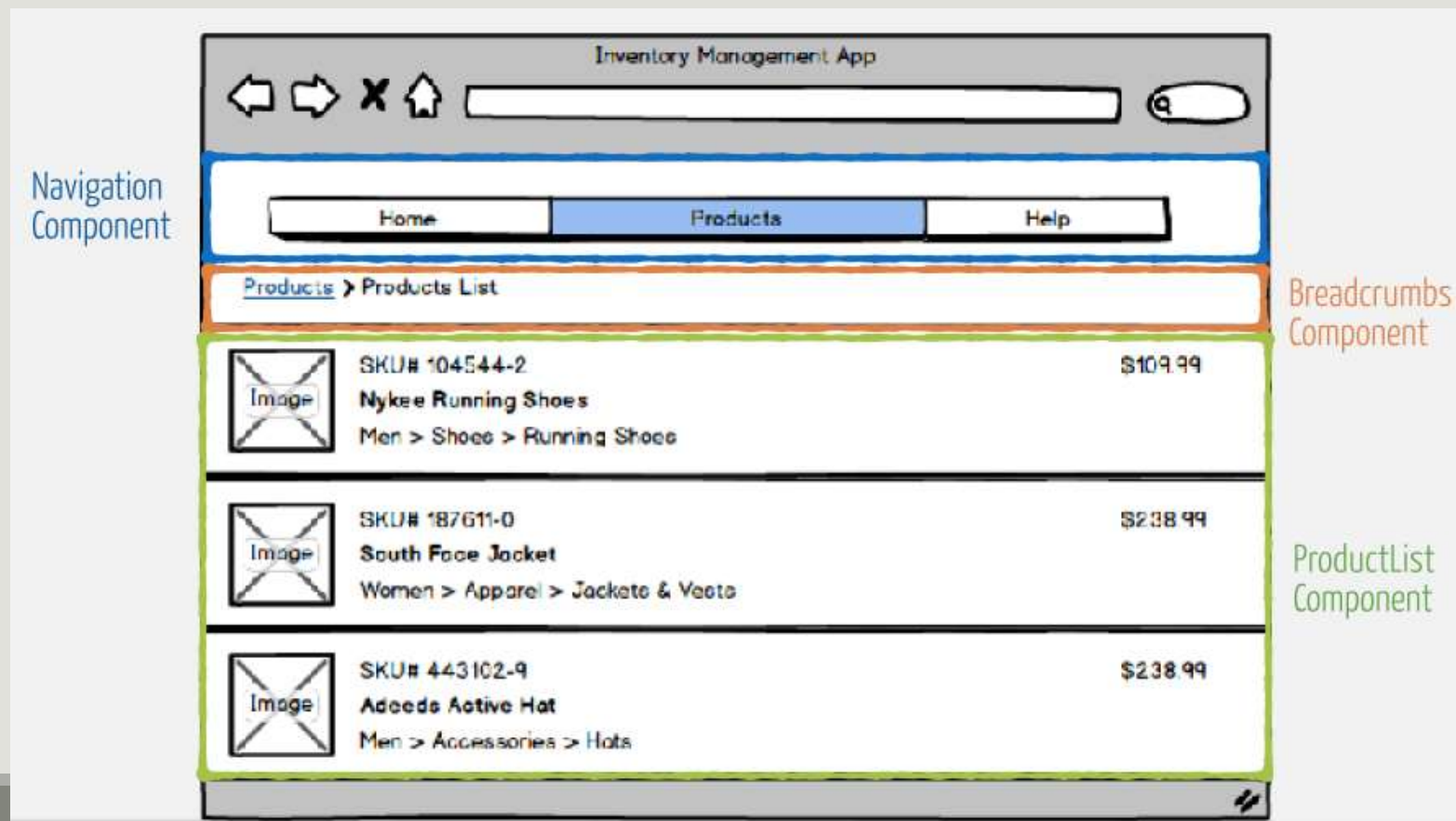
- Un **composant** est une **classe** qui permet de **gérer une vue**. Il se charge **uniquement** de cette vue là.  
Plus simplement, un composant est un fragment HTML géré par une classe JS (component en angular et controller en angularJS)
- Une application Angular est un **arbre de composants**
-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

**NB : Dans le reste du cours les mots composant et component représentent toujours un composant Angular.**

# Quelques exemples



# Quelques exemples





# Quelques exemples

---

Product Row  
Component

	SKU# 104544-2 <b>Nykee Running Shoes</b> Men > Shoes > Running Shoes	\$109.99
	SKU# 187611-0 <b>South Face Jacket</b> Women > Apparel > Jackets & Vests	\$238.99
	SKU# 443102-9 <b>Adeeds Active Hat</b> Men > Accessories > Hats	\$238.99

# Quelques exemples

---

Product Image Component	Product Department Component	Price Display Component
	SKU# 104544-2 <b>Nykee Running Shoes</b> Men > Shoes > Running Shoes	\$109.99

# Premier Composant

---

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works for tekup people !';
}
```

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.

**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

# Création d'un composant

---

- Deux méthodes pour créer un composant

- Manuelle
- Avec le Cli

- 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**

- Cli

- Avec la commande **ng generate component my-new-component** ou son raccourci **ng g c my-new-component**

# Création d'un composant

---

- La commande `generate` possède plusieurs options

OPTION	DESCRIPTION
<code>--inlineStyle=true   false</code>	Inclus les styles css dans le composant Aliases: <code>-s</code>
<code>--inlineTemplate=true   false</code>	Inclus le template dans le composant Aliases: <code>-t</code>
<code>--prefix=<i>prefix</i></code>	Le préfixe à appliquer pour la génération des composants Valeur par défaut: <code>app</code> Aliases: <code>-p</code>

# Property Binding

---



# 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]="**varOuCte**"
  - **bind**-propriété="**varOuCte**"

```
<div [style.backgroundColor]="color">  
  Color  
</div>
```

# 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)="fct()">>
  - **on**-evenement

```
a (click)="goToCv()">Go to Cv</a>
```



# Property Binding et Event Binding

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

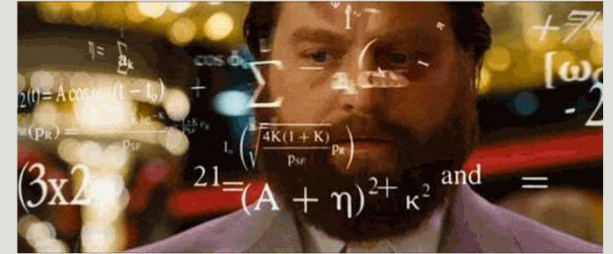
@Component({
  selector: 'inter-interpolation',
  template: `interpolation.html`,
  styles: []
})
export class InterpolationComponent {
  nom:string ='Aymen Sellaouti';
  age:number =35;
  adresse:string ='Chez moi ou autre part :)';
  getName() {
    return this.nom;
  }
  modifier(newName) {
    this.nom=newName;
  }
}
```

Component

```
<hr>
Nom : {{nom}}<br>
Age : {{age}}<br>
Adresse : {{adresse}}<br>
//Property Binding
<input #name
[value]="getName()">
//Event Binding
<button
(click)="modifier(name.value)"
>Modifier le nom</button>
<hr>
```

Template

# Exercice



- Créer un nouveau composant. Ajouter y un Div et un input de type texte.
- Fait en sorte que lorsqu'on écrit une couleur dans l'input, ca devienne la couleur du Div.
- Ajouter un bouton. Au click sur le bouton, il faudra que le Div reprenne sa couleur par défaut.
- Ps : pour accéder au contenu d'un élément du Dom utiliser `#nom` dans la balise et accéder ensuite à cette propriété via le `nom`.
- Pour accéder à une propriété de style d'un élément on peut binder la propriété `[style.nomPropriété]` exemple `[style.backgroundColor]`

# 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**
- Afin de pouvoir utiliser ngModel vous devez importer le FormsModule dans app.module.ts

# Property Binding et Event Binding

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

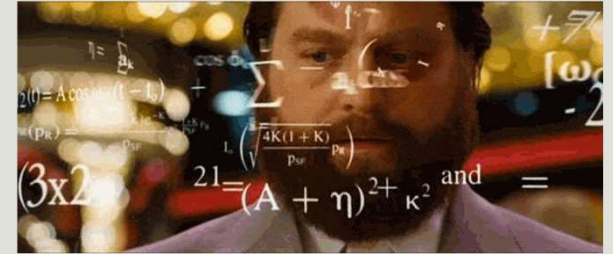
@Component({
  selector: 'app-two-way',
  templateUrl: './two-
way.component.html',
  styleUrls: ['./two-
way.component.css']
})
export class TwoWayComponent {
  two:any="myInitValue";
}
```

Component

```
<hr>
Change me if u can
<input
 [ (ngModel) ]="two">
<br>
it's always me :d
{{two}}
```

Template

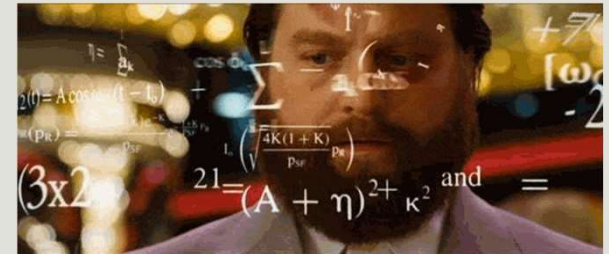
# Exercice




- Le but de cet exercice est de créer un aperçu de carte visite.
- 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>

# Exercice





Aymen Sellaouti  
trainer

"I'm the new Sinatra, and since I made it here I can make it anywhere, yeah,  
they love me everywhere"

↻ Auto Rotation

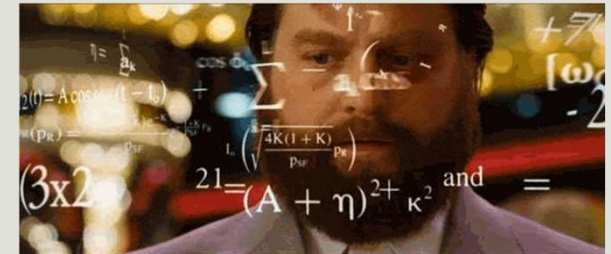
name :

firstname :

job :

path :

# Exercice



## Two Way Binding



A user profile card for Sellaouti Aymen. It features a circular profile picture of a man with a beard. Below the picture, the name 'Sellaouti Aymen' is displayed, followed by the title 'Enseignant'. A quote 'tant qu'il y a de la vie il y a de l'espoir' is shown in a light blue box. At the bottom, there is a small icon of a refresh button and the text 'Auto Rotation'.

Nom :  
Sellaouti

Prénom :  
aymen

Job :  
Enseignant

image :  
as.jpg

Citation Favorite :  
tant qu'il y a de la vie il y a de l'espoir

Décrivez nous votre travail :  
J enseigne aux étudiants les technos du Web

Mots clé de votre travail :  
HTML CSS JS PHP Symfony Angular

## Two Way Binding

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

J enseigne aux étudiants les technos du Web

HTML CSS JS PHP Symfony Angular

235	114	35
Followers	Following	Projects

f G+ t

Nom :  
Sellaouti

Prénom :  
aymen

Job :  
Enseignant

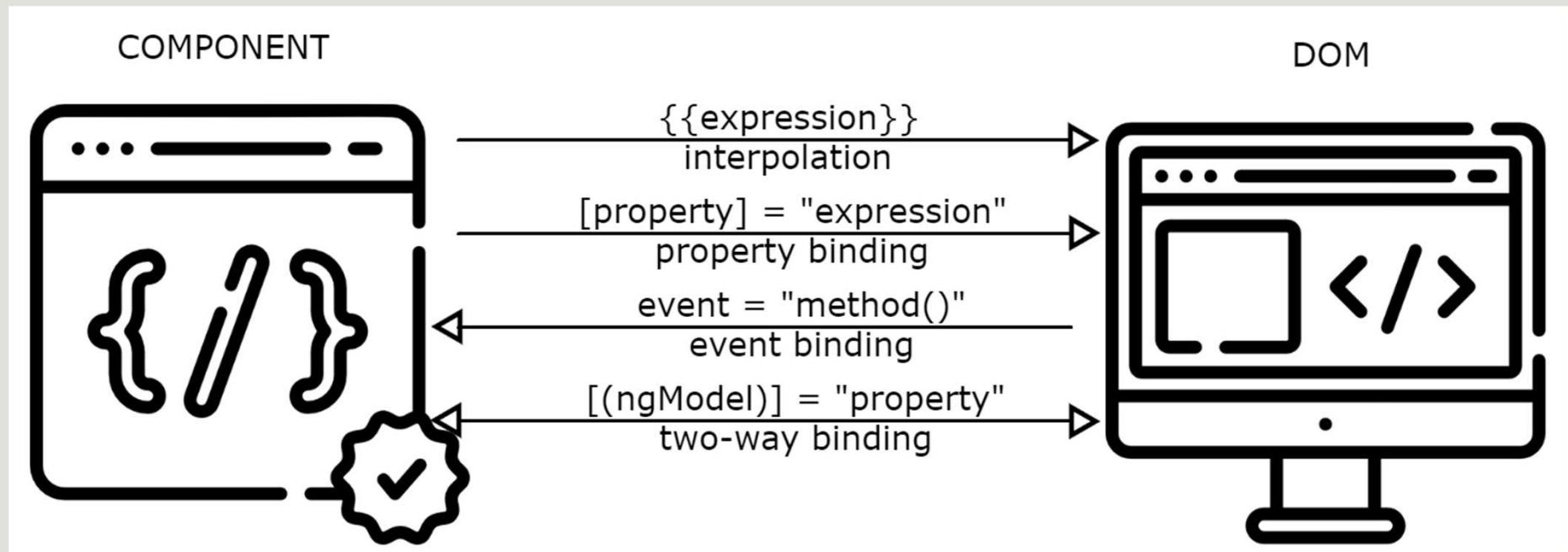
image :  
as.jpg

Citation Favorite :  
tant qu'il y a de la vie il y a de l'espoir

Décrivez nous votre travail :  
J enseigne aux étudiants les technos du Web

Mots clé de votre travail :  
HTML CSS JS PHP Symfony Angular

# Résumé : Property Binding





# Récap Binding

```
<div [style.backgroundColor]="color">
  Color
</div>

<input [(ngModel)]="color"
  type="text"
  class="form-control"
>
le contenu de la propriété color est {{color}}
<button (click)="loggerMesData()">log data</button>
<br>
<a (click)="goToCv()">Go to Cv</a>
```

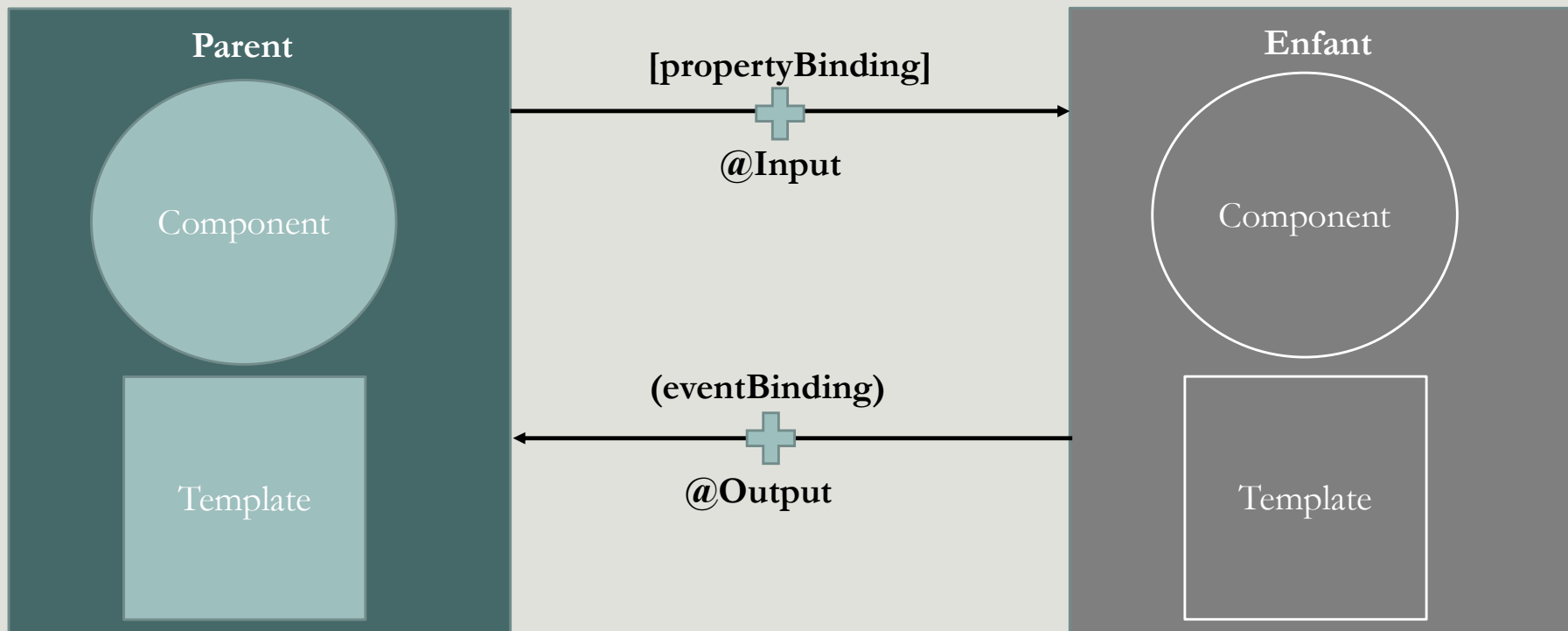
HTML

```
@Component ({
  selector: 'app-color',
  templateUrl: './color.component.html',
  styleUrls: ['./color.component.css'],
  providers: [PremierService]
})
export class ColorComponent implements OnInit {
  color = 'red';
  constructor() { }

  ngOnInit() {}
  processReq(message: any) {
    alert(message);
  }
  loggerMesData() {
    this.premierService.logger('test');
  }
  goToCv() {
    const link = ['cv'];
    this.router.navigate(link);
  }
}
```

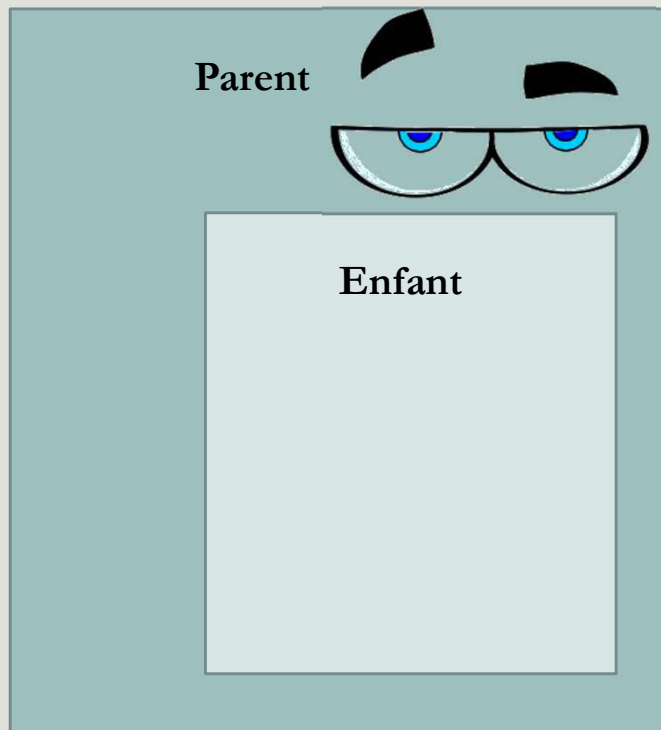
TS

# Interaction entre composants



# Pourquoi ?

Le père voit le fils, le fils ne voit pas le père

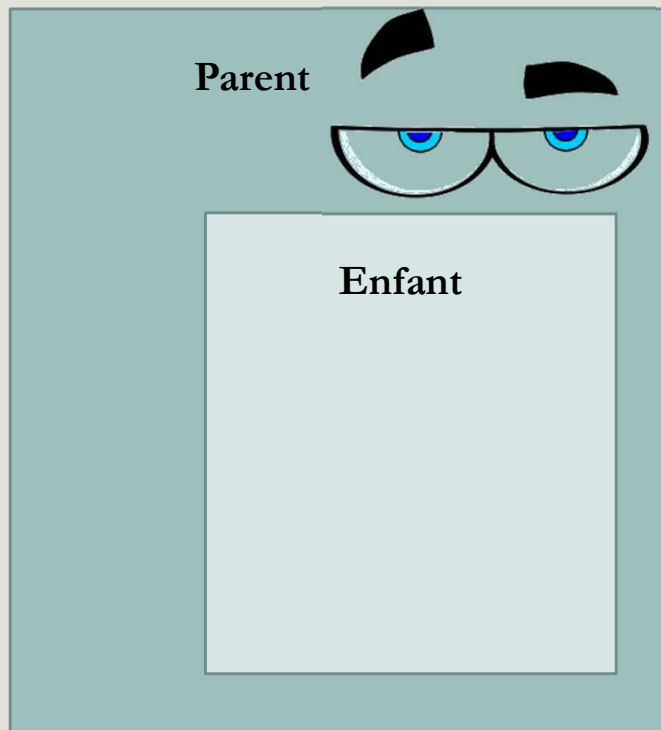


```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <p>Je suis le composant père </p>
    <forma-fils></forma-fils>
  `,
  styles:
})
export class AppComponent {
  title = 'app works !';
}
```

# Interaction du père vers le fils

Le père peut directement envoyer au fils des données par **Property Binding**

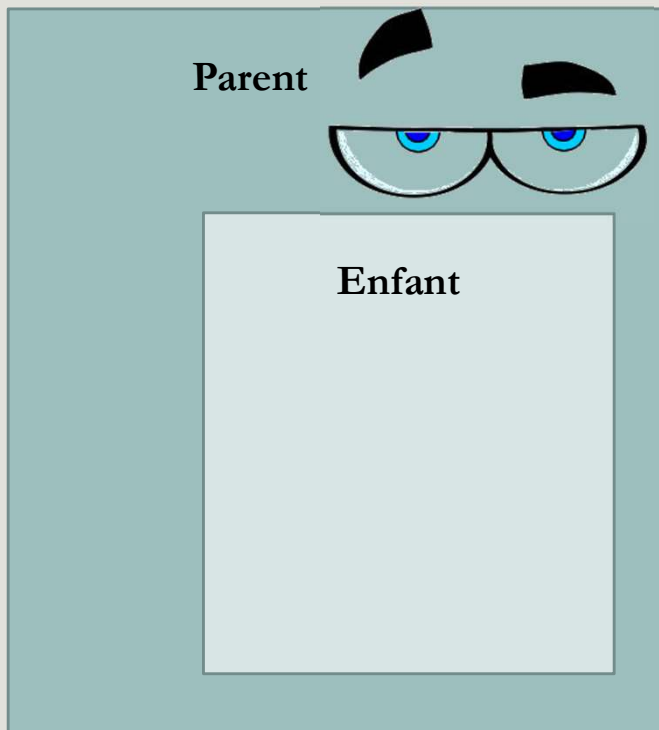


```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <p>Je suis le composant père </p>
    <forma-fils></forma-fils>
  `,
  styles:
})
export class AppComponent {
  title = 'app works !';
}
```

# Interaction du père vers le fils

**Problème :** Le père voit le fils mais pas ces propriétés !!! **Solution :** les rendre visible avec Input

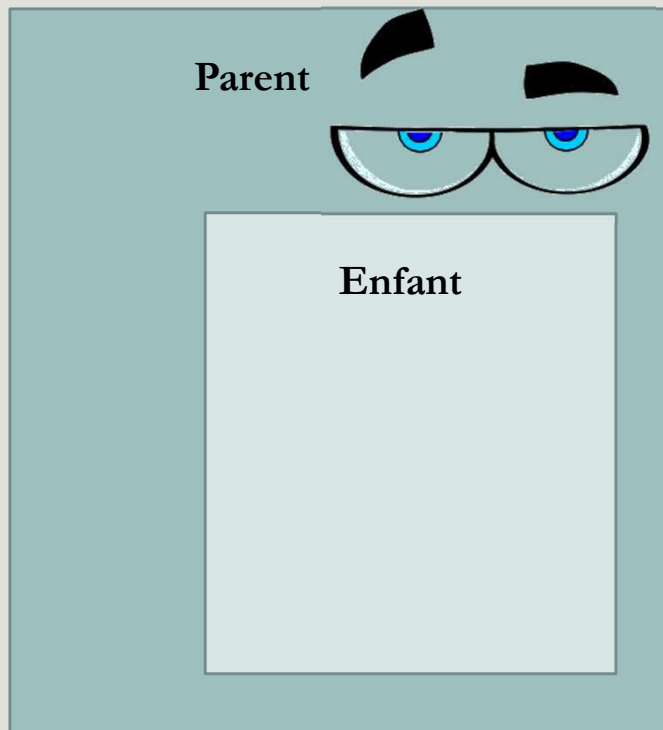


```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <p>Je suis le composant père </p>
    <forma-fils></forma-fils>
  `,
  styles:
})
export class AppComponent {
  title = 'app works !';
}
```

# Interaction du père vers le fils

Problème : Le père voit le fils mais pas ces propriétés !!! Solution : les rendre visible avec Input



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

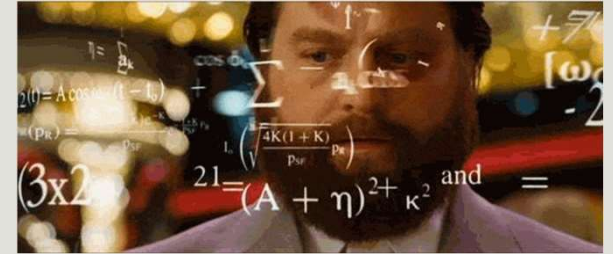
@Component({
  selector: 'app-root',
  template: `
    <p>Je suis le composant père </p>
    <forma-fils [external]="title">
  </forma-fils>
`,
  styles:
})
export class AppComponent {
  title = 'app works !';
}
```

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

@Component ({
  selector: 'app-input',
  templateUrl:
    './input.component.html',
  styleUrls:
    ['./input.component.css']
})
export class InputComponent
{
  @Input () external:string;
}
```

# Exercice

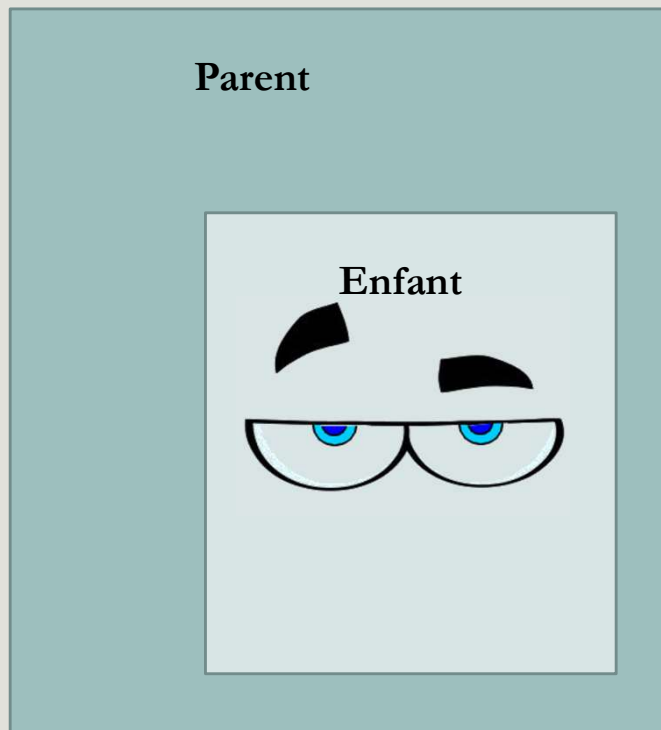
---



- Créer un composant fils
- Récupérer la couleur du père dans le composant fils
- Faire en sorte que le composant fils affiche la couleur du background de son père

# Interaction du fils vers le père

L'enfant ne peut pas voir le parent. Appel de l'enfant vers le parent. Que faire ?



```
import {Component} from '@angular/core';
@Component({
  selector: 'bind-output',
  template: `Bonjour je suis le fils`,
  styleUrls: ['./output.component.css']
})
export class OutputComponent {
  valeur:number=0;
}
```



# Interaction du fils vers le père

Solution : Pour entrer c'est un input pour sortir c'est sûrement un output. Externaliser un évènement en utilisant l'Event Binding.



```
import {Component, EventEmitter, Output} from '@angular/core';
@Component({
  selector: 'bind-output',
  template: `
    <button (click)="incrementer()"></button> `,
  styleUrls: ['./output.component.css']
})
export class OutputComponent {
  valeur:number=0;
  // On déclare un evenement
  @Output() valueChange=new EventEmitter();
  incrementer() {
    this.valeur++;
    this.valueChange.emit
      this.valeur);
  }
}
```

# Interaction du père vers le fils

La variable \$event est la variable utilisée pour transmettre les informations.

Parent

Enfant

Mon père va ensuite intercepter l'événement et récupérer ce que je lui ai envoyé à travers la variable \$event et va l'utiliser comme il veut

```
import { Component, EventEmitter, Output } from '@angular/core';
@Component({
  selector: 'bind-output',
  template: `
<button (click)="incrementer()">+</button> `,
  styleUrls: ['./output.component.css']
})
export class OutputComponent {
  valeur:number=0;
  // On déclare un événement
  @Output() valueChange=new EventEmitter()
  incrementer(){
    this.valeur++;
    this.valueChange.emit(
      this.valeur
    );
  }
}
```

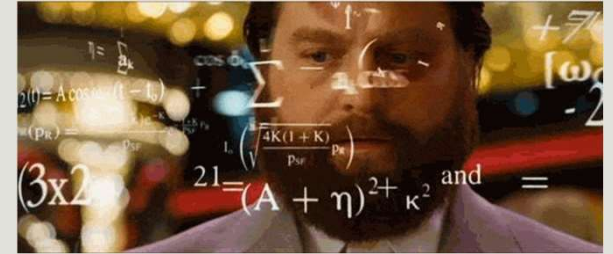
Enfant

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
<h2> {{result}}</h2>
<bind-output
(valueChange)="showValue($event)"
"></bind-output>
`,
  styles: [],
})
export class AppComponent {
  title = 'app works !';
  result:any='N/A';
  showValue(value){
    this.result=value;
  }
}
```

Parent

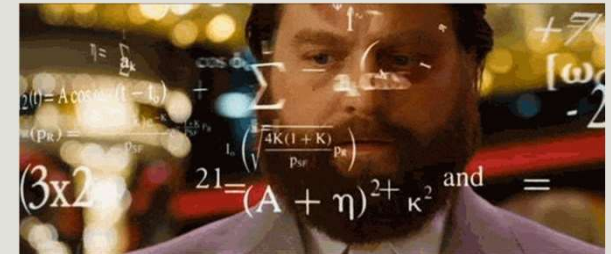
# Exercice

---

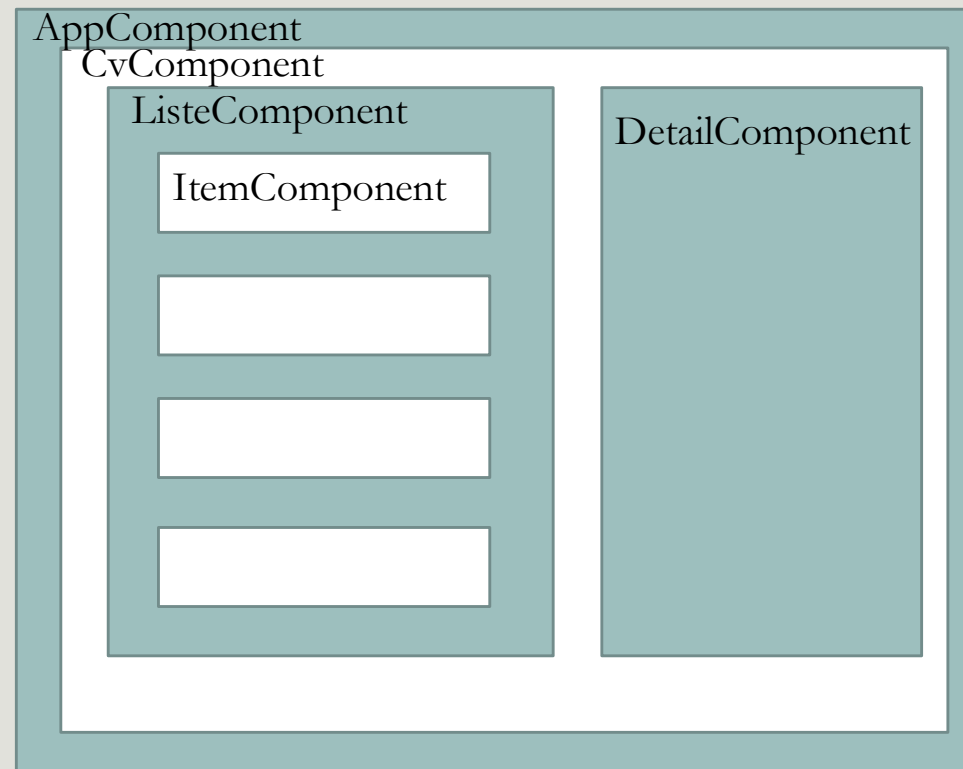


- Ajouter une variable `myFavoriteColor` dans le composant du fils.
- Ajouter un bouton dans le composant Fils
- Au click sur ce bouton, la couleur de background du Div du père doit prendre la couleur favorite du fils.

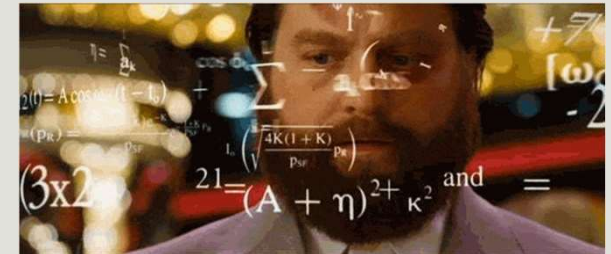
# Exercice



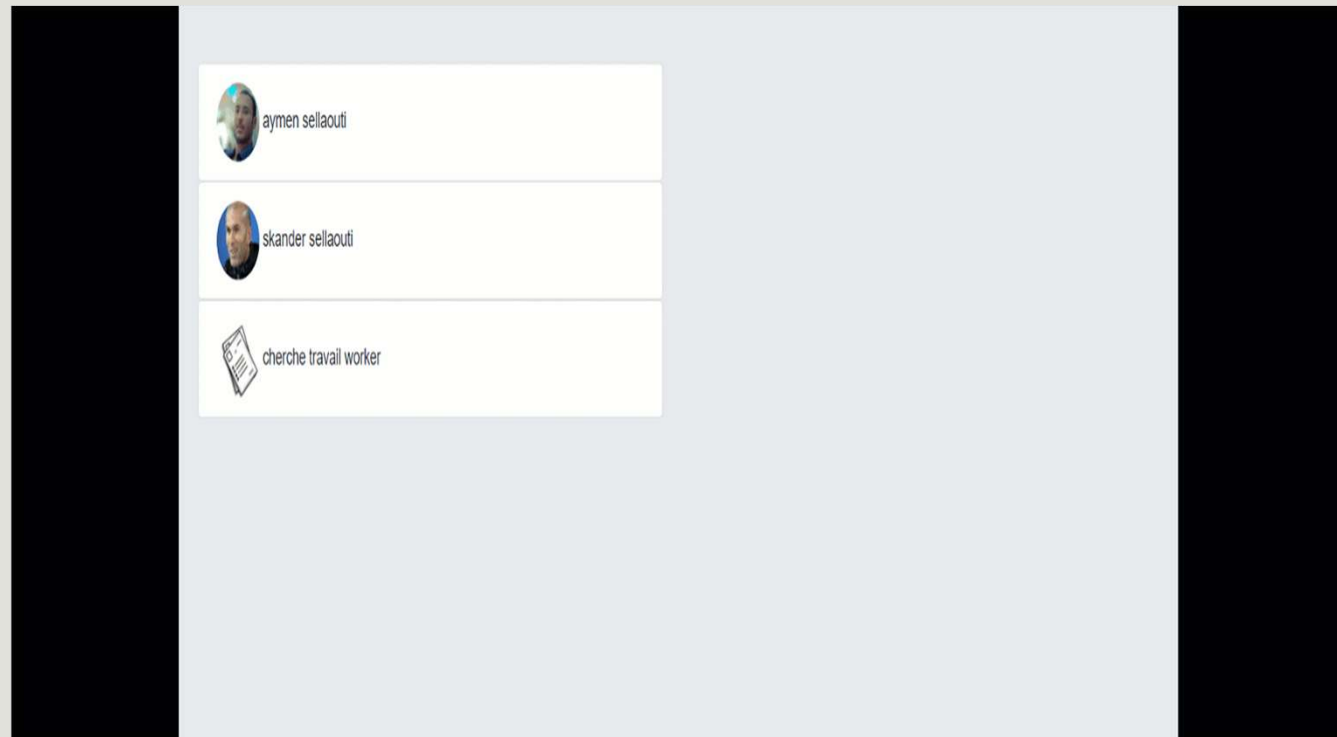
- Le but de cet exercice est de créer une mini plateforme de recrutement.
  - La première étape est de créer la structure suivante avec une vue contenant deux parties :
    - Liste des Cvs inscrits
    - Détail du Cv qui apparaîtra au click
  - Il vous est demandé juste d'afficher un seul Cv et de lui afficher les détails au click.
- Il faudra suivre cette architecture.



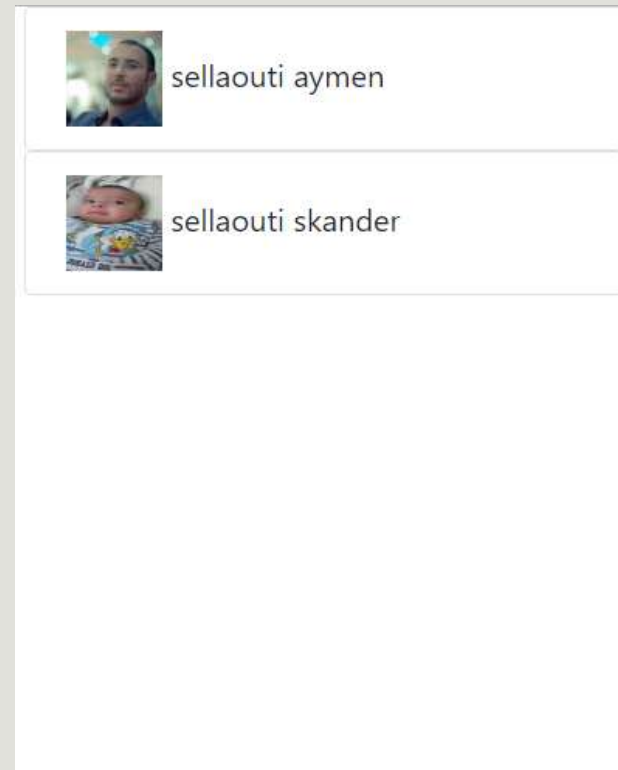
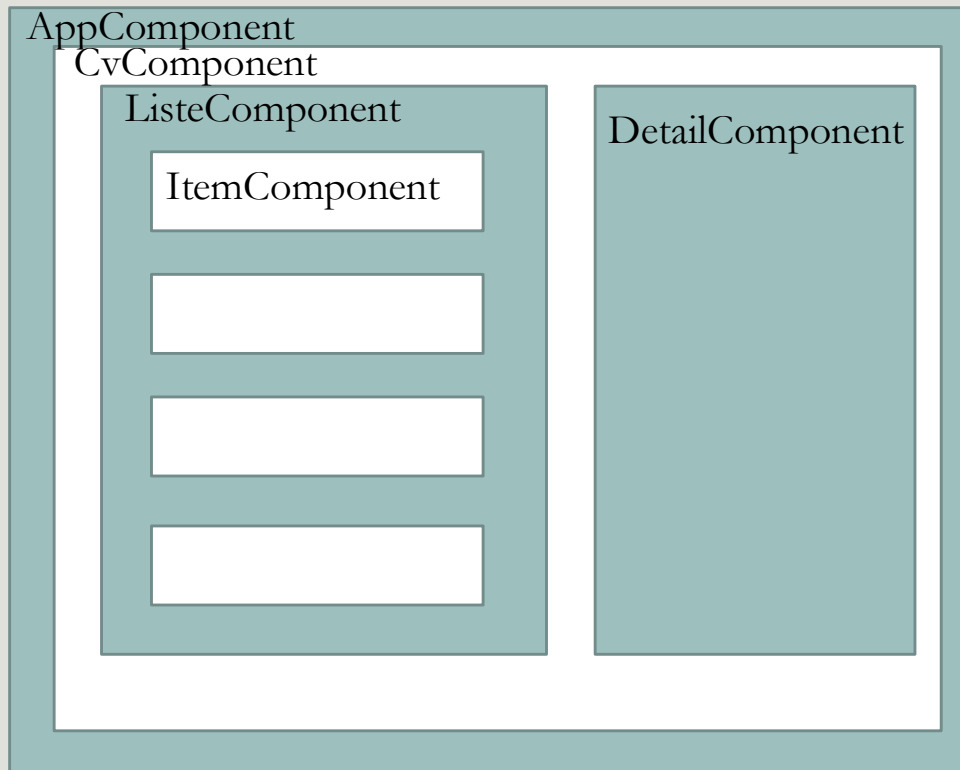
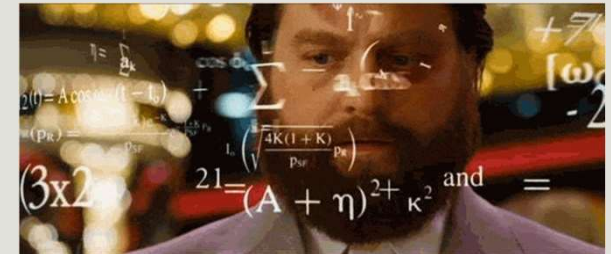
# Exercice

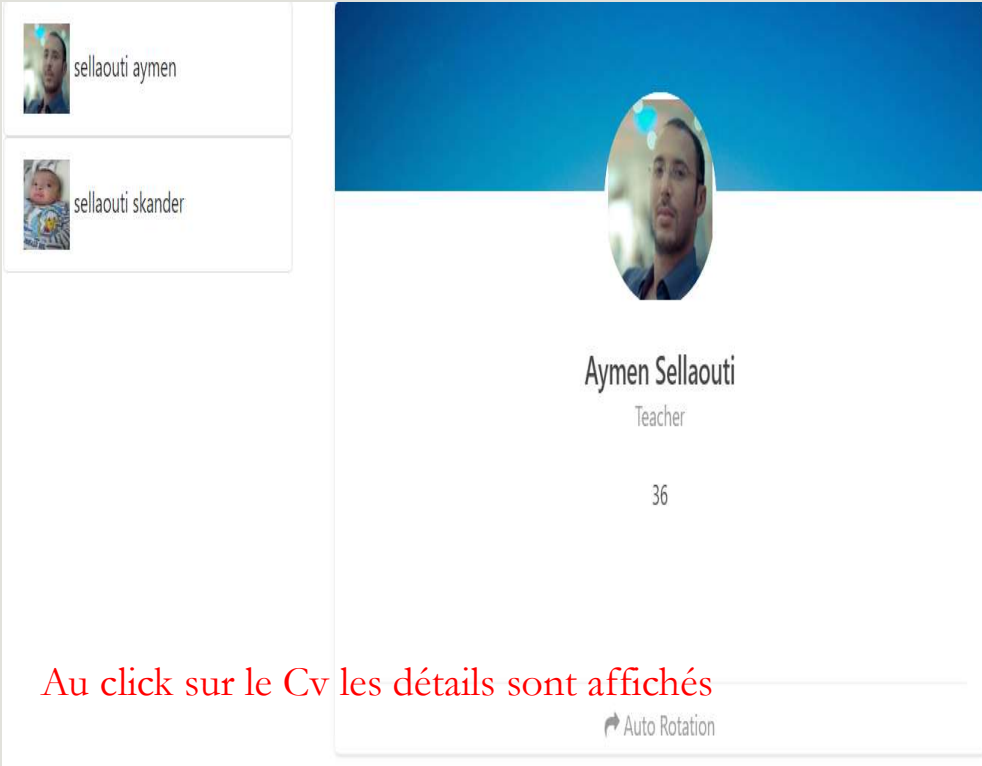


- Le but de cet exercice est de créer une mini plateforme de recrutement.
  - La première étape est de créer la structure suivante avec une vue contenant deux parties :
    - Liste des Cvs inscrits
    - Détail du Cv qui apparaîtra au click
  - Il vous est demandé juste d'afficher un seul Cv et de lui afficher les détails au click.
- Il faudra suivre cette architecture.

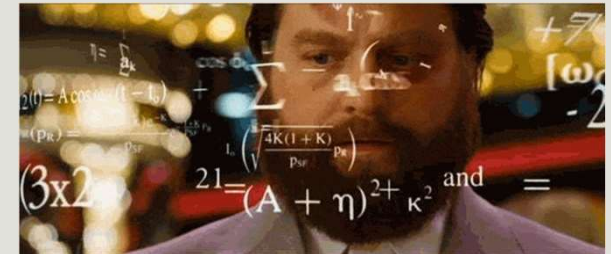


# Exercice





# Exercice



Un cv est caractérisé par :

id

name


firstname

Age


Cin

Job


path



sellaouti aymen



sellaouti skander



Aymen Sellaouti  
Teacher  
36

[Auto Rotation](#)

Au click sur le Cv les détails sont affichés



# Angular

# Les directives

---

AYMEN SELLAOUTI

# Objectifs

---

1. Comprendre la définition et l'intérêt des directives.
2. Voir quelques directives d'attributs offertes par angular et savoir les utiliser
3. Créer votre propre directive d'attributs
4. Voir quelques directives structurelles offertes par angular et savoir les utiliser

# Qu'est ce qu'une directive

- Une **directive** est une **classe** permettant d'**attacher** un **comportement** aux **éléments** du **DOM**. Elle est décorée avec l'annotation **@Directive**.
- Apparaît dans un élément comme un **tag** (comme le font les **attributs**).
- La command pour créer une directive est  
➤ **ng g d nomDirective**



```
import {Directive, HostBinding, HostListener}
from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @HostBinding('style.backgroundColor') bg = '';
  constructor() { }
  @HostListener('mouseenter') mouseenter() {
    this.bg = 'yellow';
  }
  @HostListener('mouseleave') mouseleave() {
    this.bg = 'red';
  }
}
```

```
<div appHighlight>
  Bonjour je teste une directive
</div>
```

# Qu'est ce qu'une directive

---

- La documentation officielle d'Angular identifie trois types de directives :
  - Les **composants** qui sont des directives avec des templates.
  - Les **directives structurelles** qui changent **l'apparence** du **DOM** en ajoutant et supprimant des éléments.
  - Les **directives d'attributs** (attribute directives) qui permettent de changer **l'apparence** ou le **comportement** d'un **élément**.

# Les directives d'attribut (ngStyle)

---

- Cette directive permet de modifier **l'apparence** de **l'élément cible**.
- Elle est placée entre [ ] **[ngStyle]**
- Elle prend en paramètre un **attribut** représentant un **objet** décrivant le **style** à appliquer.
- Elle utilise le **property Binding**.

# Les directives d'attribut (ngStyle)

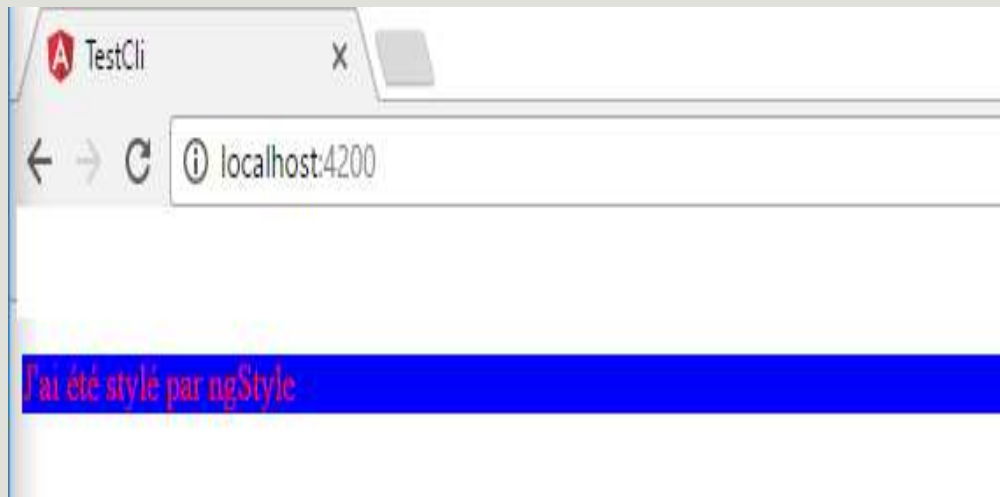
```
import { Component } from
 '@angular/core';

@Component ({
  selector: 'direct-direct',
  template: `
    <p [ngStyle]="{'color':'red',
                  'font-family':'garamond',
                  'background-color' : 'yellow'}">
    <ng-content></ng-content>
    </p>
  `,
  styleUrls: ['./direct.component.css']
})
export class DirectComponent {
}
```

```
import { Component } from
 '@angular/core';
@Component ({
  selector: 'direct-direct',
  template: `
    <p [ngStyle]="{'color':myColor,'font-
family':myfont,'background-color' :
myBackground}">
    <ng-content></ng-content>
    </p>`,
  styleUrls: ['./direct.component.css']
})
export class DirectComponent {
  private myfont:string="garamond";
  private myColor:string="red";
  private myBackground:string="blue"
}
```

# Les directives d'attribut (ngStyle)

---



# Les directives d'attribut (ngStyle)

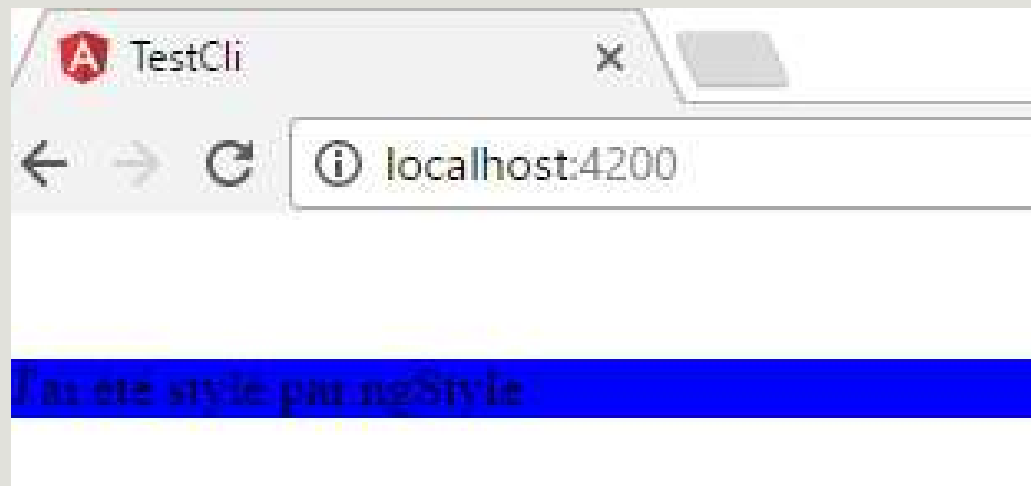
```
@Component ({
  selector: 'app-root',
  template: `
    <direct-direct [myColor]="gray">J'ai
été stylé par ngStyle</direct-direct>
  `,
  styles: [
    h1 { font-weight: normal; }
    p{color:yellow;background-color: red}
  ],
})
export class AppComponent {
}
```

```
import {Component, Input} from
'@angular/core';
@Component ({
  selector: 'direct-direct',
  template: `
    <p [ngStyle]="{'color':myColor,
'font-family':myfont,
'background-color' : myBackground}">
    <ng-content></ng-content>
  </p>
  `,
  styleUrls: ['./direct.component.css']
})
export class DirectComponent {
  private myfont:string="garamond";
  @Input() private myColor:string="red";
  private myBackground:string="blue"
}
```

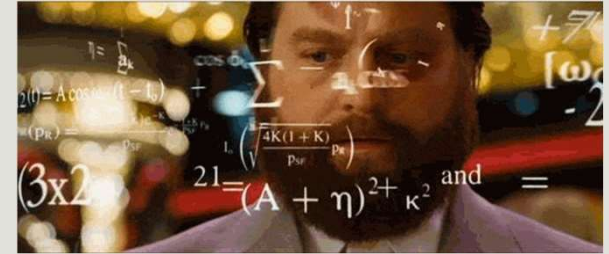


# Les directives d'attribut (ngStyle)

---



# Exercice



- Nous voulons simuler un Mini Word pour gérer un paragraphe en utilisant ngStyle.
- Préparer un input de type color, un input de type number, et un select box.
- Faite en sorte que lorsqu'on écrit une couleur dans le texte input, ca devienne la couleur du paragraphe. Et que lorsque on change le nombre dans le number input la taille de l'écriture.
- Finalement ajouter une liste et mettez y un ensemble de police. Lorsque le user sélectionne une police dans la liste, la police dans le paragraphe change.

Test

30

arial

# Les directives d'attribut (ngClass)

---

- Cette directive permet de modifier **l'attribut class**.
- Elle cohabite avec l'attribut **class**.
- Elle prend en paramètre
  - Une chaine (string)
  - Un tableau (dans ce cas il faut ajouter les [ ] donc [ngClass])
  - Un objet (dans ce cas il faut ajouter les [ ] donc [ngClass])
- Elle utilise le **property Binding**.

# Les directives d'attribut (ngClass)

```
import {Component, Input} from '@angular/core';
@Component({
  selector: 'direct-direct',
  template: `

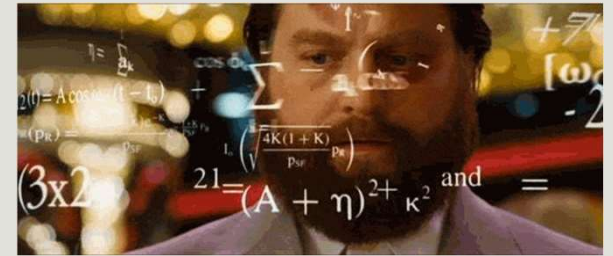
    <div ngClass="colorer arrierplan" class="encadrer">
      test ngClass
    </div>
  `,
  styles: [
    .encadrer{ border: inset 3px black; }
    .colorer{ color: blueviolet; }
    .arrierplan{background-color: salmon; }
  ]
})
export class DirectComponent{
  private myfont:string="garamond";
  @Input() private myColor:string="red";
  private myBackground:string="blue«
  private isColoree:boolean=true;
  private isArrierPlan:boolean=true
}
```

```
// Tableau
<div [ngClass]="['colorer', 'arrierplan'] "
class="encadrer">
// Objet

<div [ngClass]="{ colorer: isColoree,
arrierplan: isArrierPlan} "
class="encadrer">
```

# Exercice

---



- Préparer 3 classes présentant trois thèmes différents (couleur font-size et font-police)
- Au choix du thème votre cible changera automatiquement

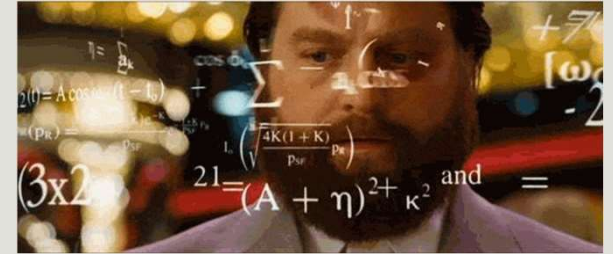
# Customiser un attribut directive

---

- Afin de créer sa propre « attribut directive » il faut utiliser un `HostBinding` sur la **propriété** que vous voulez **binder**.
  - **Exemple** : `@HostBinding('style.backgroundColor')`  
`bg:string="red";`
- 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') mouseover() {`  
`this.bg =this.highlightColor;`  
`}`
- Afin d'utiliser le `HostBinding` et le `HostListener` il faut les importer du `core d'angular`

# Exercice

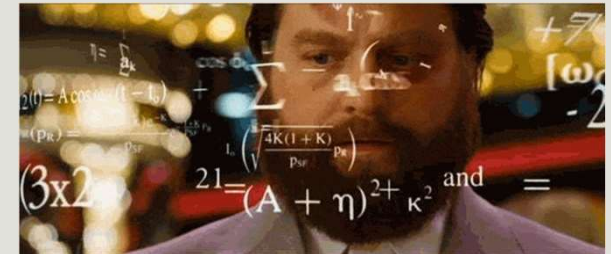
---

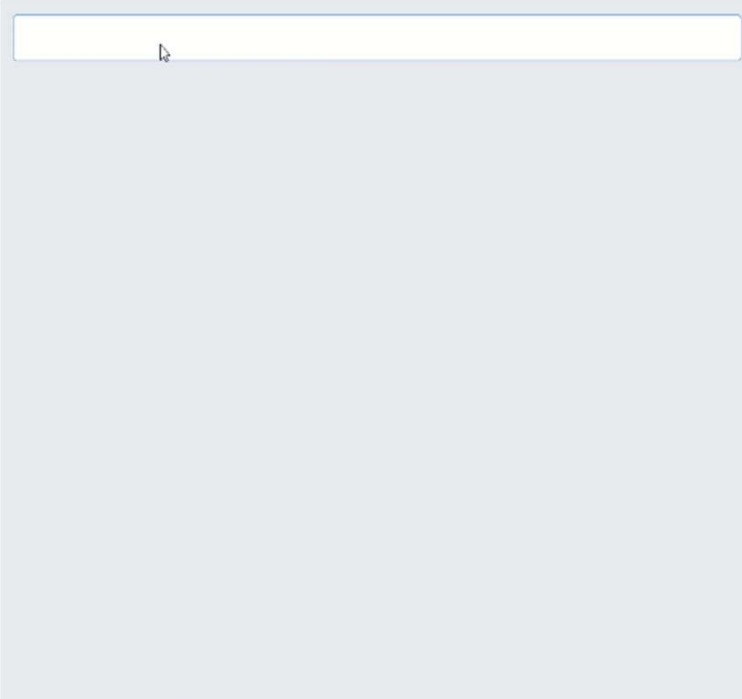


Un truc plus sympas on va créer un simulateur d'écriture arc en ciel.

- Créer une directive
- Créer un hostbinding sur la couleur et la couleur de la bordure.
- Créer un tableau de couleur dans votre directive.
- Faite en sorte qu'en appliquant votre directive à un input, à chaque écriture d'une lettre (event keyup) la couleur change en prenant aléatoirement l'une des couleurs de votre tableau. Pensez à utiliser `Math.random()` qui vous retourne une valeur entre 0 et 1.

# Exercise





```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <div class="container">
      <app-root _ngghost-pfp-c0 ng-version="8.2.14">
        <app-ng-style _ngcontent-pfp-c0 _ngghost-pfp-c1>
          ...
          <input _ngcontent-pfp-c1 apprainbow class=
            "form-control" style="border-color: rgb(125,
              162, 230); color: rgb(125, 162, 230);"> == $0
          </app-ng-style>
        </app-root>
      ...
    </div>
  </body>
</html>
```

div.container app-root app-ng-style input.form-control

Styles Computed Event Listeners DOM Breakpoints >>

Filter :hov .cls +

```
element.style {
  border-color: ▸ rgb(125, 162, 230);
  color: ▸ rgb(125, 162, 230);
}
```



# Customiser une attribut directive

---

- Nous pouvons aussi utiliser le `@Input` afin de rendre notre directive paramétrable
- Tous les paramètres de la directive peuvent être mises en `@Input` puis récupérer à partir de la cible.
- Exemple
  - Dans la directive `@Input ()` `private myColor:string="red";`
  - `<direct-direct [myColor]="gray">`

# Les directives structurelles

---

- Une **directive** structurelle permet de modifier le DOM.
- Elles sont appliquées sur **l'élément HOST**.
- Elles sont généralement précédées par le **préfix \***.
- Les directives les plus connues sont :
  - `*ngIf`
  - `*ngFor`

# Les directives structurelles \*ngIf

---

- Prend un booléen en paramètre.
- Si le booléen est true alors l'élément host est visible
- Si le booléen est false alors l'élément host est caché

Exemple

```
<p *ngIf="true">  
  Je suis visible :D</p>  
<p *ngIf="false">  
  Le *ngIf c'est fâché contre  
  moi et m'a caché :(  
</p>
```

# Les directives structurelles \*ngFor

➤ 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 courant

➤ first : vrai si premier élément

➤ last vrai si dernier élément

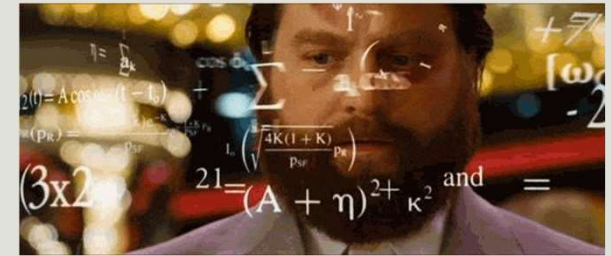
➤ even : vrai si l'indice est paire

➤ odd : vrai si l'indice est impaire

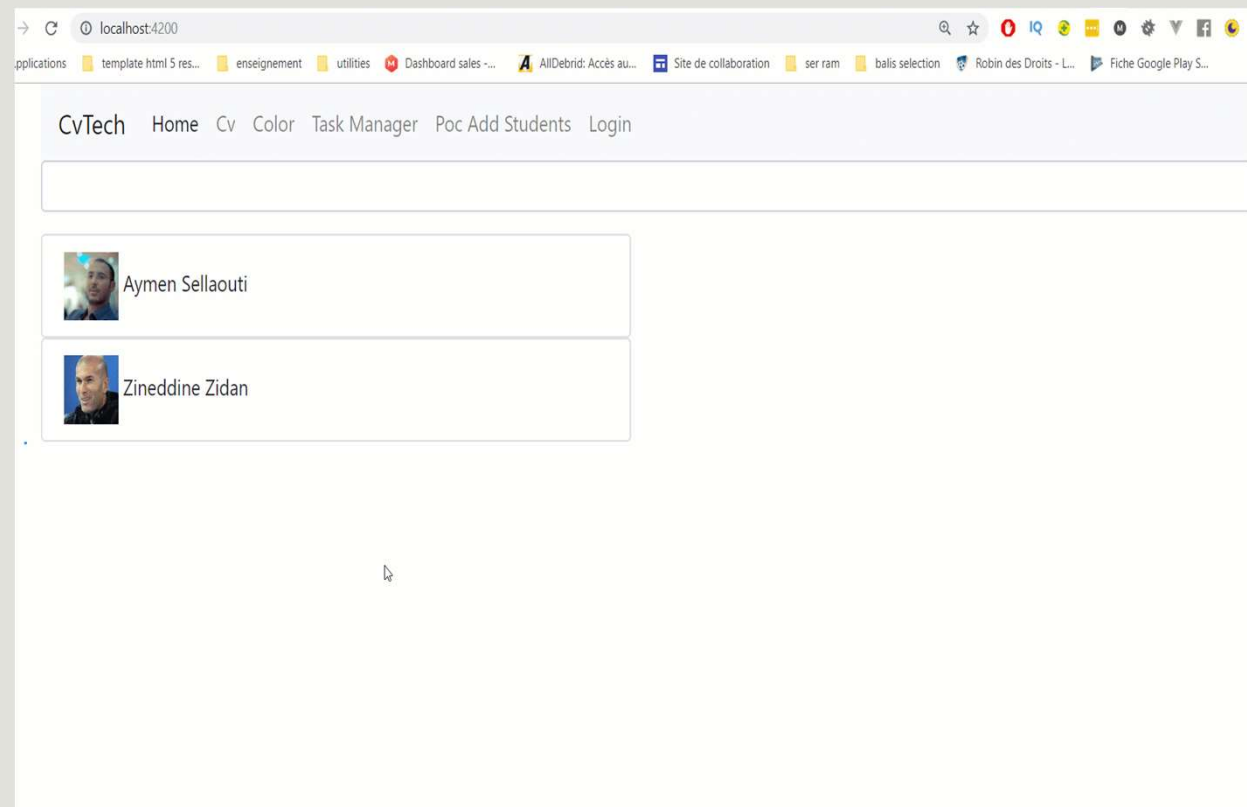
```
<ul>
  <li *ngFor="let episode of episodes">
    {{episode.title}}
  </li>
</ul>
```

```
<ul>
  <li *ngFor="let episode of episodes; let i = index;
    let isOdd = odd; let isFirst=first"
    [ngClass]="{ odd: isOdd , bgfonce: isFirst}"
  >
    Episode {{i+1}}{{episode.title}}
  </li>
</ul>
```

# Exercice (Notre Projet)



- Reprenons notre plateforme d'embauche.
- Utilisez les directives vues dans ce cours pour afficher une liste de Cv et pour améliorer l'affichage.
- Les détails ne sont affichés qu'au click sur un des cvs.



# Angular

# Les pipes

---

AYMEN SELLAOUTI

# Plan du Cours

---

1. Introduction
2. Les composants
3. Les directives
- 3 Bis. Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP

# Objectifs

---

1. Définir les pipes et l'intérêt de les utiliser
2. Vue globale des pipes prédéfinies
3. Créer un pipe personnalisé



# Qu'est ce qu'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.
- Il existe des pipes **offerts par Angular** et prêt à l'emploi.
- Vous pouvez créer vos **propres pipes**.

Avec le pipe uppercase :

Sans aucun pipe :

```
<input type="text" [(ngModel)]="pipeVar" class="form-control">  
<br>Avec le pipe uppercase : {{pipeVar|uppercase}}<br>  
Sans aucun pipe : {{pipeVar}}
```

# 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
- ...

# 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.
- Pout créer un pipe avec le cli : `ng g p nomPipe`

# Exemple de pipe

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

@Pipe({
  name: 'team'
})
export class TeamPipe implements PipeTransform {

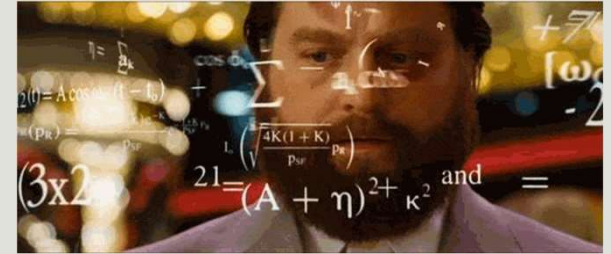
  transform(value: any, args?: any): any {
    switch (value) {
      case 'barca' : return ' blaugrana';
      case 'roma' : return ' giallorossa';
      case 'milan' : return ' rossoneri';
    }
  }
}
```

```
<li>
  <ol *ngFor="let team of
teams">
    {{team | team}}
  </ol>
</li>
```

```
ngOnInit() {
  this.teams = ['milan', 'barca', 'roma'];
}
```

# Exercice

---



Créer un pipe appelé `defaultImage` qui retourne le nom d'une image par défaut que vous stockerez dans vos assets au cas où la valeur fournie au pipe est une chaîne vide ou ne contient que des espaces.

# Angular Service et injection de dépendances

---

AYMEN SELLAOUTI





# Objectifs

---

1. Définir un service
2. Définir ce qu'est l'injection de dépendance
3. Injecter un service
4. Définir la portée d'un service
5. Réordonner son code en utilisant les services



# Qu'est ce qu'un service ?

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

Component 1

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

Component 2

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

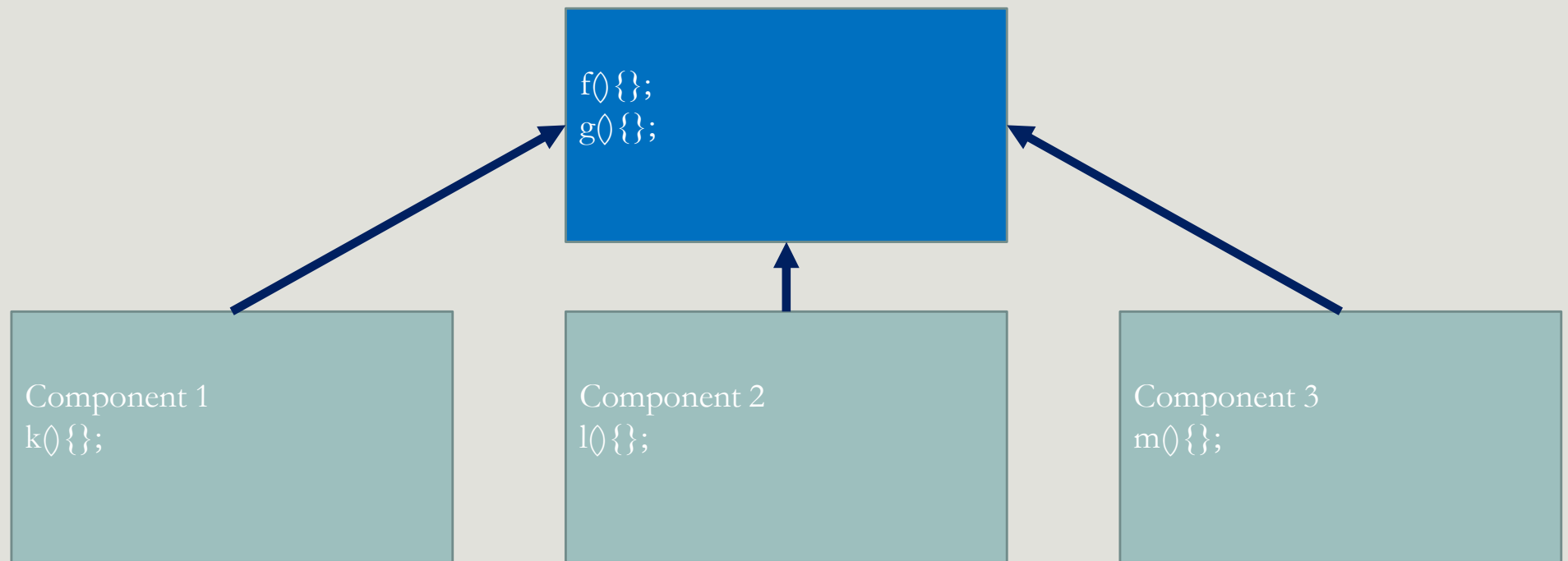
Component 3

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

**Redondance de code**

**Maintenabilité difficile**

# Qu'est ce qu'un service ?



# Qu'est ce qu'un service ?

---



- Un service peut :
  - Interagir avec les données (fournit, supprime et modifie)
  - Interaction entre classes et composants
  - Tout traitement métier (calcul, tri, extraction ...)

# Création d'un service

---

- Via CLI

- `ng generate service nomDuService`

- `ng g s nomDuService`

# Premier Service

---

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

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

  constructor() { }

}
```

# Injection de dépendance (DI)



- L'injection de dépendance est un patron de conception.

```
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 l'instanciation de ces classes dans les différentes classes qui en dépendent?

# Injection de dépendance (DI)



- Déléguer cette tâche à une entité tierce.

```
Classe A1 {  
  Constructor(B b, C c)  
  ...  
}
```

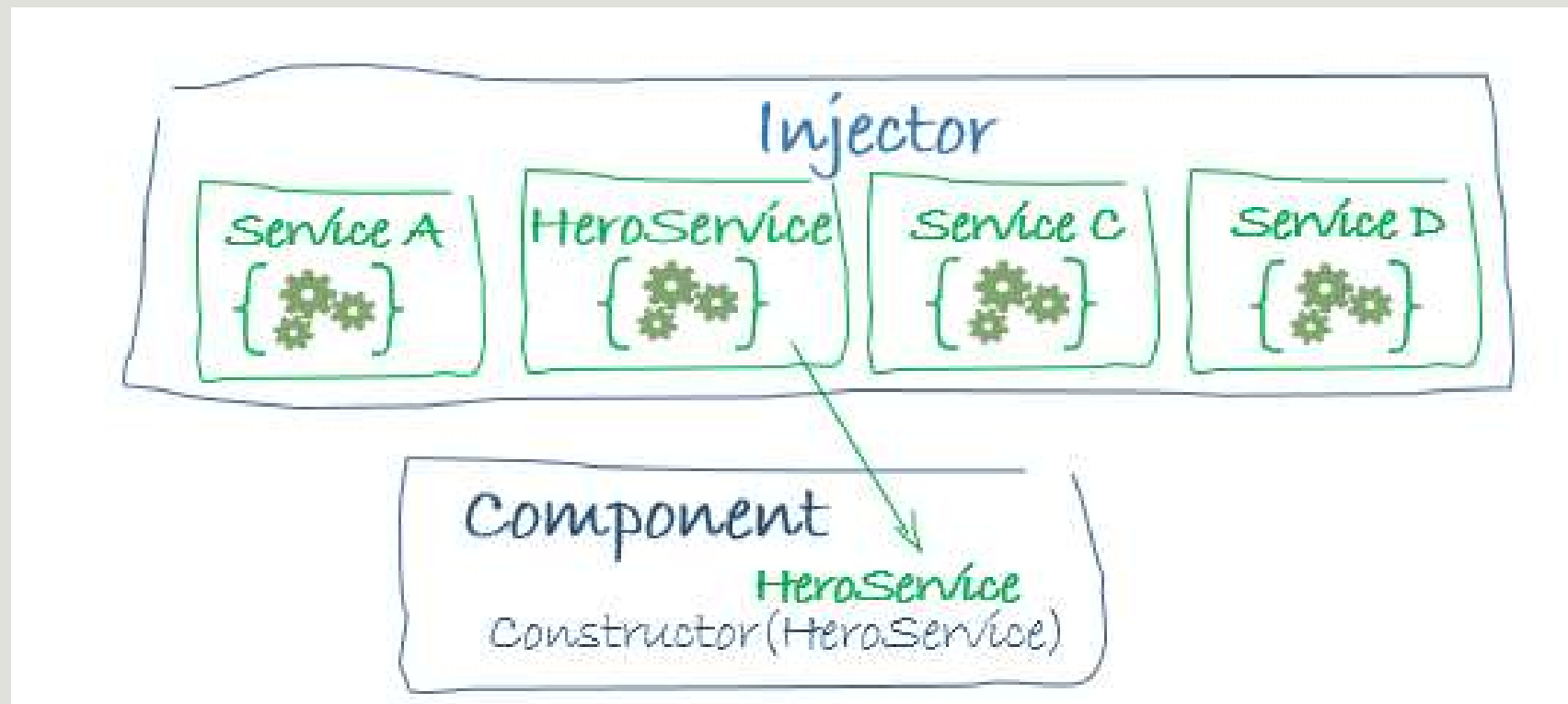
```
Classe A2 {  
  Constructor(B b)  
  ...  
}
```

```
Classe A3 {  
  Constructor(C c)  
  ...  
}
```

INJECTOR



# Injection de dépendance (DI)



# Injection de dépendance (DI)

---

- Comment les injecter ?
- Comment spécifier à l'injecteur quel service et où est-il visible ?

# Injection de dépendance (DI)

---

- L'injection de dépendance utilise les étapes suivantes :
  - Déclarer le service via l'annotation `@Injectable`, dans le provider du module **ou** du composant.
  - Passer le service comme paramètre du constructeur de l'entité qui en a besoin.

# Injection de dépendance (DI)

```
import { BrowserModule, } from '@angular/platform-browser';
import { CUSTOM_ELEMENTS_SCHEMA, NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
import { AppComponent } from './app.component';
import { CvService } from './cv.service';
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [CvService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

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

@Injectable()
export class CvService {

  constructor() { }

}
```

# Injection de dépendance (DI)

---

```
import { Component, OnInit } from '@angular/core';
import { Cv } from '../cv';
import { CvService } from "../cv.service";
@Component({
  selector: 'app-cv',
  templateUrl: './cv.component.html',
  styleUrls: ['./cv.component.css'],
  providers: [CvService] // on peut aussi l'importer ici
})
export class CvComponent implements OnInit {
  selectedCv : Cv;
  constructor(private monPremierService:CvService) { }
  ngOnInit() {
  }
}
```

# Chargement automatique du service

---

- 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 l'annotation `@Injectable` et sa propriété `providedIn`. 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'`.

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

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

# Avantage de l'utilisation du 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.

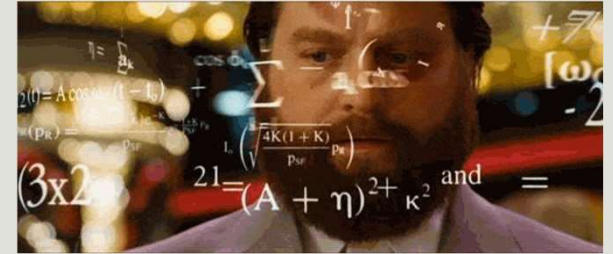
# @Injectable

---

- 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'aller injecter aucun service dans votre service, cette annotation n'est plus nécessaire.
- **Remarque** : Angular conseille de toujours mettre cette annotation.



# Exercice



- Créons un service de Todo, le Model et le composant qui va avec. Un Todo est caractérisé par un nom et un contenu.
- Ce service permettra de faire les fonctionnalités suivantes :
  - Logger les todos
  - Ajouter un Todo
  - Récupérer la liste des Todos
  - Supprimer un Todo

Name :  
I

Content :

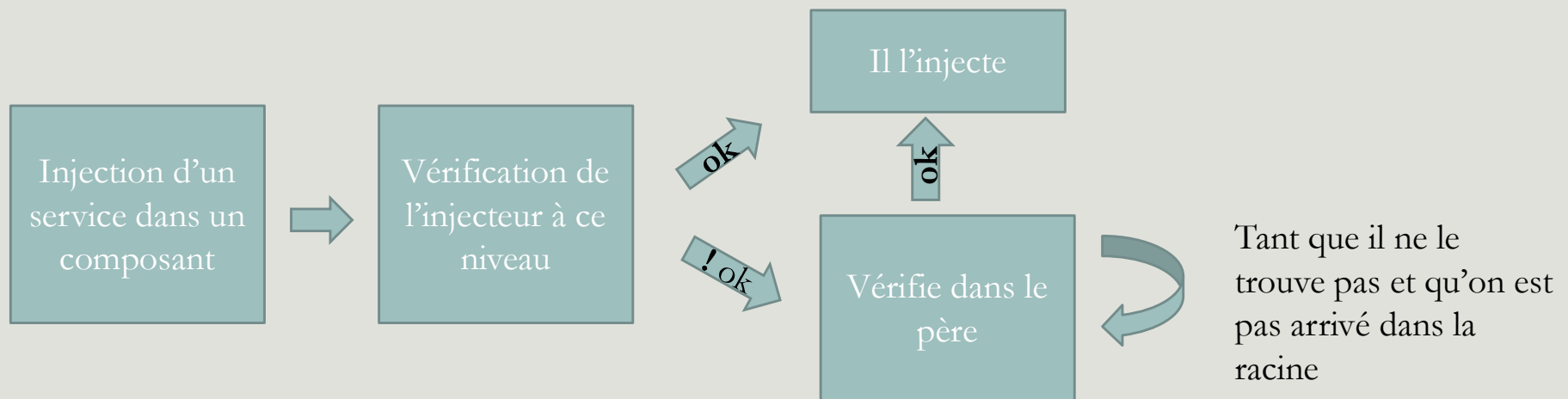
Add Todo

# Exemple

```
import { Injectable } from '@angular/core';
@Injectable()
export class LoggerService {
  constructor() { }
  Logs: string[]=[];
  logger(message:string) {
    this.Logs.push(message); console.log(message);
  }
  info(message:string) {
    this.Logs.push(message); console.info(message);
  }
  debugger(message:string) {
    this.Logs.push(message); console.debug(message);
  }
  avertir(message:string) {
    this.Logs.push(message); console.warn(message);
  }
  erreur(message:string) {
    this.Logs.push(message); console.error(message);
  }
}
```

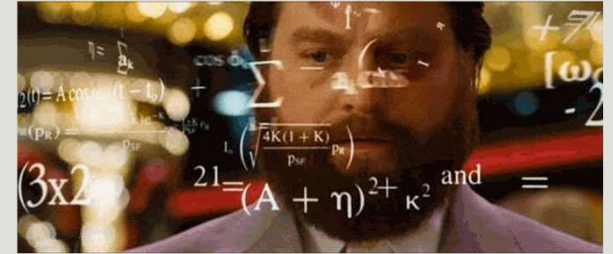
# DI Hiérarchique

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



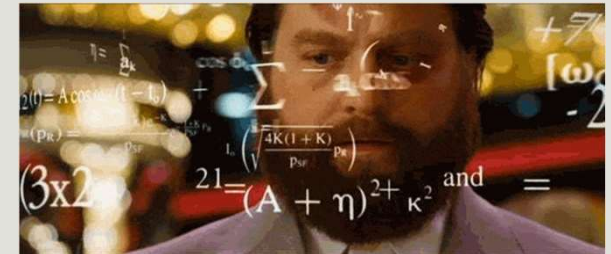
# Exercice

---



- Ajouter les services suivants afin d'améliorer la gestion de notre plateforme d'embauche.
  - Un premier **service CvService** qui gérera les Cvs. Pour le moment c'est lui qui contiendra la liste des cvs que nous avons.
  - Ajouter aussi un **composant** pour afficher la liste des cvs embauchées ainsi qu'un **service EmbaucheService** qui gérer les embauches.
  - Au click sur le bouton embaucher d'un Cv, le cv est ajoutés à la liste des personnes embauchées et une liste des embauchées apparait.

# Exercice



sellabouti aymen



sellabouti skander

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

12345678

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

235  
Followers

114  
Following

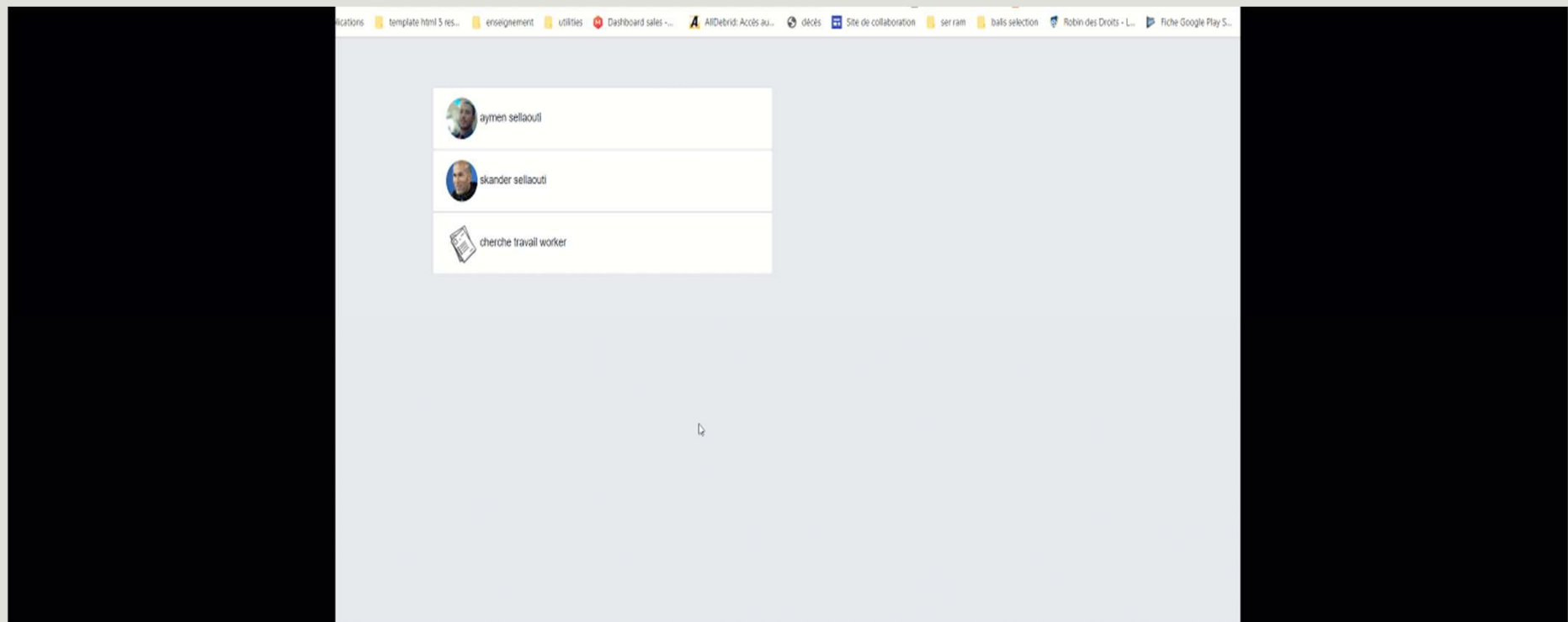
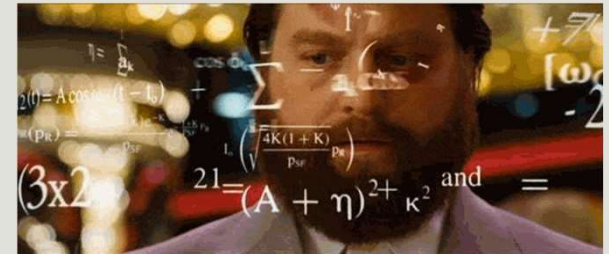
35  
Projects

Embaucher

## Liste des cvs sélectionnés pour embauche



# Exercice



# Angular Routing

---

AYMEN SELLAOUTI

# Objectifs

---

1. Définir le routeur d'Angular
2. Définir une route
3. Déclencher une route à partir d'un composant
4. Ajouter des paramètres à une route
5. Récupérer les paramètres d'une route à partir du composant.
6. Préfixer un ensemble de routes
7. Gérer les routes inexistantes



# Qu'est ce que le routing

---

- Tout système de routing permet d'associer une route à un traitement
- Angular SPA. Pourquoi parle-on de route ??
  - Séparer différentes fonctionnalités du système
  - Maintenir l'état de l'application
  - Ajouter des règles de protection
- Que risque t-on d'avoir si on n'utilise pas un système de routing ?
  - On ne peut plus rafraichir notre page
  - Plus de Favoris ☹
  - Comment partager vos 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' Importer le service de routing d'Angular
  - `import { RouterModule, Routes } from '@angular/router';`
  - Le `RouterModule` va permettre de configurer les routes dans votre projet
  - Le `Routes` va permettre de créer les routes

# Création d'un système de Routing

1

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Cv</title>
  <base href="/">

  <meta name="viewport" content="width=device-
width, initial-scale=1">
  <link rel="icon" type="image/x-icon"
href="favicon.ico">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap
/3.3.7/css/bootstrap.min.css"
integrity="sha384-
BVYiISIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+P
mSTsz/K68vbdEjh4u"
crossorigin="anonymous"></head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

2

```
import {Routes, RouterModule} from
"@angular/router";
import {CvComponent} from "./cv/cv.component";
import {HeaderComponent} from
"./header.component";
```

App.routing.ts

# Création d'un système de Routing

---

3. Créer la constante qui est un tableau d'objet de type `Routes` représentant chacun la route à décrire.
4. Intégrer les routes à notre application dans le app module à travers le `RouterModule` et sa méthode `forRoot`

# Création d'un système de Routing

```
import {Route, RouterModule} from
"@angular/router";
import {CvComponent} from "./cv/cv.component";
import {HeaderComponent} from
"./header.component";

const APP_ROUTES : Routes = [
  {path: '', component:CvComponent},
  {path:'onlyHeader', component:HeaderComponent}
];

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

App.routing.ts

```
import { BrowserModule, } from
'@angular/platform-browser';
import {CUSTOM_ELEMENTS_SCHEMA, NgModule} from
'@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
import { AppComponent } from './app.component';
import {routing} from './app.routing';
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    ROUTING
  ],
  providers: [CvService,EmbaucheService],
  schemas: [CUSTOM_ELEMENTS_SCHEMA],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

# Préparer l'emplacement d'affichage des vues correspondantes aux routes

---

- 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>`

# Préparer l'emplacement d'affichage des vues correspondantes aux routes

---

```
<as-header></as-header>  
  
<div class="container">  
  <router-outlet></router-outlet>  
</div>
```

# Syntaxe minimaliste d'une route

---

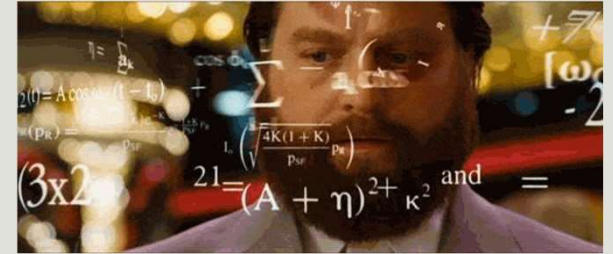
- Une route est un objet.
- Les deux propriétés essentielles sont `path` et `component`.
- `component` permet de spécifier le composant à exécuter.

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



# Exercice

---



- Configurer votre routing
- Créer deux composants
- Créer deux routes qui pointent sur ces deux composants
- Vérifier le fonctionnement de votre routing

# Déclencher une route routerLink

---

- L'idée intuitive pour déclencher une route est d'utiliser la balise **a** et son attribut **href**. Est-ce que ça risque de poser un problème ?
- 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 **directive routerLink** qui comme son nom l'indique liera la directive à la route que nous souhaitons déclencher **sans recharger la page**.
- Exemple :

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

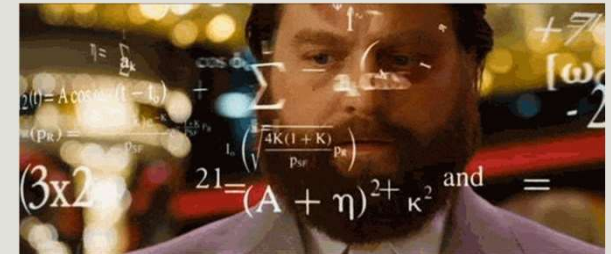
# Déclencher une route routerLink

---

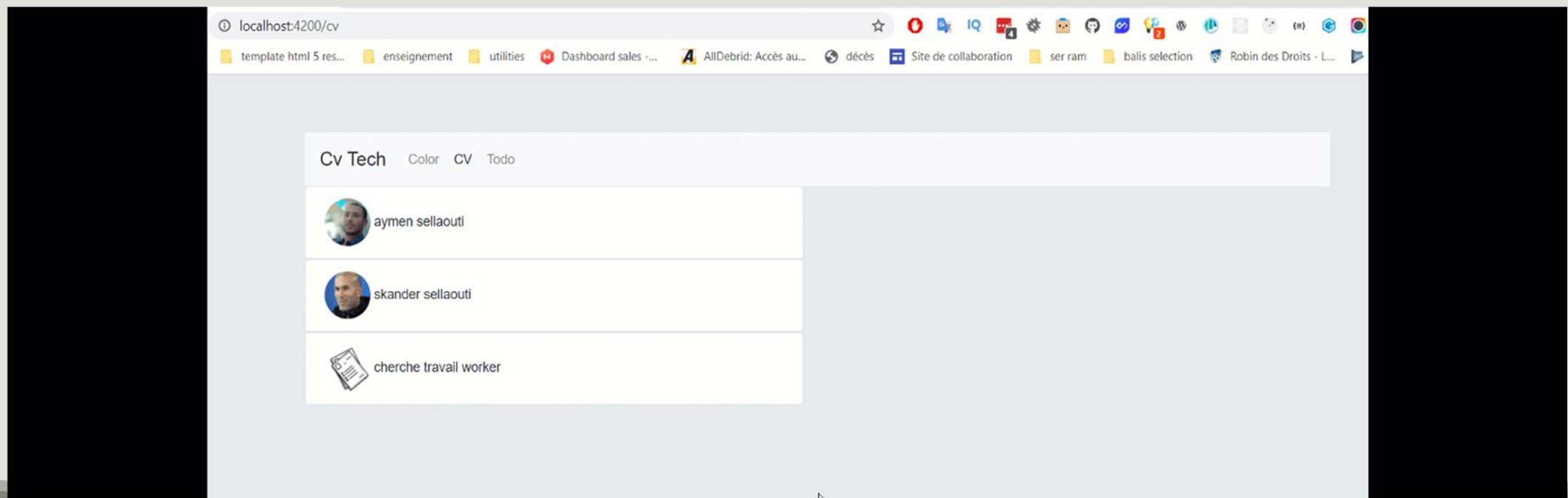
- **routerLinkActive="active"** va associer la classe active à l'uri cible ainsi qu'à tous ses ancêtres.
- Par exemple si on a l'uri 'cv/liste' la classe active sera ajoutée à cet uri ainsi qu'à l'uri 'cv' et ''.
- Pour identifier uniquement l'uri cible, ajouter la directive suivante :

[routerLinkActiveOptions]="{exact: true}"

# Exercice



- Faites en sorte d'avoir un composant Header dans votre application qui permet d'afficher l'ensemble de vos liens.
- En cliquant sur un lien, le composant qui lui est associé doit être affiché.



# Déclencher une route à partir du composant

---

- Afin de déclencher une route à travers le composant on utilise le service **Router** et sa méthode **navigate**.
- 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 l'**@angular/router** et l'injecter dans votre composant.

# Déclencher une route à partir du composant

---

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent {
  constructor(private router: Router) { }
  onNaviger() {
    this.router.navigate(['/about/10']);
  }
}
```

# 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
  - /cv/:id permet de dire que la root contient au début cv ensuite un paramètre de root appelé id.

# Récupérer les paramètres d'une route

---

- 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.
  3. Utilisez l'objet **snapshot**



# Récupérer les paramètres d'une route

## ActivatedRoute / snapshot

---

- La propriété **snapshot** de l'objet **ActivatedRoute** contient un instantané de l'état de la route actuelle.
- Elle contient des **informations** telles que **l'URL actuelle**, **les paramètres de route actuels**,...
- Elle est généralement utilisée pour **accéder aux informations de route** dans un composant **lors de son initialisation**.
- Il est important de noter que **l'instantané de la route ne change pas lorsque la route change**. Il représente un **état figé** de la route lors de son instanciation.

# Récupérer les paramètres d'une route

## ActivatedRoute / snapshot

---

- Voici quelques propriétés courantes de l'API snapshot :
  - **url**: Retourne l'URL de la route actuelle sous forme de tableau de segments d'URL.
  - **params**: Retourne un objet qui contient les paramètres de route actuels.
  - **queryParams**: Retourne un objet qui contient les paramètres de requête actuels.
  - **fragment**: Retourne la partie de l'URL après le symbole "#".
  - **data**: Retourne les données de route associées à la route actuelle.
  - **component**: Retourne le composant de route actuel.
  - **routeConfig**: Retourne la configuration de la route actuelle.

# Récupérer les paramètres d'une route

## ActivatedRoute / snapshot

```
▼ snapshot: ActivatedRouteSnapshot
  ▶ component: class DetailsCvComponent
  ▶ data: {cv: {...}}
    fragment: null
    outlet: "primary"
  ▶ params: {id: '27'}
  ▶ queryParams: {}
  ▼ routeConfig:
    ▶ component: class DetailsCvComponent
    path: ":id"
    ▶ resolve: {cv: f}
    ▶ [[Prototype]]: Object
  ▶ url: [UrlSegment]
    _lastPathIndex: 1
  ▶ _paramMap: ParamsAsMap {params: {...}}
  ▶ _resolve: {cv: f}
  ▶ _resolvedData: {cv: {...}}
  ▶ _routerState: RouterStateSnapshot {_root: TreeNode, url: '/cv/2'}
  ▶ _urlSegment: UrlSegmentGroup {segments: Array(2), children: {...}}
  ▶ children: Array(0)
    firstChild: null
  ▼ paramMap: ParamsAsMap
    ▶ params: {id: '27'}
    keys: (...)
    ▶ [[Prototype]]: Object
    parent: (...)
```

# Récupérer les paramètres d'une route

## ActivatedRoute / snapshot

---

- Donc pour accéder à votre propriété, passez par l'objet `snapshot`
- Avec `snapshot`, vous avez deux méthodes pour récupérer les paramètres:
- Via la **propriété `params`** qui retourne un tableau d'objet des paramètres
- Via la propriété **`paramMap`**
  - Appeler sa méthode **`get`**
  - Passez lui le nom de la propriété souhaitée.

```
this.activatedRoute.snapshot.params['id']
```

```
this.activatedRoute.snapshot.paramMap.get('id')
```

# Passer le paramètre à travers le tableau de routerLink

---

- Une autre méthode permet de passer le paramètre de la route est en l'ajoutant comme un autre attribut du tableau associé au routerLink

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent {
  constructor(private router: Router) { }
  id:number=10;
  onNaviger() {this.router.navigate(['/about', this.id]);}
}
```

# Les queryParameters

---

- Les **queryParameters** sont les paramètres envoyés à travers une requête **GET**.
- Identifié avec le **?**.
- Afin d'insérer un **queryParameters** on dispose de deux méthodes
- On ajoute dans la méthode **navigate** du **Router** un **second paramètre de type objet**.
- L'une des propriétés de cet objet est aussi un **objet** dont la **clé** est **queryParams** dont le contenu est aussi un **objet** contenant les identifiants des queryParams et leurs valeurs.

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

# Les queryParameters

---

- La deuxième méthode est en l'intégrant 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. Soit d'une façon statique avec **snapshot via la propriété queryParams** ou sa propriété **queryParamMap** et sa méthode **get**.
- Soit dynamiquement via l'observable **queryParams**

```
this.activatedRoute.snapshot.queryParams['page']
```

```
this.activatedRoute.snapshot.queryParamMap.get('page')
```



# La route joker

---

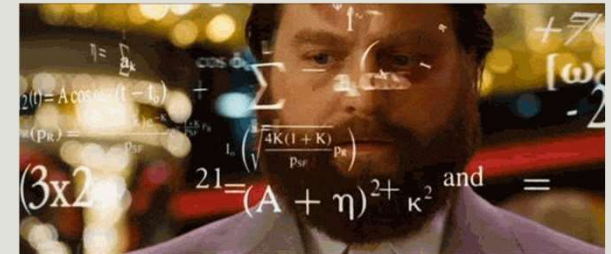
- Il existe une route **joker** qui **matche n'importe quelle autre route**.  
C'est la route **'\*\*'**.

# Exemple

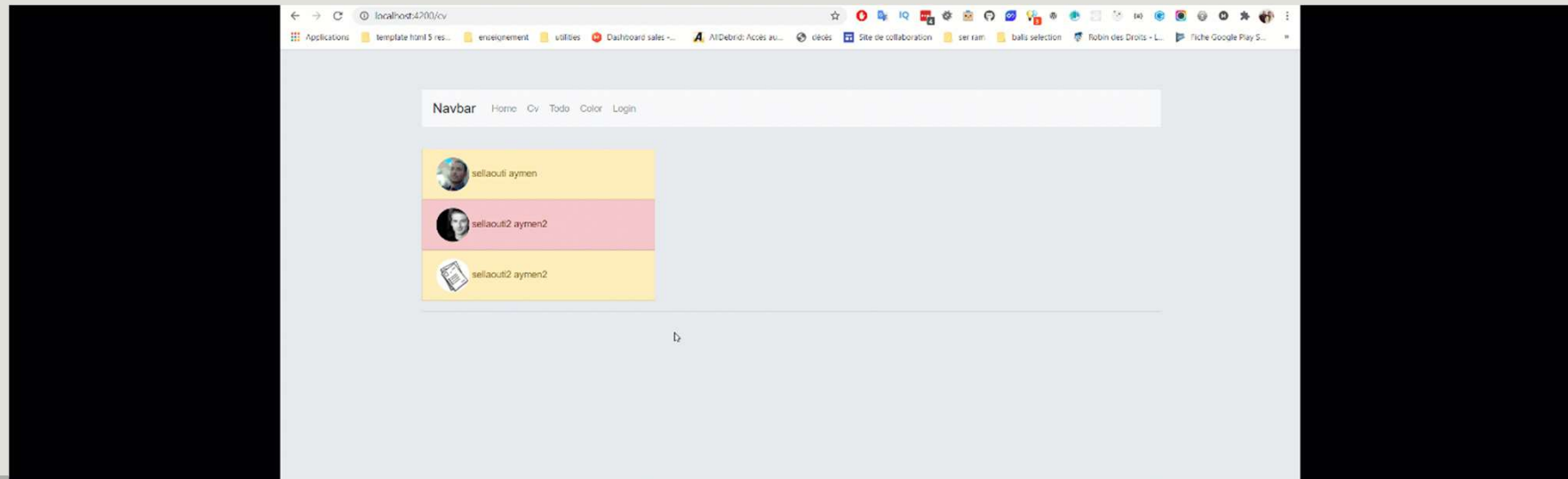
---

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

# Exercice



- Ajouter les fonctionnalités suivantes à votre cvTech:
  - Une page détail qui va afficher les détails d'un cv.
  - Un bouton dans chaque cv qui au click vous envoie vers la page détails.
  - Dans la page détail, un bouton delete qui au click supprime ce cv et vous renvoie à la liste des cvs.



# Angular Form

---

AYMEN SELLAOUTI

# Approche de gestion de FORM

---

1. Approche basée Template
2. Approche réactive

# Objetctifs

---

1. Créer un formulaire
2. Ajouter des validateurs
3. Appréhender les classes Css générées par le formulaire
4. Manipuler l'objet ngForm
5. Manipuler les controles du formulaire

# Approche basée Template/ Template Driven Approach

---

- 1 Importer le module FormsModule dans app.module.ts
- 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 ajouter la directive angular **ngModel**.
  - 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 et la passer à votre fonction en utilisant le référencement interne # et la directive **ngForm**

```
<input
  type="text"
  id="username"
  class="form-
control"
  ngModel
  name="username"
>
```

# Approche basée Template/ Template Driven Approach

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

Template

```
export class
TmeplateDrivenComponent{
  onSubmit(formulaire:
NgForm) {

console.log(formulaire);
  }
}
```

Component.ts



# Approche basée Template Validation

---

Afin de valider les propriétés des différents contrôles, Angular utilise des attributs et des directives

- required
- email

La propriété `valid` de `ngForm` permet de vérifier si le formulaire est valide ou non en se basant sur les validateurs qu'ils contiennent.

# Approche basée Template

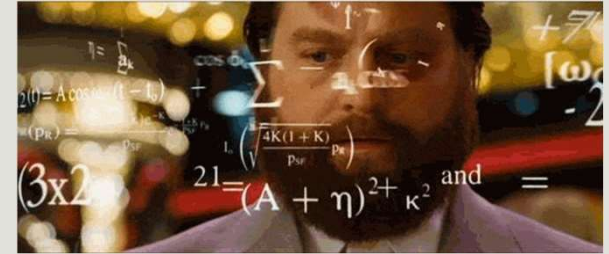
## NgForm

---

En détectant le formulaire, Angular décore les différents éléments du formulaire avec des classes qui informe sur leur état :

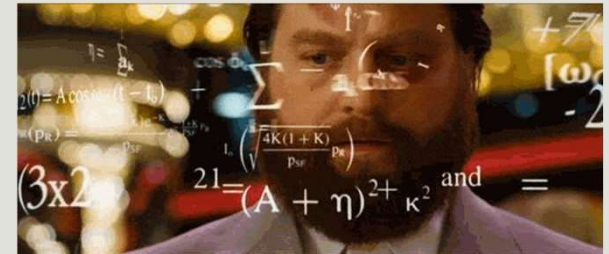
- **dirty** : informe sur le fait que l'une des propriétés du formulaire a été modifié ou non
- **Valid / invalid** : informe si le formulaire est valide ou non
- **Untouched / touched** : informe si le formulaire est touché ou non
- **pristine** : le formulaire n'a pas été touché, c'est l'opposé du dirty

# Exercice



- Créer un formulaire d'authentification contenant les champs suivants :
  - Email
  - Password
  - Envoyer
- Si un champ est invalide alors il devra avoir une bordure rouge.
- Les deux champs sont obligatoires
- Le password doit avoir au moins 4 caractères.
- 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 *disabled*.

# Exercice



← → ↻ localhost:4200/login

Applications template.html 5 res... enoionement utilites Dashboard sales -... AllDebrid: Accès au... clés Site de collaboration ser ram batis selection Robin des Droits - L... Fiche Google Play S...

Navbar Home Cv Todo Color Login

Email :

password :

Login

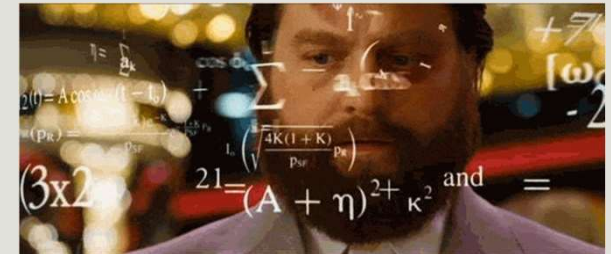
# Approche basée Template

## Accéder aux propriétés d'un champ (contrôle) du formulaire

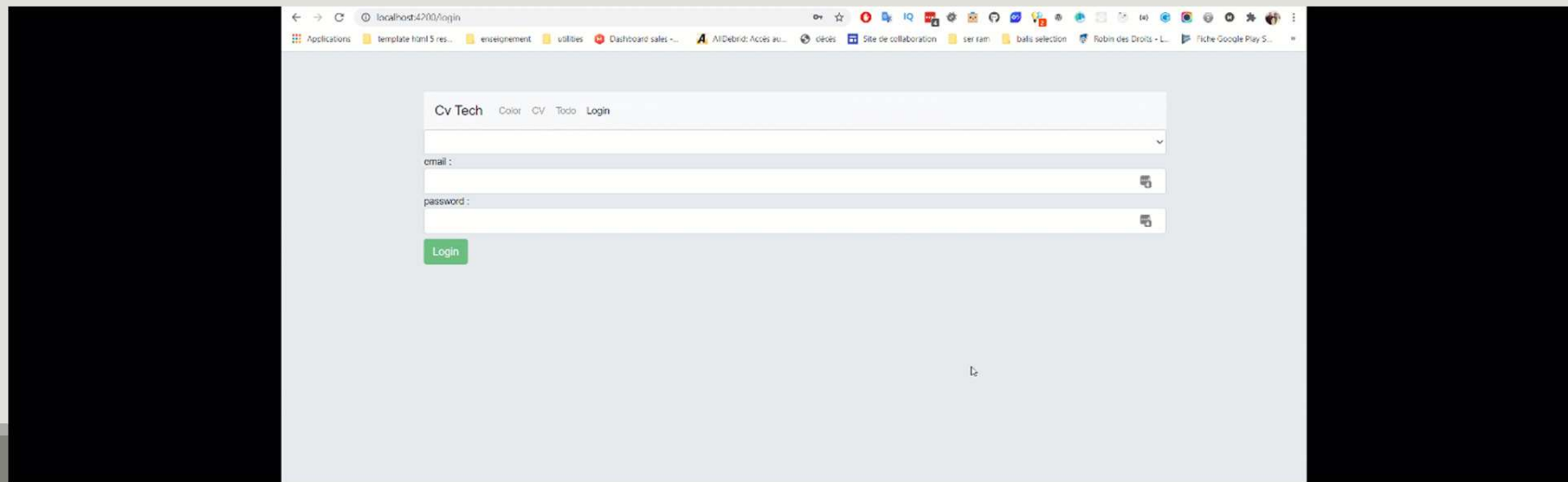
---

- Pour accéder à l'objet form et ces propriétés nous avons utilisé `#notreForm=«ngForm »`
- Pour les champs du formulaire c'est la même chose mais au lieu du ngForm c'est un `ngModel`  
`#notreChamp=« ngModel »`

# Exercice



- Ajouter un petit message d'erreur qui devra s'afficher sous le champs de l'email s'il est invalide. Ce champ ne devra apparaitre que si l'utilisateur accède ou modifie le champ email.
- Ajouter un champ d'erreur pour signaler l'erreur à l'utilisateur.



# Approche basée Template

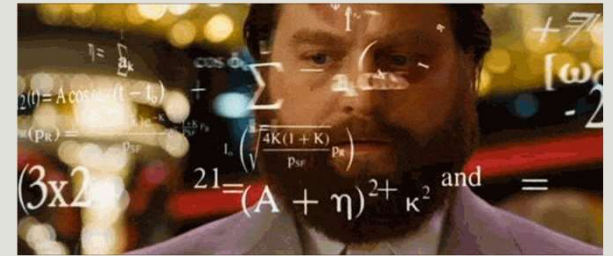
## Associer des valeurs par défaut aux champs

---

- Pour associer des valeurs par défaut aux champs d'un formulaire associé à Angular il faut le faire à partir du composant.
- Afin de gérer les valeur du formulaire à partir du composant il faut du binding.
- Au lieu d'avoir juste la primitive ngModel associée au contrôle d'un élément on ajoute le **property binding** avec **[ngModel]**

# Exercice

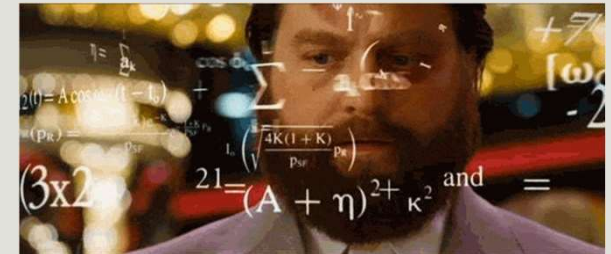
---



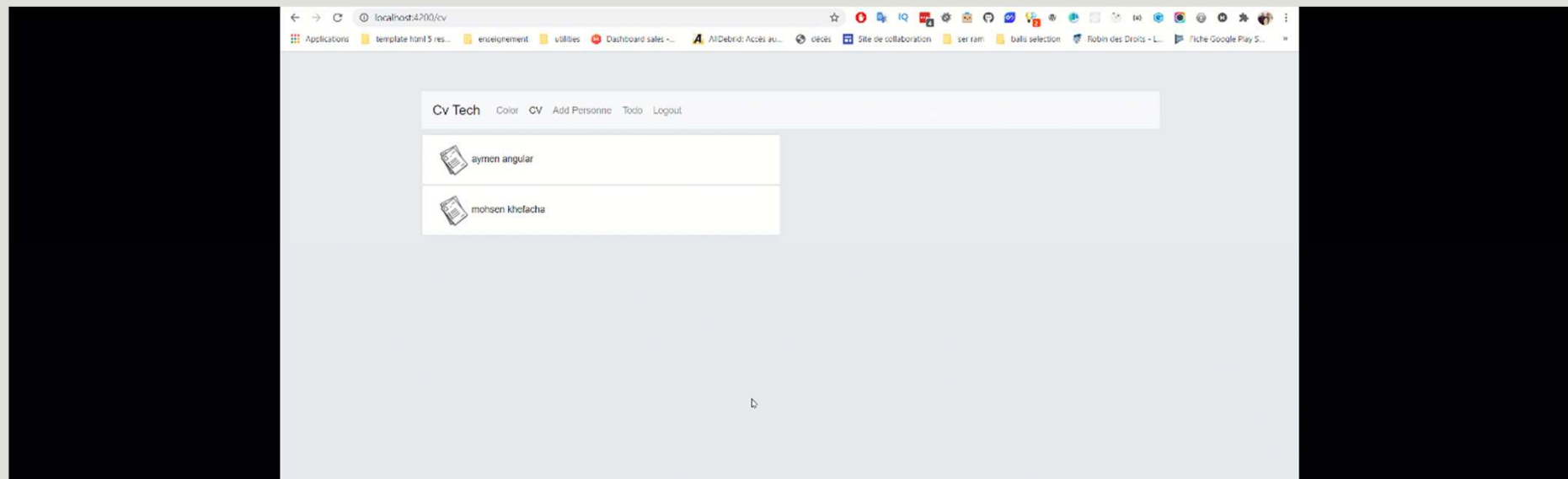
- Ajouter la valeur par défaut « myUserName » au champ username.



# Exercice



- Ajouter dans votre cvTech un composant contenant un formulaire. Ce formulaire devra vous permettre d'ajouter un utilisateur.
- Après l'ajout forwarder le user vers la liste des cvs.



# Angular HTTP et Déploiement

---

AYMEN SELLAOUTI

# Objectifs

---

1. Comprendre le design pattern Observable et son implémentation avec RxJs
2. Appréhender le Module HTTPClientModule d'Angular
3. Utiliser les différents services du module HTTPClientModule
4. Comprendre le principe d'authentification via les tokens
5. Utiliser les protecteurs de routes (les guards)
6. Utiliser les Interceptors afin d'intercepter les requêtes Http
7. Déployer votre application en production

# HTTP

---

- 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.

Le Module HTTPClient

# Programmation Asynchrone

---

Programmation non bloquante.

# Les promesses

---

- Ce sont des objets qui représentent une complétion ou l'échec d'une opération asynchrone.

([https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Utiliser\\_les\\_promesses](https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Utiliser_les_promesses))

- Le fonctionnement des promesses est le suivant :
  - On crée 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

# Promesse

---

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

promise2.then(
  function (x) {
    console.log('resolved with value :', x);
  }
)
```

# Qu'est ce que la programmation réactive

---

1. Nouvelle manière d'appréhender les appels asynchrones
2. Programmation avec des flux de données asynchrones

Programmation reactive =  
Flux de données (observable) + écouteurs d'événements(observer).



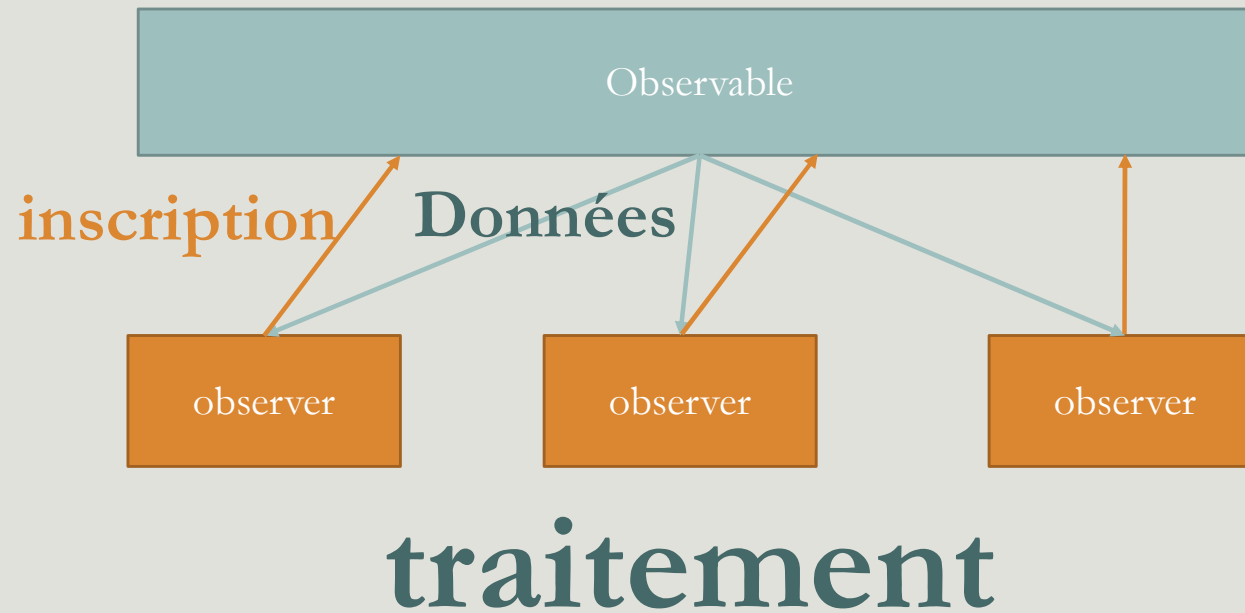
# Le pattern « Observer »

---

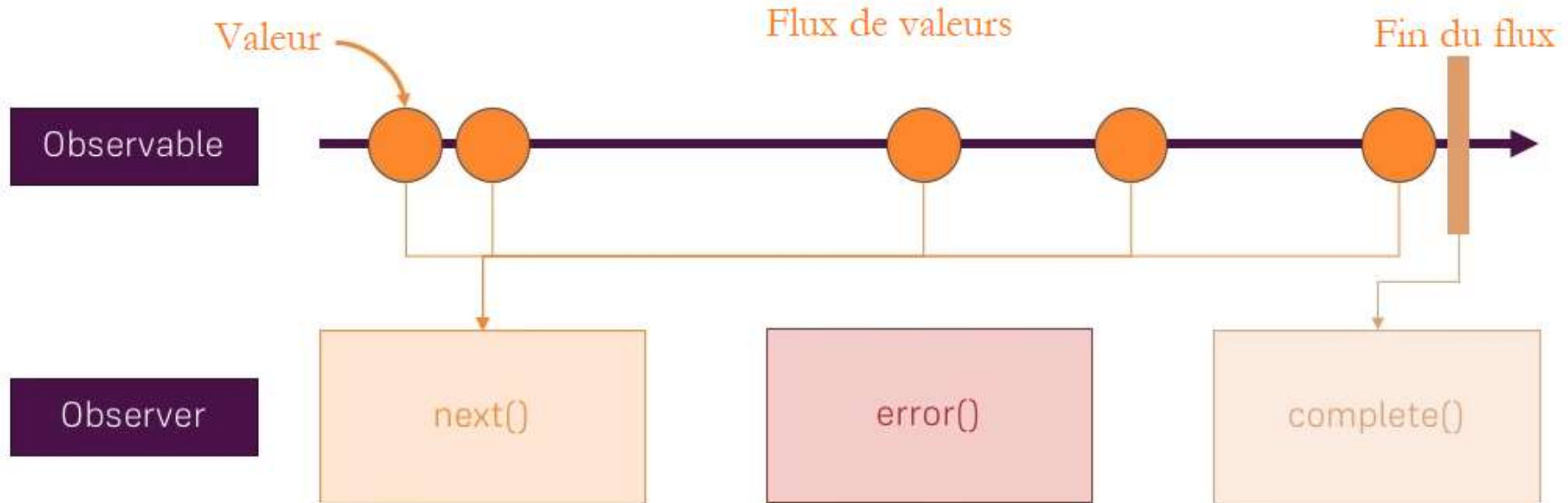
- Le patron de conception **Observable** permet à un objet de garder la trace d'autres objets, intéressés par l'état de ce dernier.
- Il définit une relation entre objets de type un-à-plusieurs.
- Lorsque l'état de cet objet **change**, il **notifie** ces **observateurs**.

# Observables, Observers et subscriptions

---



# Fonctionnement



# Promesse Vs Observable

Promesse	Observable
Un promesse gère un seul événement	Un observable gère un « flux » d'événements.
Non annulable.	Annulable.
Traitement immédiat.	Lazy : le traitement n'est déclenché qu'à la première utilisation du résultat.
Deux méthodes uniquement (then/catch).	Une centaine d'opérateurs de transformation natifs (map, reduce, merge, filter, ...).
	Opérateurs tels que retry, replay

# Observable

---

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

# asyncPipe

---

- asyncPipe est un pipe qui permet d'afficher directement un observable.
- `{{ valeurSourceAsynchrone | async }}`
- L'asyncPipe s'inscrit automatiquement à l'observable et affiche le dernier résultat envoyé.
- Quand le composant est détruit l'asyncPipe se désinscrit automatiquement de l'observable.

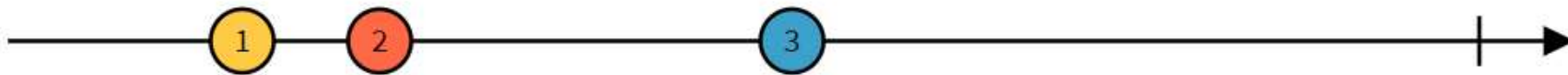
# Les operateurs de l'observable

---

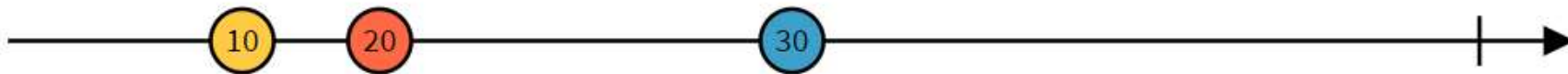
- Les opérateurs sont des fonctions. Il y a deux types d'opérateurs :
- **Un opérateur pipeable** est une fonction qui prend un observable comme entrée et renvoie un autre observable. C'est une opération pure : le précédent Observable reste inchangé.
  - Syntaxe : `monObservable.pipe(opertaeur1(), operateur2(), ...)`.
- **Les opérateurs de création** sont l'autre type d'opérateur, qui peut être appelé comme fonctions autonomes pour créer un nouvel Observable. Par exemple : `of(1, 2, 3)` crée un observable qui va émettre 1, 2, et 3, l'un après l'autre.

# Quelques opérateurs utiles de l'Observable

## map



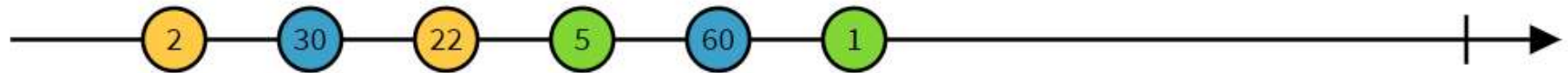
`map(x => 10 * x)`





# Quelques opérateurs utiles de l'Observable

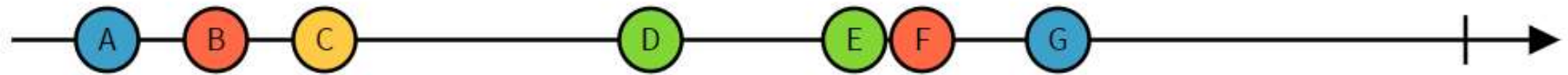
## filter



`filter(x => x > 10)`



# Quelques opérateurs utiles de l'Observable



`throttleTime(25)`



# Quelques opérateurs utiles de l'Observable

---

<https://angular.io/guide/rx-library>

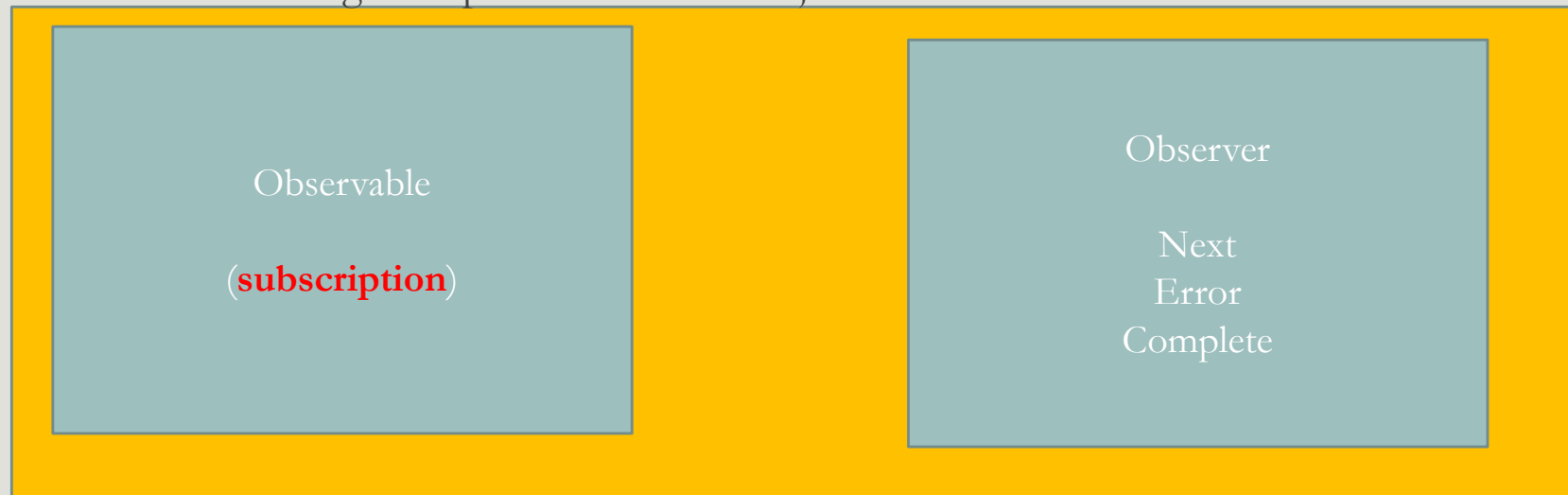
<http://reactivex.io/rxjs/manual/overview.html#operators>

<http://rxmarbles.com/>

# Les subjects

---

- Un **subject** est un type particulier d'observable. En effet Un **subject** est en même temps un **observable** et un **observer**, il possède donc les méthodes next, error et complete.
- Pour **broadcaster** une nouvelle valeur, il suffit d'appeler la méthode **next**, et elle sera diffusé aux Observateurs enregistrés pour écouter le Subject.



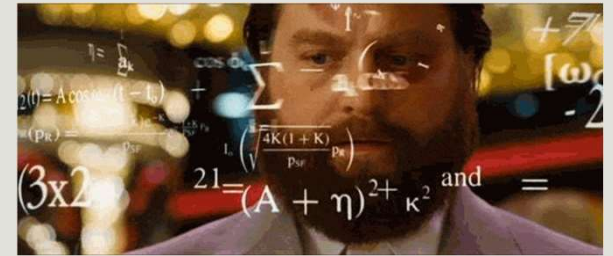
# Les subjects

---



# Exercice

---



- Modifier l'affichage des détails d'une personne au click. Enlever tous les outputs et remplacer les par l'utilisation d'un subject.

# Installation de HTTP

---

- Le module permettant la consommation d'API externe s'appelle le HTTP MODULE.
- Afin d'utiliser le module HTTP, il faut l'importer de `@angular/common/http` (`@angular/http` dans les anciennes versions) 

```
import {HttpClientModule} from "@angular/common/http";
```
- Il faudra aussi l'ajouter dans le fichier `module.ts` dans le tableau d'imports.

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

# Installation de HTTP

---

- Afin d'utiliser le module HTTP, il faut l'injecter 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 module http nous offre une méthode **get**.
- Cette méthode retourne un **Observable**.
- Cet observable a 3 callback function 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');  
  }  
);
```

# Interagir avec une API POST Request

---

- Afin d'exécuter une requête POST le module http nous offre une méthode post.
- Cette méthode retourne un Observable.
- Diffère de la méthode get avec un attribut supplémentaire : body
- Cette observable a 3 callback function 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 POST Request

---

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

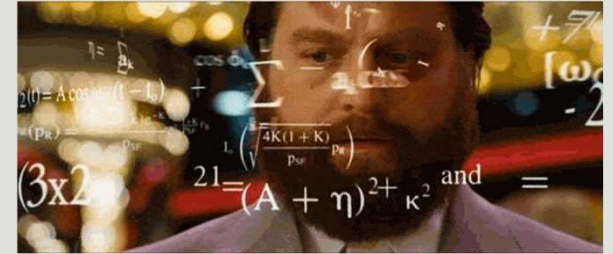
# Documentation

---

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

# Exercice

---



- Accéder au site <https://jsonplaceholder.typicode.com/>
- Utiliser l'API des posts pour afficher la liste des posts. En attendant le chargement des données afficher un message « loading... ».
- Ajouter un input. A chaque fois que vous écrivez un élément dans cet input il sera ajouté dans la liste.

# Les headers

---

- Afin d'ajouter des headers à vos requêtes, le HttpClient vous offre la classe HttpHeaders.
- Cette classe est une classe immutable (read Only).

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

- 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 HttpHeaders permettant un chainage d'appel.

<https://angular.io/api/common/http/HttpHeaders>

# Les paramètres

---

- Afin d'ajouter des paramètres à vos requêtes, le HttpClient vous offre la classe HttpParams.
- Cette classe est une classe immuable (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.

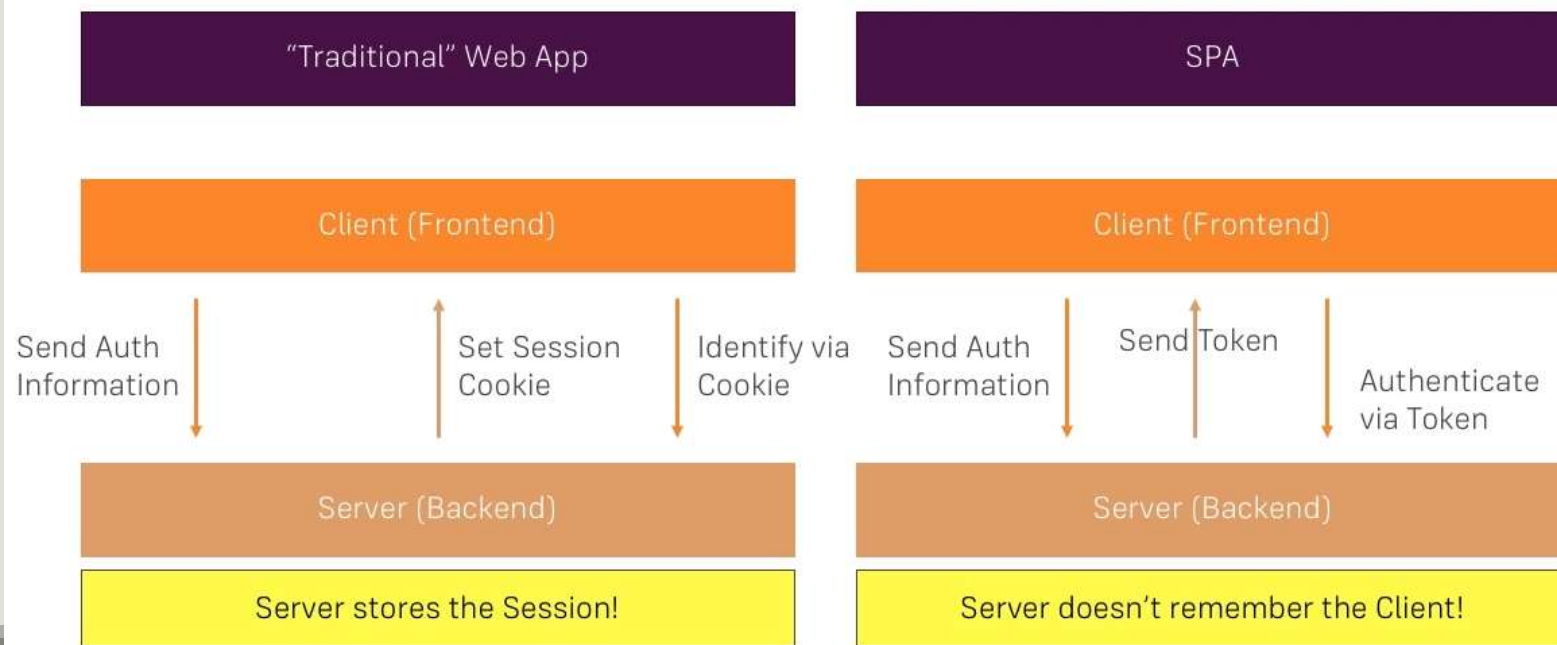
Toutes les méthodes de modification retourne un HttpParams permettant un chainage d'appel.

<https://angular.io/api/common/http/HttpParams>



# Authentication

## How does Authentication work?



# Ajouter le token dans la requête

---

- 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});
```

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

# Ajouter le token dans la requête

---

- 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});
```

# Sécuriser vos routes

---

- Dans vos applications, certaines routes ne doivent être accessibles que si vous êtes authentifié. 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 **Guard**.

# Guard

---

- Ce sont des classes 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` permettre ou non l'accès à une route.
  - `CanActivateChild` permettre ou non l'accès aux routes filles.
  - `CanDeactivate` permettre ou non la sortie de la route.

# Guard / canActivate

---

- Afin d'utiliser le guard `canActivate` (de même pour les autres), vous devez créer une classe qui implémente l'interface `CanActivate` et donc qui doit implémenter la méthode `canActivate` de sorte qu'elle retourne un booléen permettant ainsi l'accès ou non à la route cible. 1
- Vous devez ensuite ajouter cette classe dans le provider. 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. 3
- Vous pouvez utiliser la méthode : `ng g g nomGuard`

# Guard / canActivate

1

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor() {
    // route contient la route appelé
    // state contiendra la futur état du routeur de l'application qui devra passer la validation du guard
    // https://vsavkin.com/routeur-angular-comprendre-l%C3%A9tat-du-routeur-5e15e729a6df
    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> |
    Promise<boolean> | boolean {
      if (// your condition) {
        return true;
      }
      return false;
    }
  }
}
```

# Guard / canActivate

---

2

```
providers: [  
  TodoService,  
  CvService,  
  LoginService,  
  AuthGuard,  
],
```

App.module.ts



# Guard / canActivate

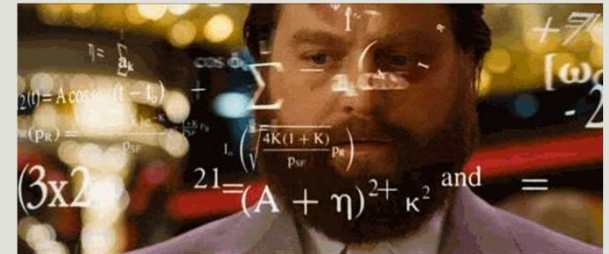
---

3

```
{  
  path: 'lampe',  
  component: ColorComponent,  
  canActivate: [AuthGuard]  
},
```

# Exercice

---

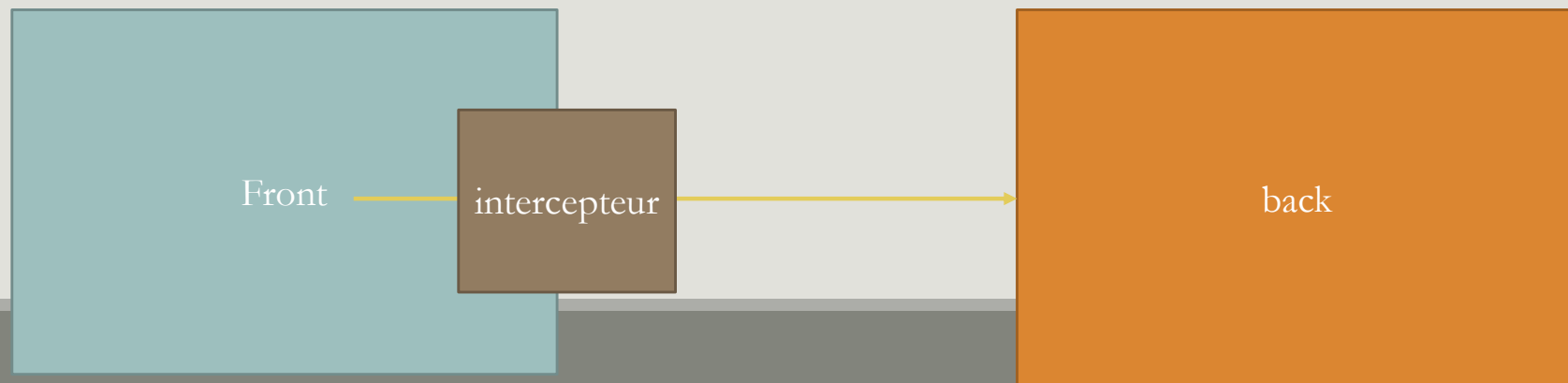


- Ajouter les guards nécessaires afin de sécuriser vos routes.
- Une personne déjà connectée ne peut pas accéder au composant de login.

# 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 intercepteur 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

---

```
export class AuthenticationInterceptor implements HttpInterceptor {  
  intercept(req: HttpRequest<any>, next: HttpHandler):  
    Observable<HttpEvent<any>> {  
    console.log('intercepted', req);  
    return next.handle(req);  
  }  
}
```

# Les intercepteurs

---

- Un intercepteur est injecté au niveau du provider. Si vous voulez intercepter toutes les requêtes, vous devez le provider au niveau du module principal.
- L'inscription au niveau du provider se fait de la façon suivante :

```
export const
AuthenticationInterceptorProvider = {
  provide: HTTP_INTERCEPTORS,
  useClass: AuthenticationInterceptor,
  multi: true,
};
```

```
providers: [
  AuthenticationInterceptorProvider
],
```

# Les intercepteurs : changer 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.

```
export const
AuthenticationInterceptorProvider = {
  provide: HTTP_INTERCEPTORS,
  useClass: AuthenticationInterceptor,
  multi: true,
};
```

```
providers: [
  AuthenticationInterceptorProvider
],
```

# Cloner une requête

---

```
const newReq = req.clone({
  headers: new HttpHeaders()// faites ce que vous voulez ici ajouter des
headers, des params ...
});
// Chainer la nouvelle requete avec next.handle
return next.handle(newReq);
```

# Intercepter les erreurs

- Afin d'intercepter les erreurs, il faut récupérer la réponse et vérifier s'il y a une erreur. Dans ce cas, il faut faire le traitement souhaité.

```
intercept(req: HttpRequest<any>, next: HttpHandler):  
Observable<HttpEvent<any>> {  
    return next.handle(req)  
        .pipe(  
            tap(  
                (incoming: any) => {  
                    console.log('here its ok');  
                },  
                (error: HttpErrorResponse) => {  
                    return throwError(error);  
                }  
            )  
        );  
}
```

```
intercept(req: HttpRequest<any>, next: HttpHandler):  
Observable<HttpEvent<any>> {  
    return next.handle(req)  
        .pipe(  
            catchError(err => {  
                console.log(err);  
                return new  
Observable<HttpEvent<any>>((observer) => {  
                    observer.error(err);  
                }));  
            })  
        );  
}
```



# Déploiement

---

- Afin de déployer votre application, il vous suffit d'utiliser la commande suivante :

```
ng build --prod
```

- Un dossier dist sera créé contenant votre projet
- Pour tester localement votre projet, télécharger un serveur HTTP virtuel avec la commande suivante :

```
Npm install http-server -g
```

- Lancer maintenant votre projet à l'aide de cette commande :

```
http-server dist/NomDeVotreProjet
```

# Déployer votre application sur GitHub Pages (angularCli >= 8.3)

---

- Créer un repository et mettez y votre projet
- installer angular-cli-ghpages : `ng add angular-cli-ghpages`
- Ajouter cette configuration dans votre fichier `angular.json` si ca n'a pas été fait automatiquement

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

- Lancer la commande `ng deploy --base-href=/the-repositoryname/`
- Accéder à votre page [https://USERNAME.github.io/REPOSITORY\\_NAME](https://USERNAME.github.io/REPOSITORY_NAME)
- En cas de mise à jour, relancer le même code.

<https://github.com/angular-schule/angular-cli-ghpages>

# Déployer votre application sur GitHub Pages (angularCli $\geq$ 8.3)

---

Ajouter cette configuration dans votre fichier angular.json et utiliser uniquement ng deploy

```
"deploy": {  
  "builder": "angular-cli-ghpages:deploy",  
  "options": {  
    "baseHref": "https://username.github.io/repoName/",  
  }  
}
```

---

aymen.sellaouti@gmail.com