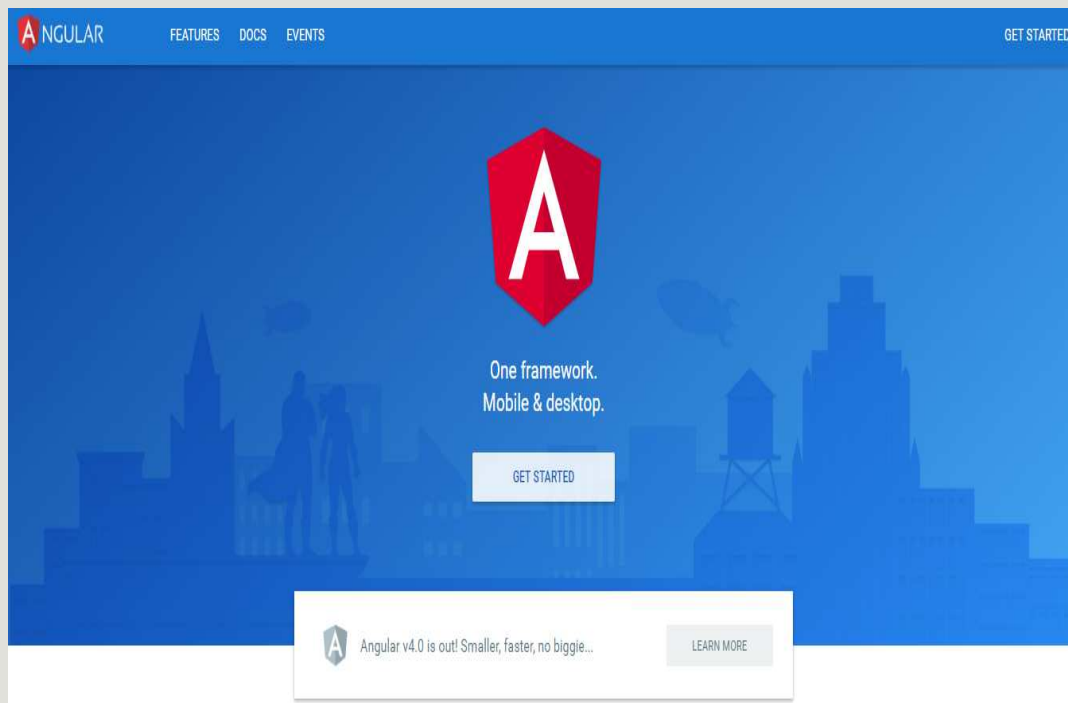


# Angular Introduction

---

AYMEN SELLAOUTI

# Références



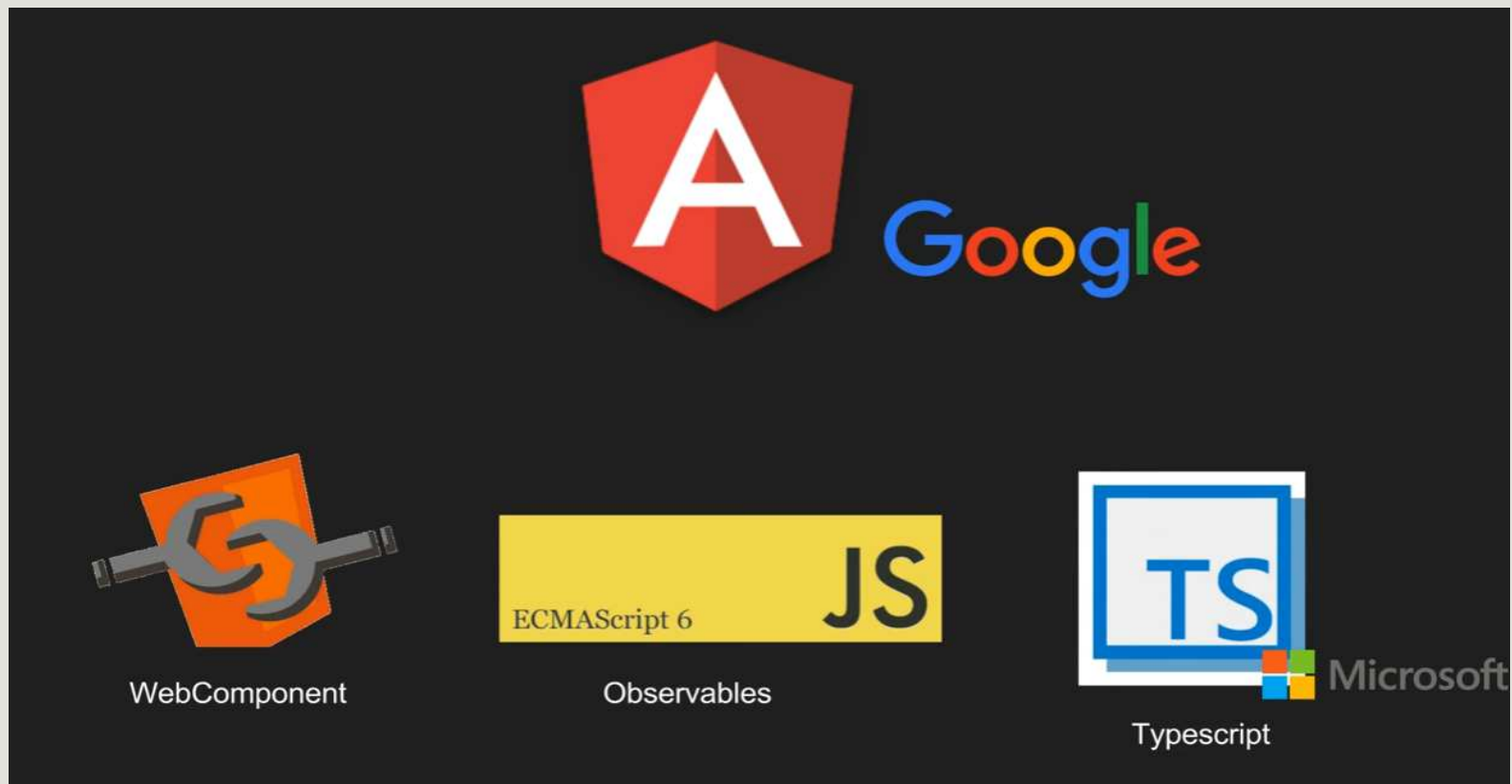
# Plan du Cours

---

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
8. Les modules
9. Les tests unitaires

# C'est quoi Angular?

---

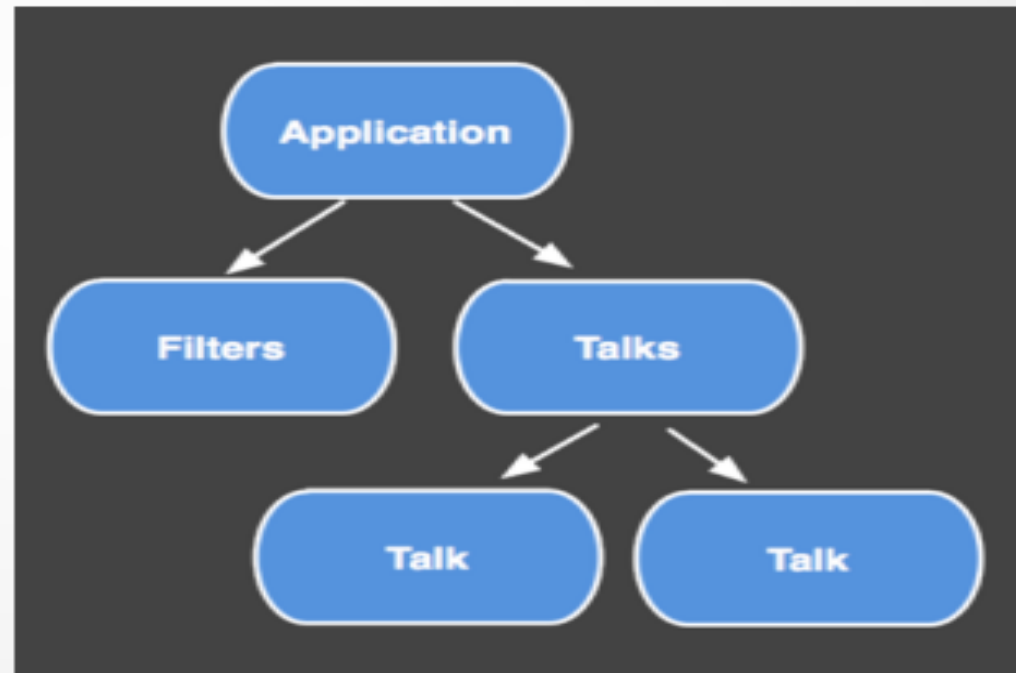
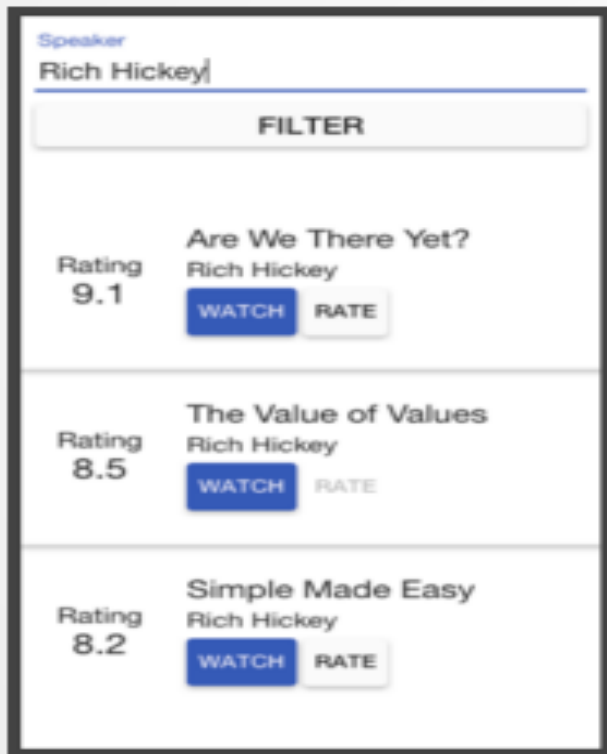


# C'est quoi Angular?

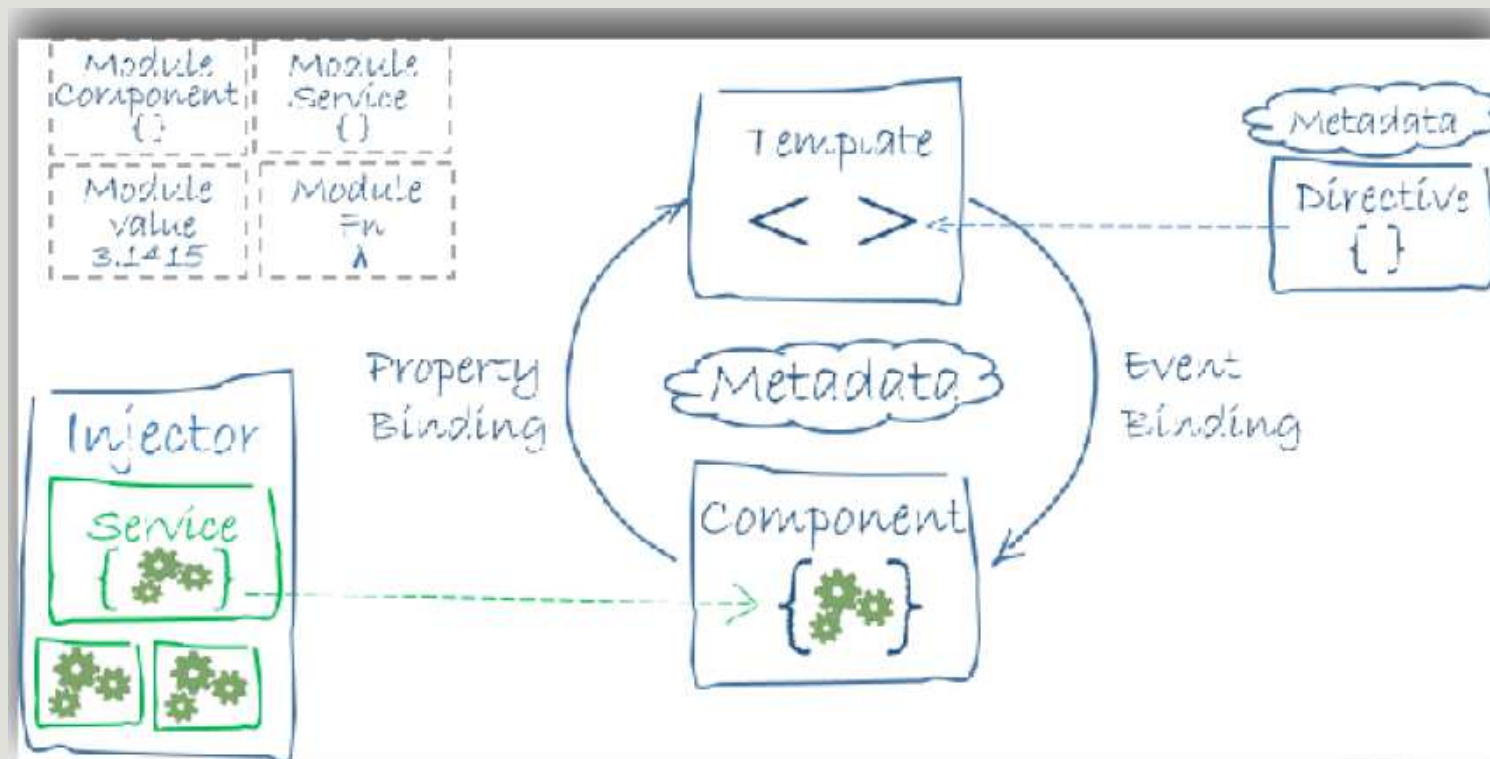
---

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

# Angular : Arbre de composants



# Architecture Angular



# Principaux concepts et notions

---

Component

Template

DataBinding

Méta données

Service

Route

Injection de  
dépendance



# Module

---

Angular est modulaire

- 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

---

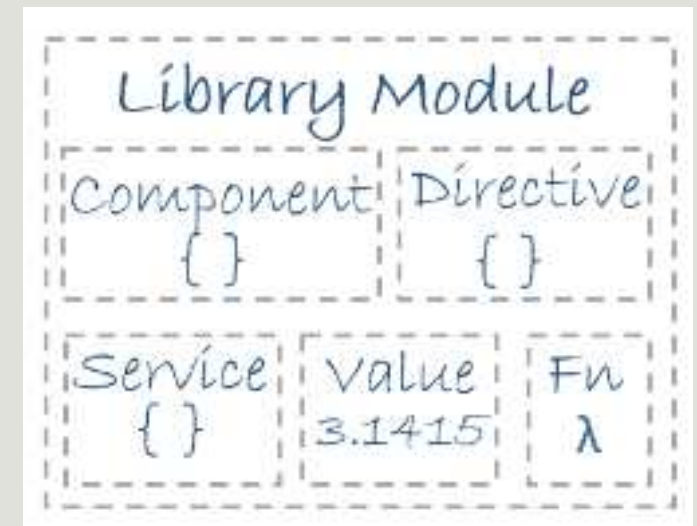
➤ Ensemble de modules JS

➤ 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

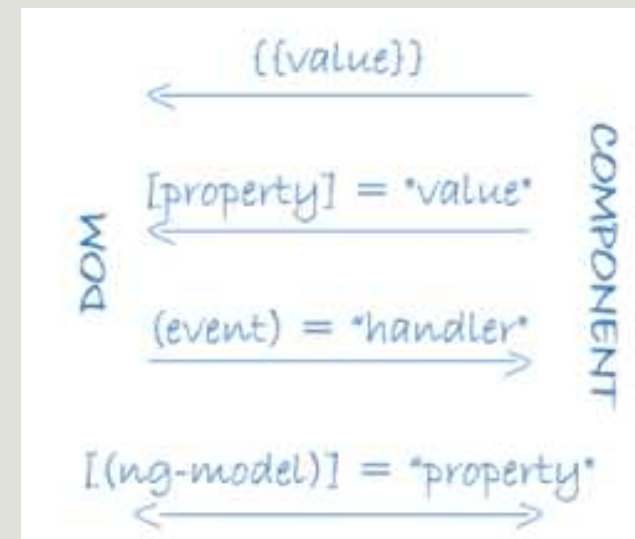
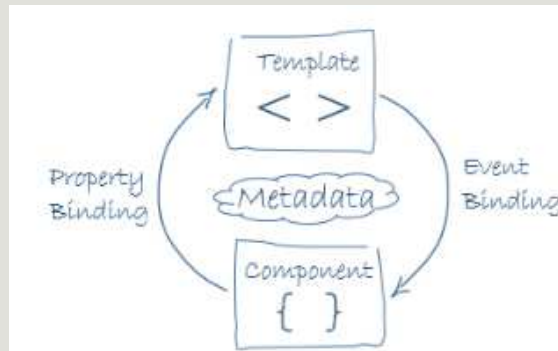
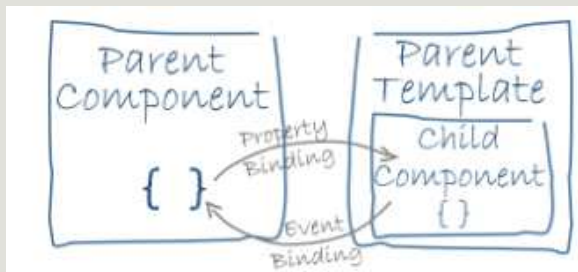
---



- 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 :

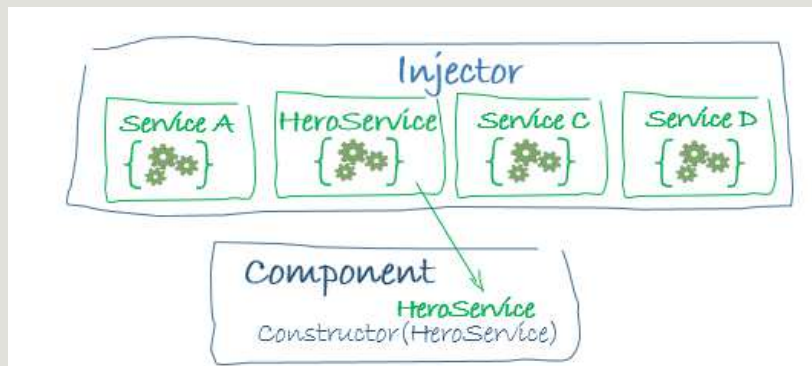
➤ Directives structurelles

➤ 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

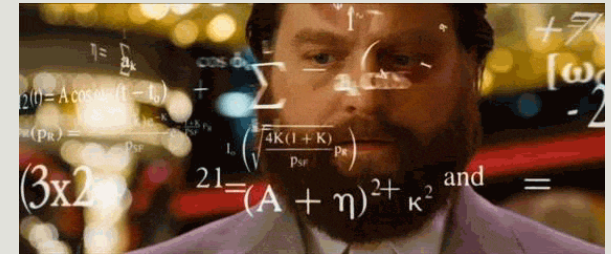
`https://github.com/angular/quickstart`

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

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

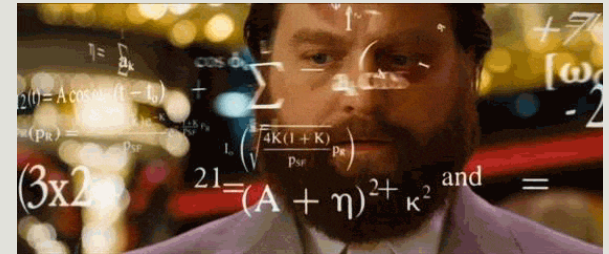
- Se positionner sur le projet
- Installer les dépendance à l'aide de npm : `npm install`
- lancer le projet à l'aide de npm : `npm start`

# 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 >= 8.3.0](#)
- Une fois installé vous disposez de npm qui est le [Node Package Manager](#). Afin de vérifier si vous avez NodeJs installé, tapez [npm -v](#).
- La version npm devra être [>=3.0.0](#)
- Installer ensuite TypeScript. Pour ce faire, tapez [npm install -g typescript](#).
- Installer maintenant le Cli en tapant la : [npm install -g @angular/cli](#)
  - [npm install -g @angular/cli@6.0.0](#) installe la version 6.0.0
  - [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](#)
- Afin d'avoir du help pour le cli tapez [ng help](#)
- Lancer le projet en utilisant la commande [ng serve](#)

# Installation d'Angular Angular Cli

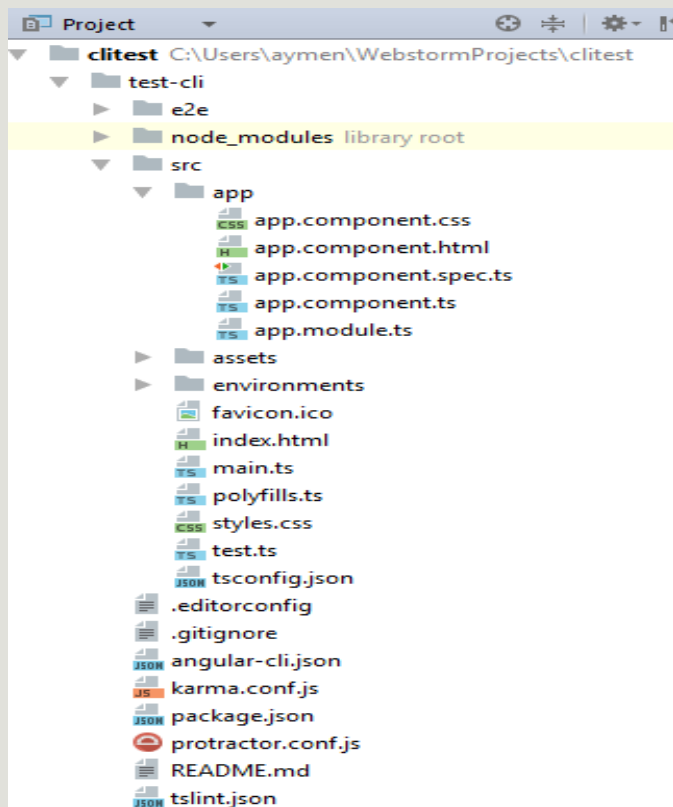


- Positionnez vous maintenant dans le dossier
- Tapez la commande suivante : `ng new nomNewProject`
- lancez le projet à l'aide de npm : `ng serve`
- Naviguez vers l'adresse mentionnée.
- Vous pouvez configurer le Host ainsi que le port avec la commande suivante : `ng serve --host leHost --port lePort`
- 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>

# Arborescence d'un projet Angular



E2e : end to end pour réaliser des tests d'intégration

Node\_modules : les dépendances

Src : dossier contenant l'index le code source les styles, main.ts qui est le bootstrap d'angular ainsi que d'autres fichiers

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.

# 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

# 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
8. Les modules
9. Les tests unitaires

# Objectifs

---

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

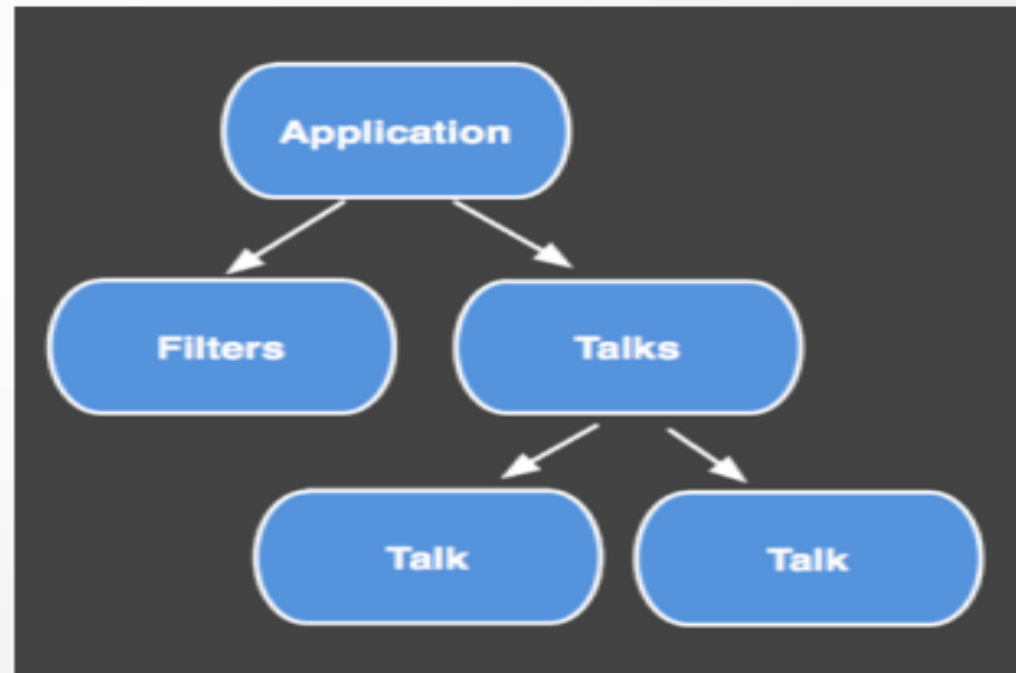
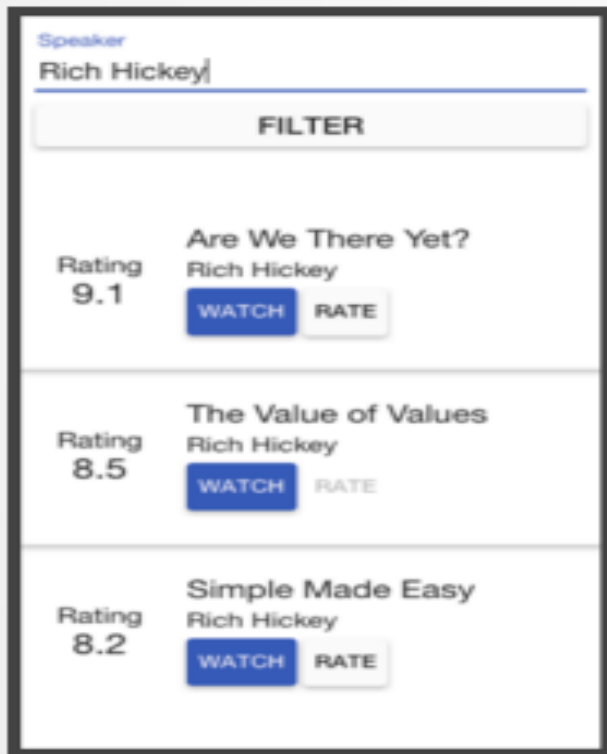
# Qu'est ce qu'un composant (Component)

---

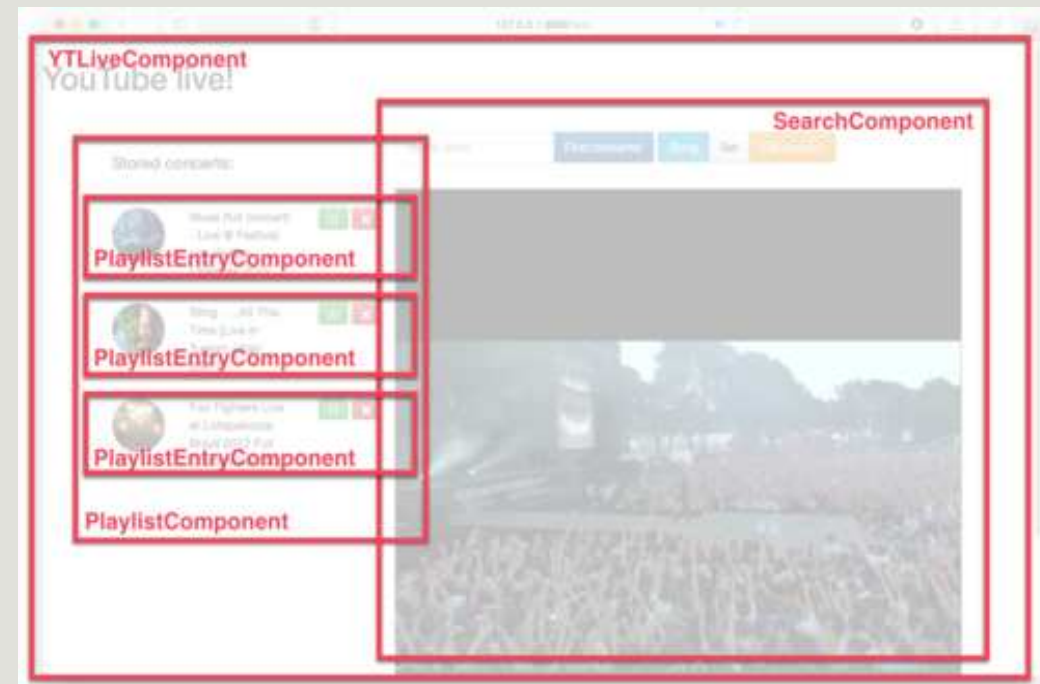
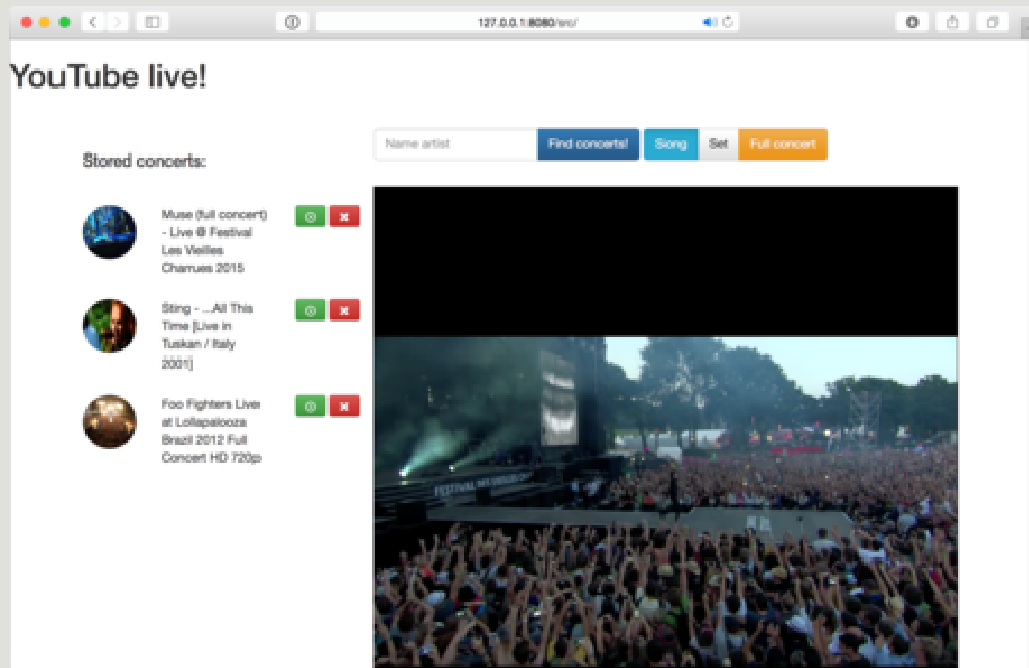
- Un **composant** est une **classe** qui permet de **gérer une vue**. Il se charge **uniquement** de cette vue là.  
Plus simplement, un composant est un fragment HTML géré par une classe JS (component en angular et controller en angularJS)
- Une application Angular est un **arbre de composants**
- 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.**

# Angular : Arbre de composants

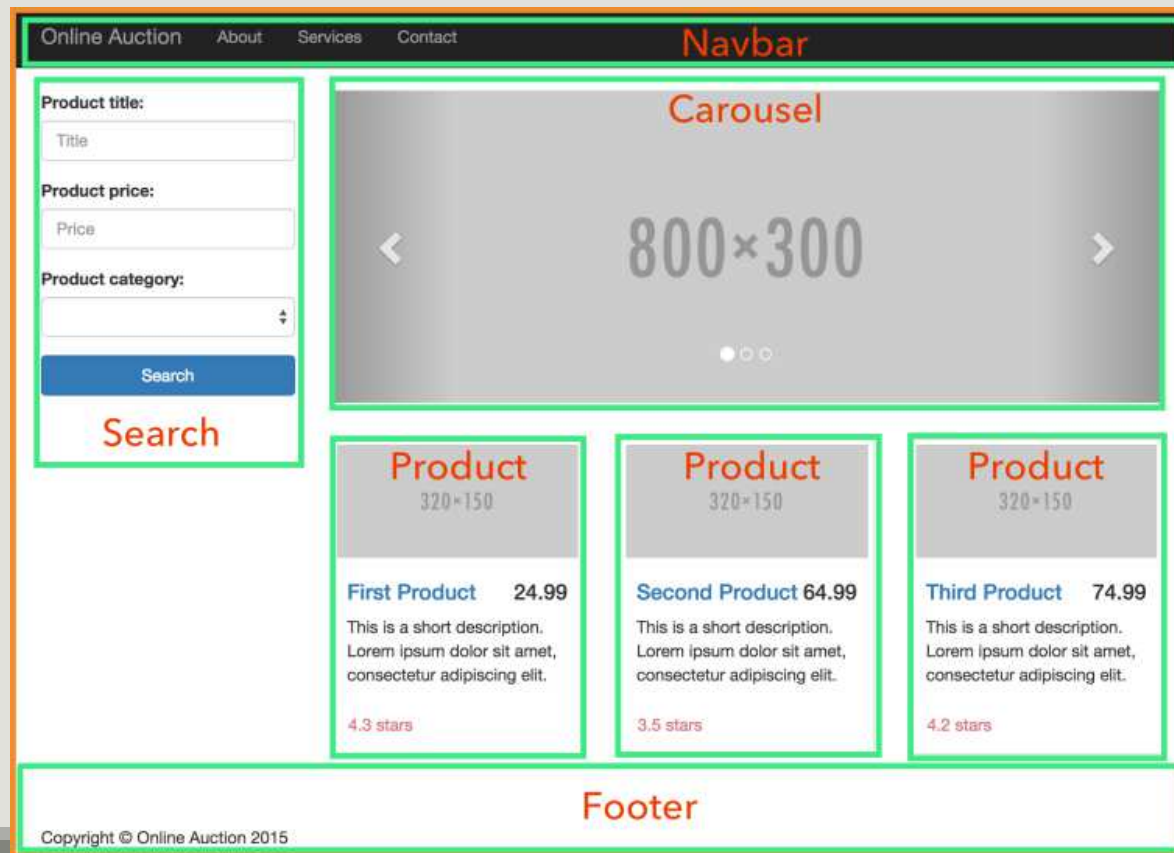


# Quelques exemples

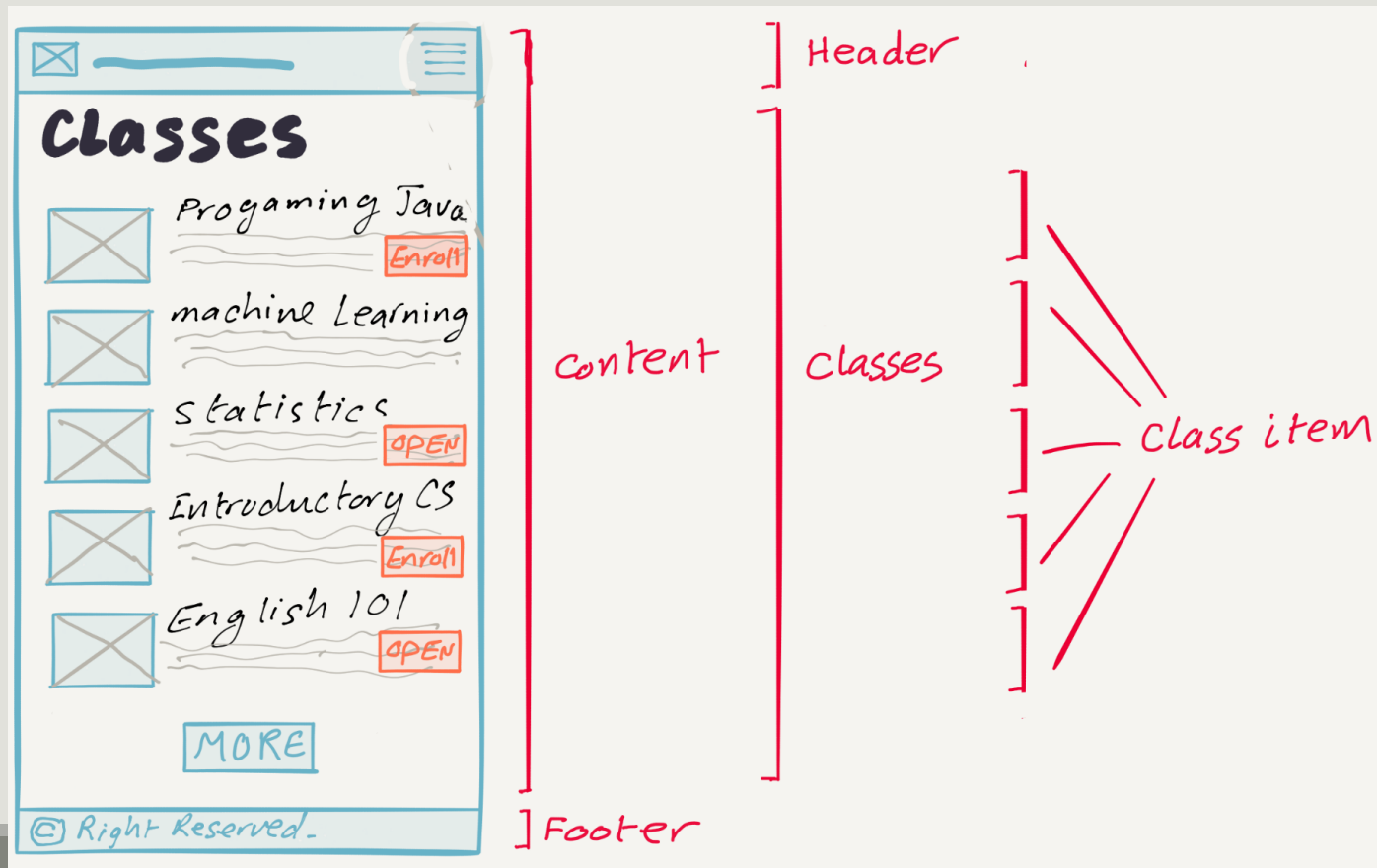




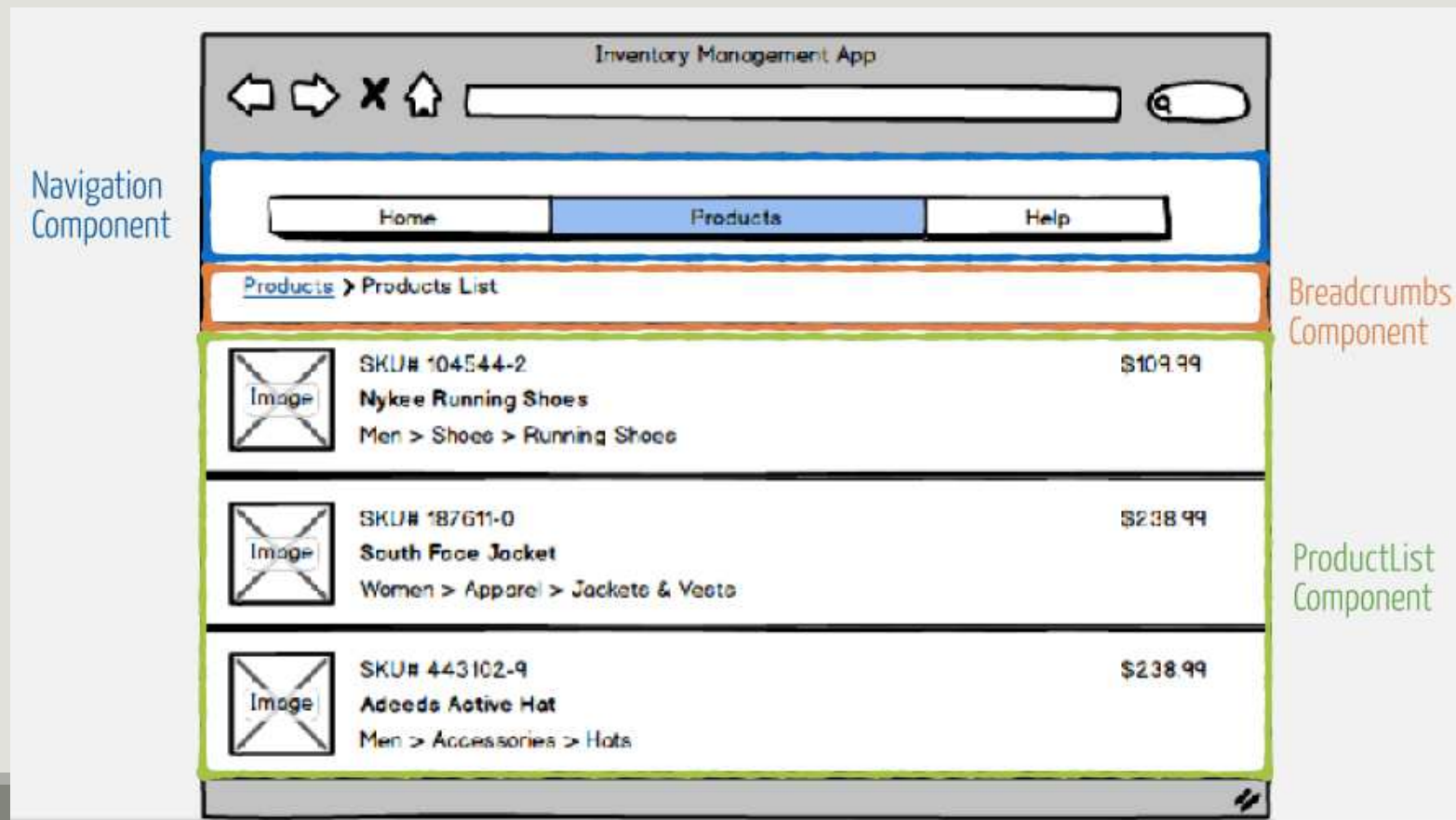
# Quelques exemples



# 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 Nykee Running Shoes 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**

# Property Binding

---

Balises

Propriétés

HTML

Attributs

Méthodes

TS



# Property 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

# 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

# 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

# Property Binding et Event Binding

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

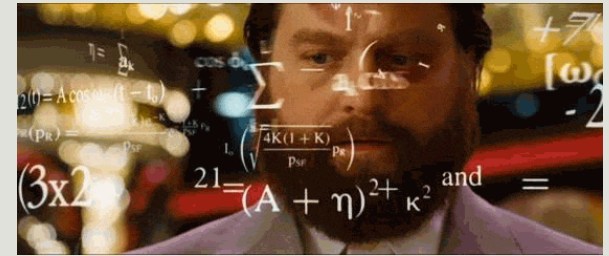
@Component({
  selector: 'inter-interpolation',
  template: `interpolation.html`,
  styles: []
})
export class InterpolationComponent {
  nom:string = 'Aymen Sellaouti';
  age:number = 35;
  adresse:string = 'Chez moi ou autre part :)';
  getName() {
    return this.nom;
  }
  modifier(newName) {
  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

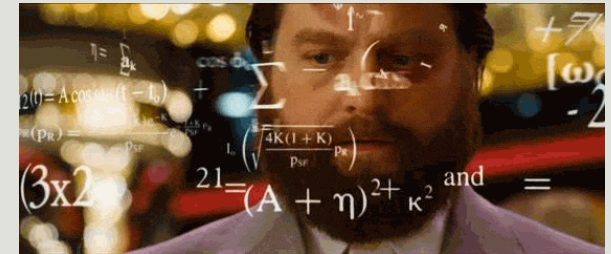
@Component ({
  selector: 'app-two-way',
  templateUrl: './two-
way.component.html',
  styleUrls: ['./two-
way.component.css']
})
export class TwoWayComponent {
  two:any="myInitValue";
}
```

Component

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

Template

# Exercice

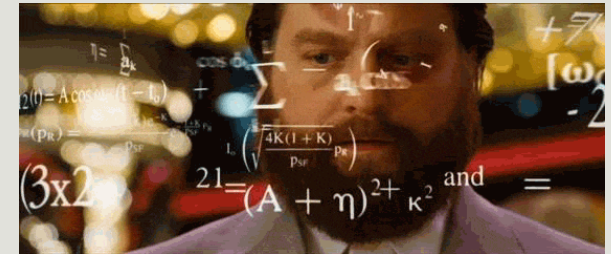


- Le but de cet exercice est de créer un aperçu de carte visite.
- Créer un composant
- Préparer une interface permettant de saisir d'un coté les données à insérer dans une carte visite. De l'autre coté et instantanément les données de la carte seront mis à jour.
- Préparer l'affichage de la carte visite. Vous pouvez utiliser ce thème gratuit :

<https://www.creative-tim.com/product/rotating-css-card>



# Exercice



## Two Way Binding



**Sellaouti Aymen**  
Enseignant

tant qu'il y a de la vie il y a de l'espoir

 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

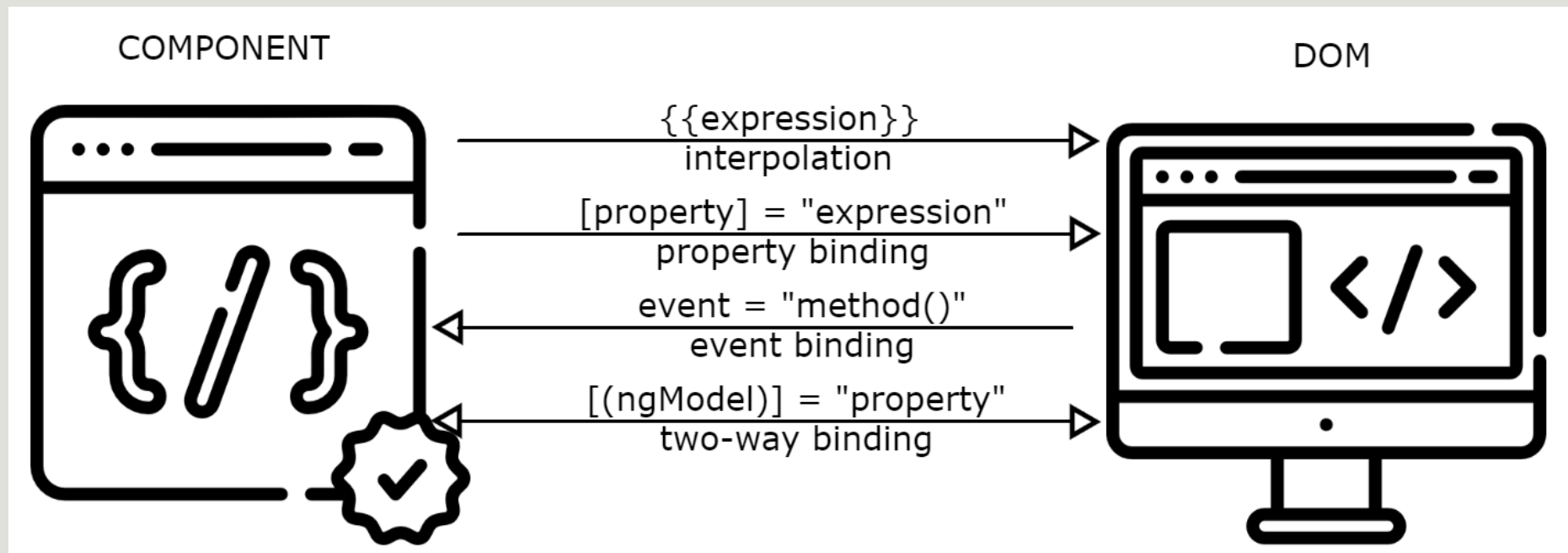
image :  
as.jpg

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



# Cycle de vie d'un composant

---

- Le composant possède un cycle de vie géré par Angular. En effet Angular :
  - Crée le composant
  - L'affiche
  - Crée ses fils
  - Les affiche
  - Ecoute le changement des propriétés
  - Le détruit avant de l'enlever du DOM

<https://medium.com/bhargav-bachina-angular-training/angular-understanding-angular-lifecycle-hooks-with-a-sample-project-375a61882478>

# Comment ça marche réellement

---

Exemple :

```
<mon-app>  
  <compo-fils [binded]='prop'>
```

1. Angular lance l'application
2. Ils crée les classes pour chaque composant, il lance donc le Constructeur de mon-app component.
3. Il gère toutes les dépendances injectées au niveau du constructeur et les définit comme des paramètres
4. Il crée le nœud du Dom qui va héberger le composant
5. Il commence ensuite la création du composant fils et appelle son constructeur. Ici la propriété binded n'est pas prise en considération par Angular. Le Life Hook cycle n'est pas encore déclenché.
6. A ce moment Angular lance le processus de détection des changements. C'est ici qu'il mettra à jour le binding de mon-App et lancera l'appel à ngOnInit, suivi de la gestion du binding de compo-fils puis l'appel de son ngOnInit.

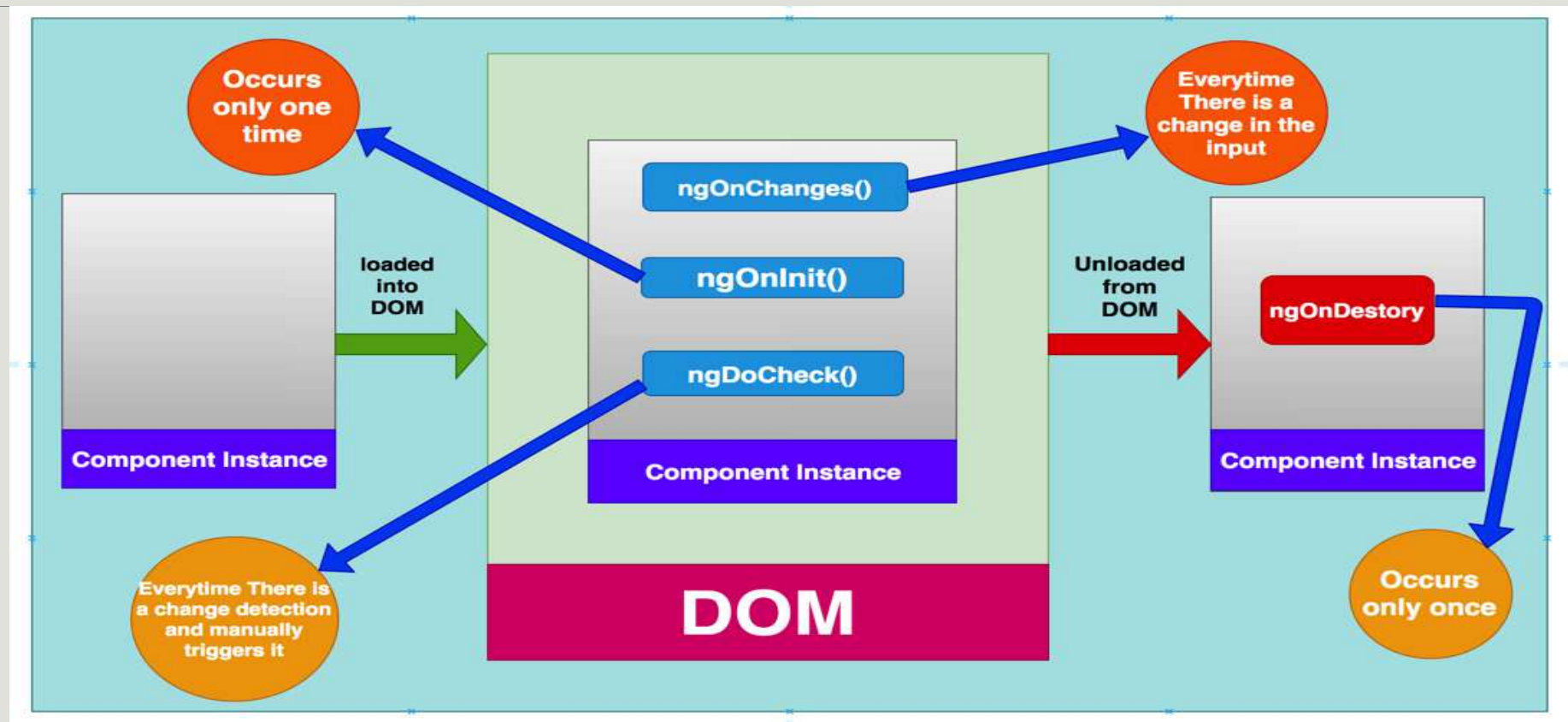
# Interfaces de gestion du cycle de vie d'un composant

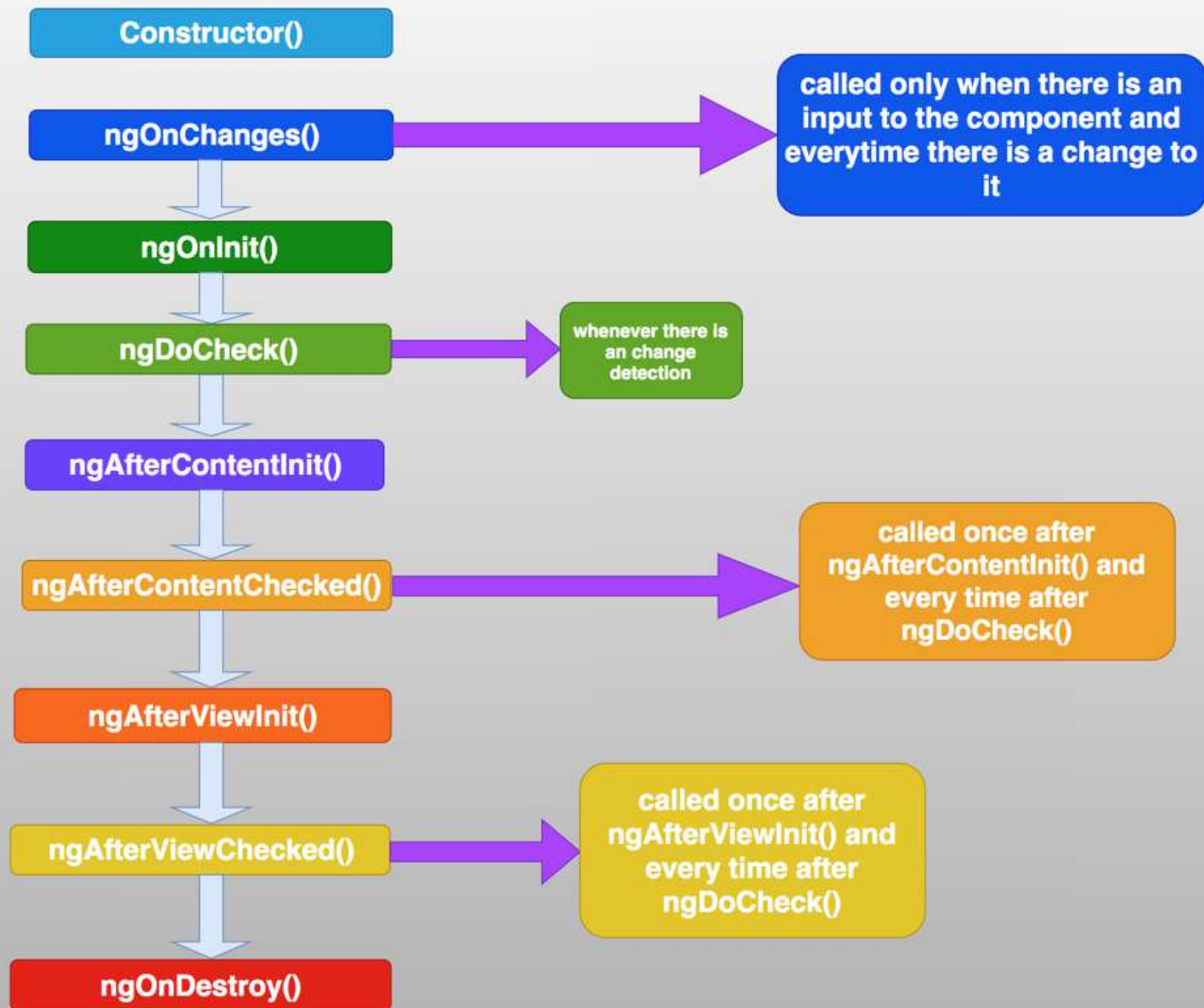
---

- Afin de gérer le cycle de vie d'un composant, Angular nous offre un ensemble d'interfaces à implémenter pour les différentes étapes du cycle de vie.
- L'ensemble des interfaces est disponible dans la librairie core d'Angular
- Chaque interface offre une seule méthode dont le nom suit la convention suivante :

**ng + NomDeL'Interface**

Exemple : Pour l'interface `OnInit` nous avons la méthode `ngOnInit`





# Interfaces de gestion du cycle de vie d'un composant

---

- **ngOnChanges**: Cette méthode est appelée lors d'un binding d'une propriété d'une directive.. La méthode reçoit en paramètre un objet représentant les valeurs actuelles et les valeurs précédentes disponibles pour ce composant.
- **ngOnInit**: Cette méthode initialise le composant après qu'Angular ait initialisé les propriétés du composant.
- **ngDoCheck**: Appelé après chaque change détection.

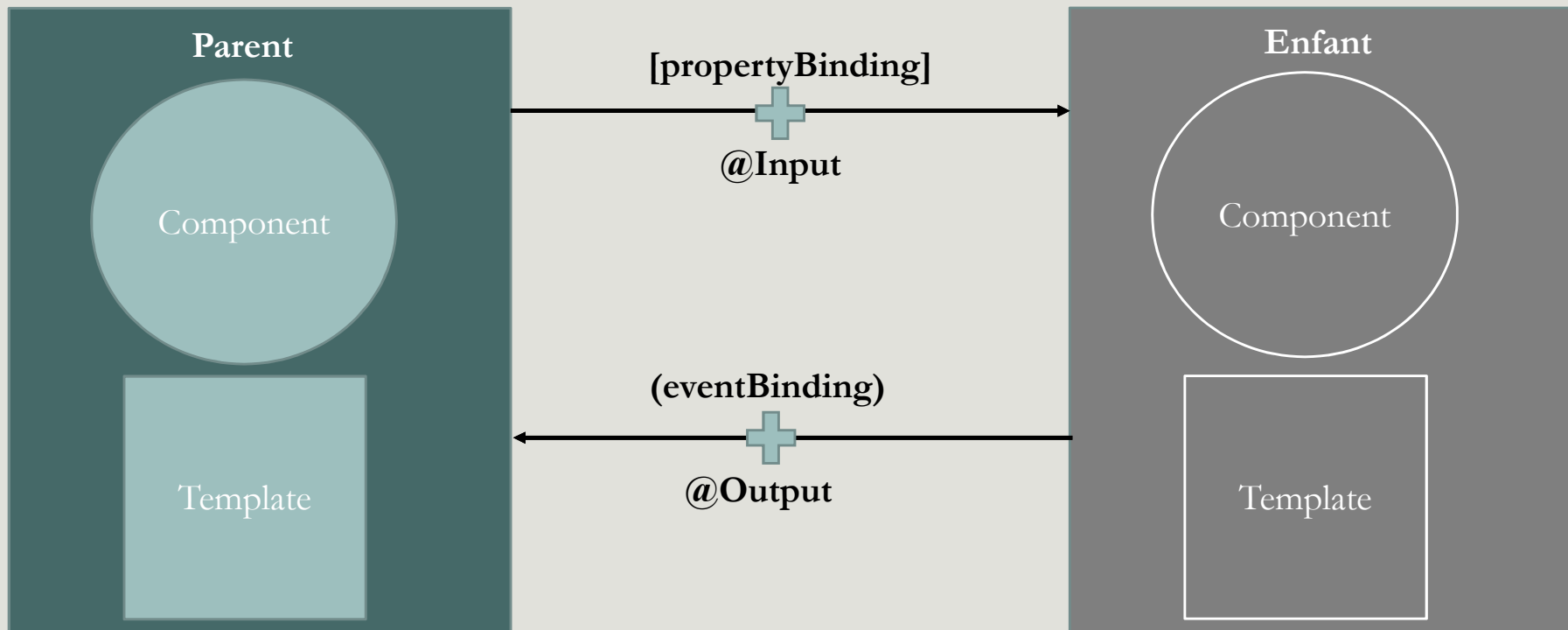


# Interfaces de gestion du cycle de vie d'un composant

---

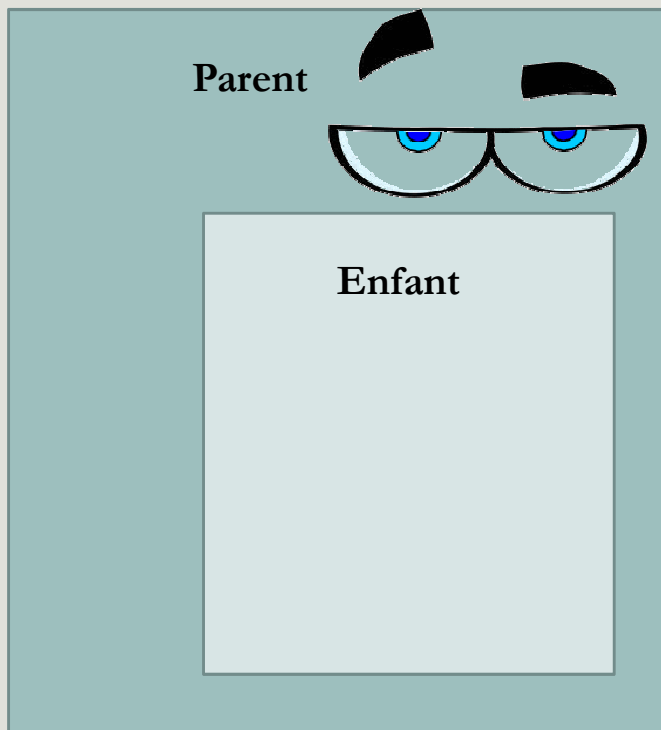
- **ngAfterViewInit**: Cette méthode est appelée juste après la mise en place de la vue d'un composant, et des vues de ses composants fils s'il en a.
- **ngOnDestroy**: Appelée en dernier, cette méthode est appelée avant qu'Angular ne détruise et ne retire du DOM le composant. Cela peut se produire lorsqu'un 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".

# Interaction entre composants



# Pourquoi ?

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

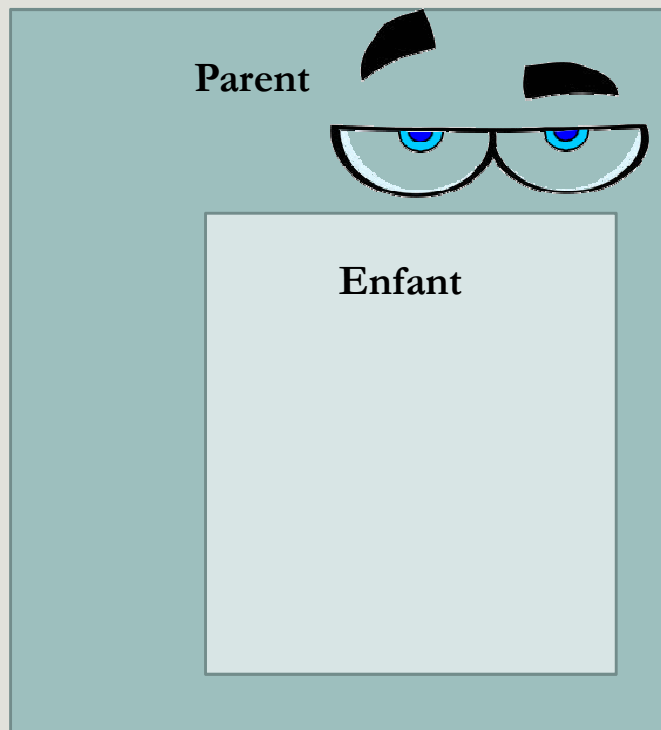


```
import { Component } from '@angular/core';

@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**

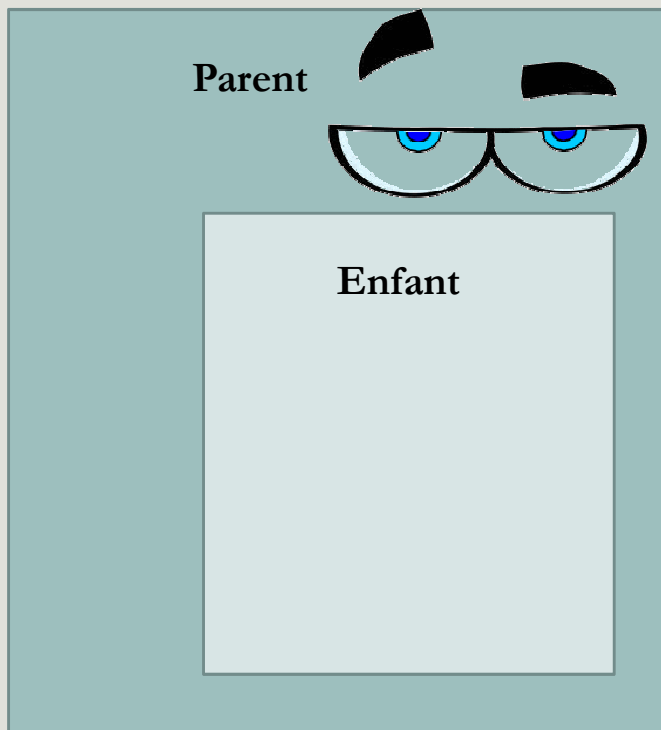


```
import { Component } from '@angular/core';

@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

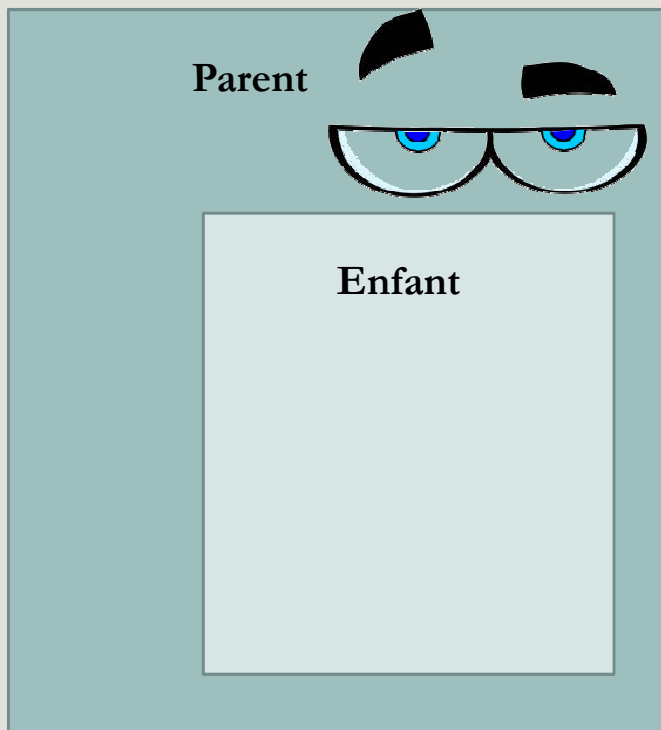


```
import { Component } from '@angular/core';

@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';

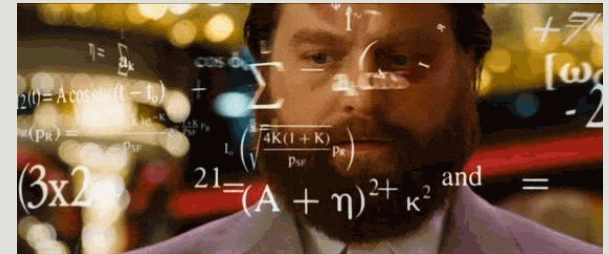
@Component({
  selector: 'app-root',
  template: `
    <p>Je suis le composant père </p>
    <forma-fils [external]="title">
  </forma-fils>
`,
  styles:
})
export class AppComponent {
  title = 'app works !';
}
```

```
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

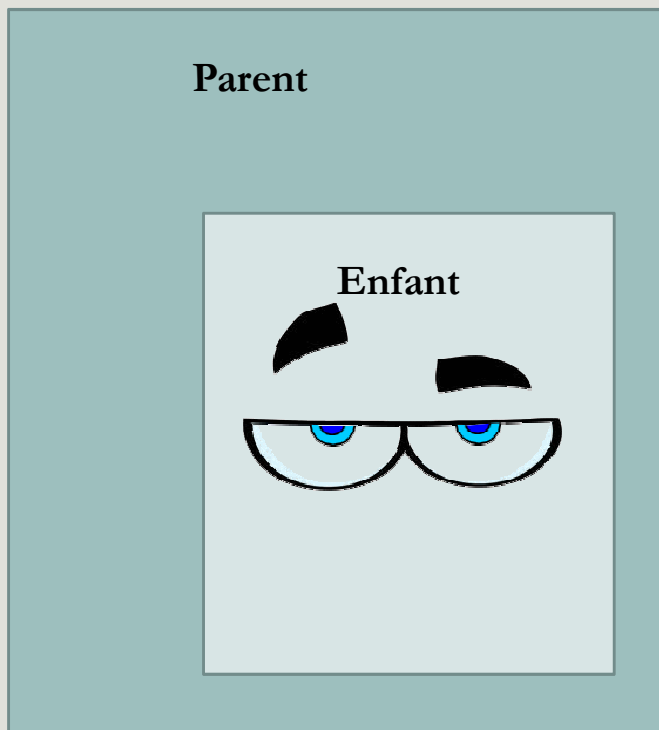
---



- 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
    );
  }
}
```

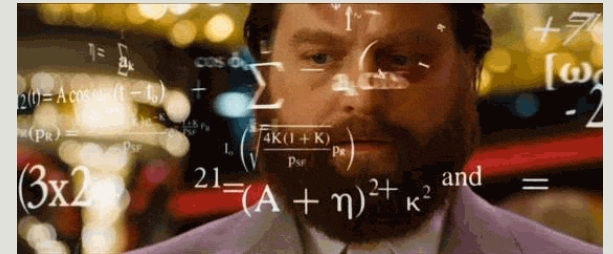
Enfant

```
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

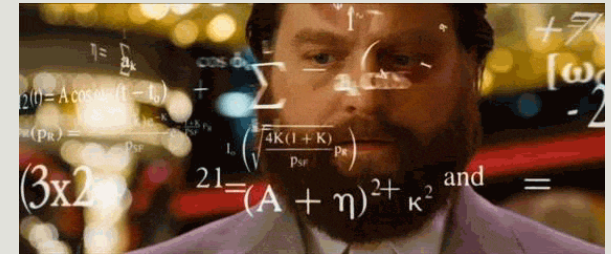
# Exercice

---

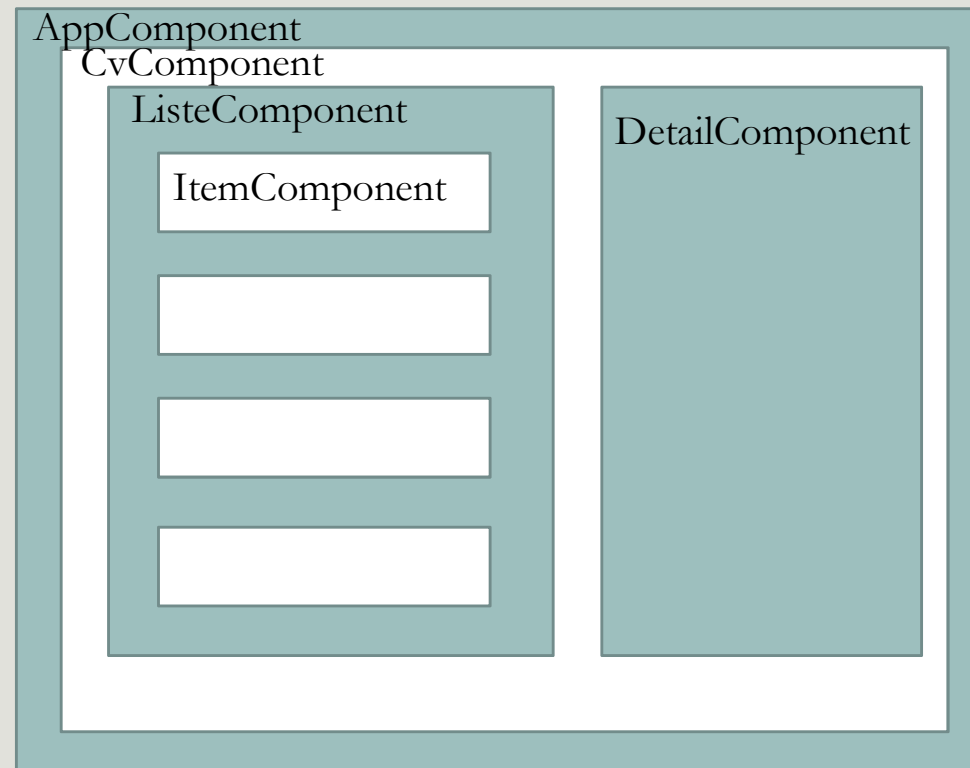


- Ajouter une variable `myFavoriteColor` dans le composant du fils.
- Ajouter un bouton dans le composant Fils
- Au click sur ce bouton, la couleur de background du Div du père doit prendre la couleur favorite du fils.

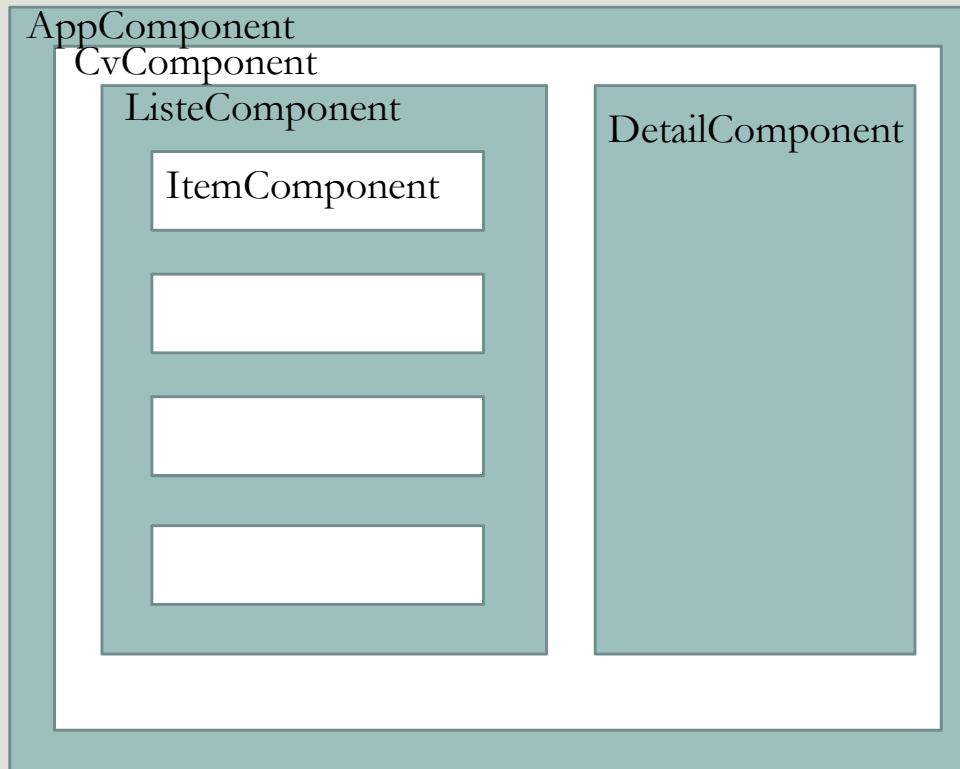
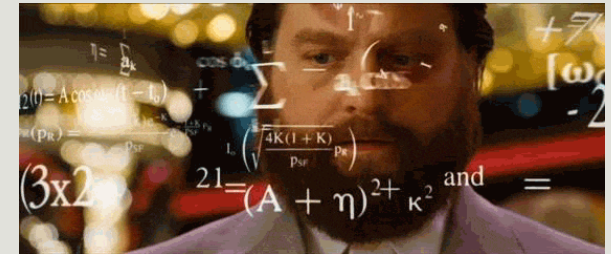
# Exercice



- 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

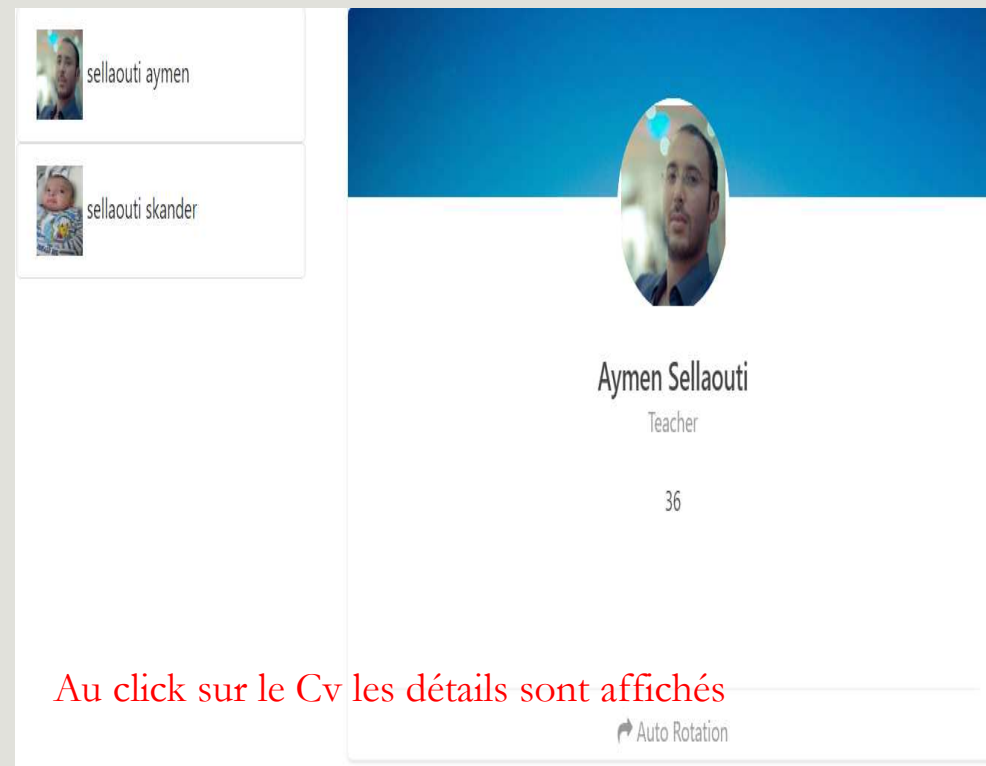
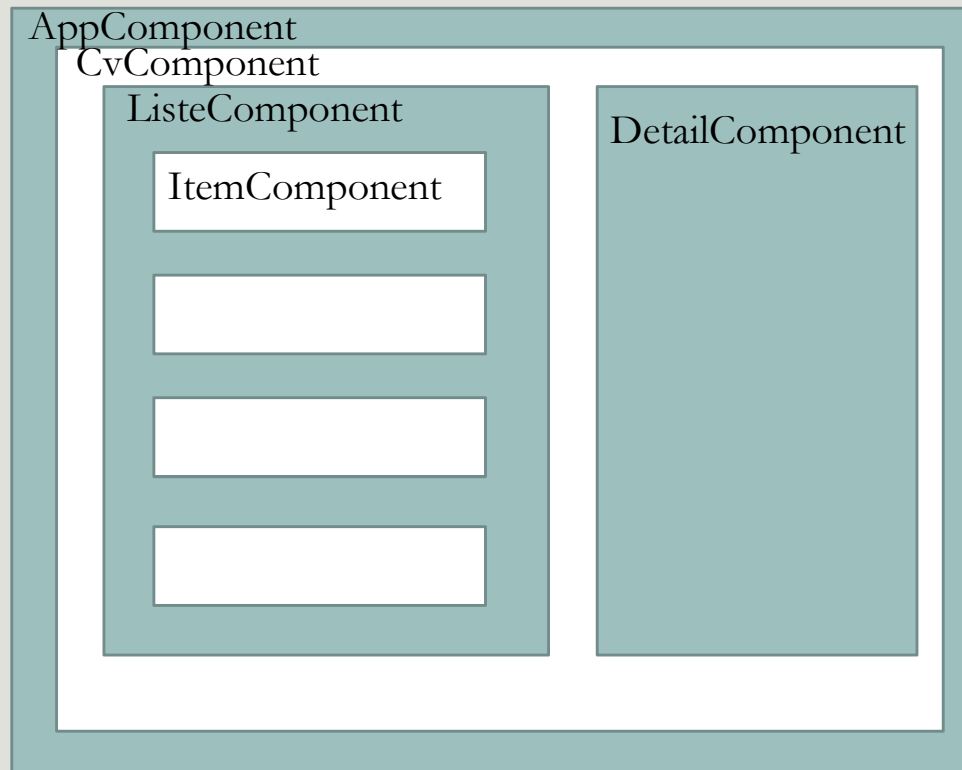
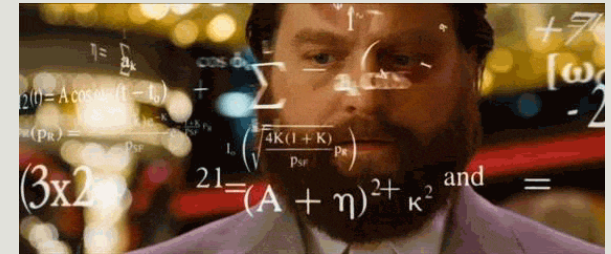


sellayouti aymen



sellayouti skander

# Exercice



# Exercice

Un cv est caractérisé par :

id

Nom

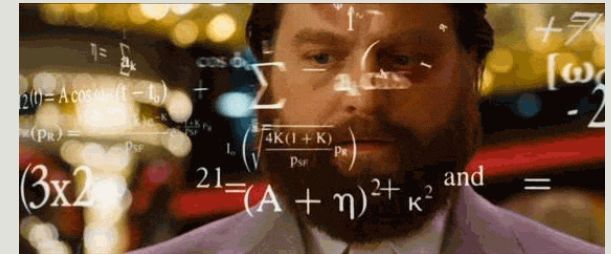
Prenom

Age

Cin

Job

path



sellaouti aymen



sellaouti skander



Aymen Sellaouti

Teacher

36

Au click sur le Cv les détails sont affichés

Auto Rotation

# Angular

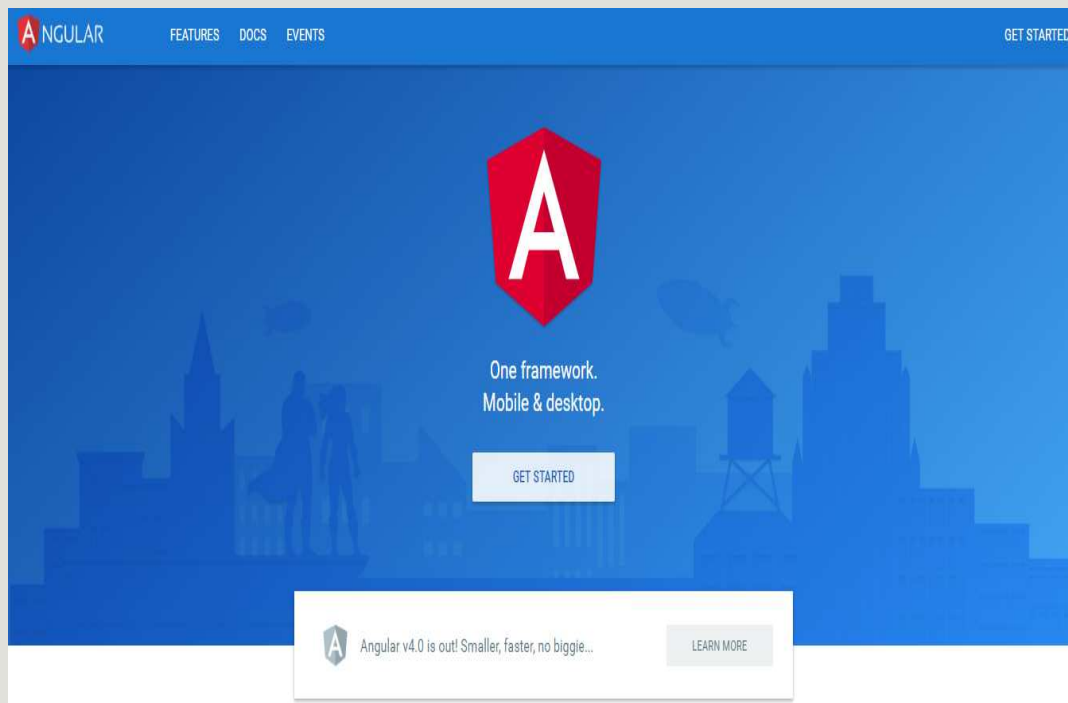
# Les directives

---

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
8. Les modules
9. Les tests unitaires

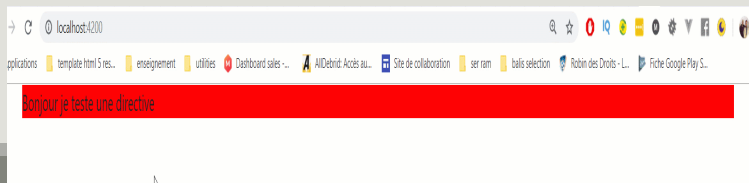
# Objectifs

---

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)

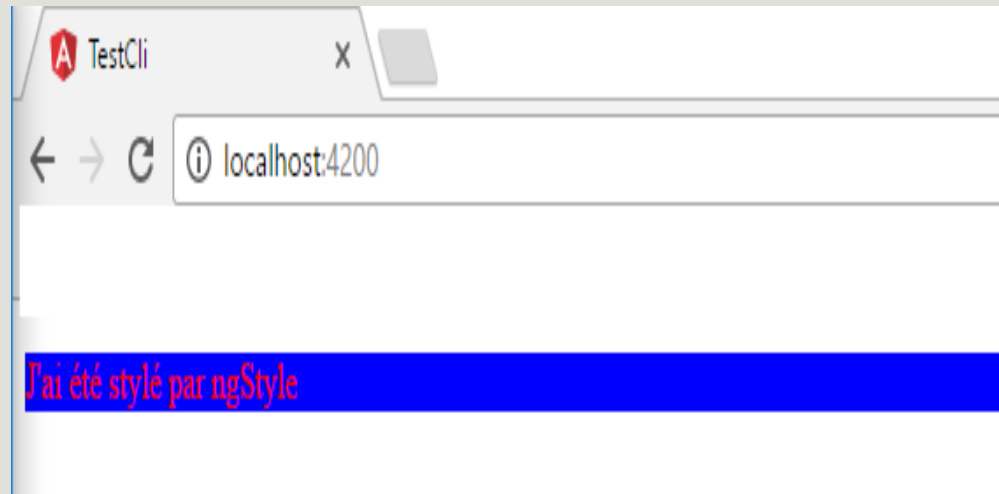
```
import { Component } from
 '@angular/core';

@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)

---





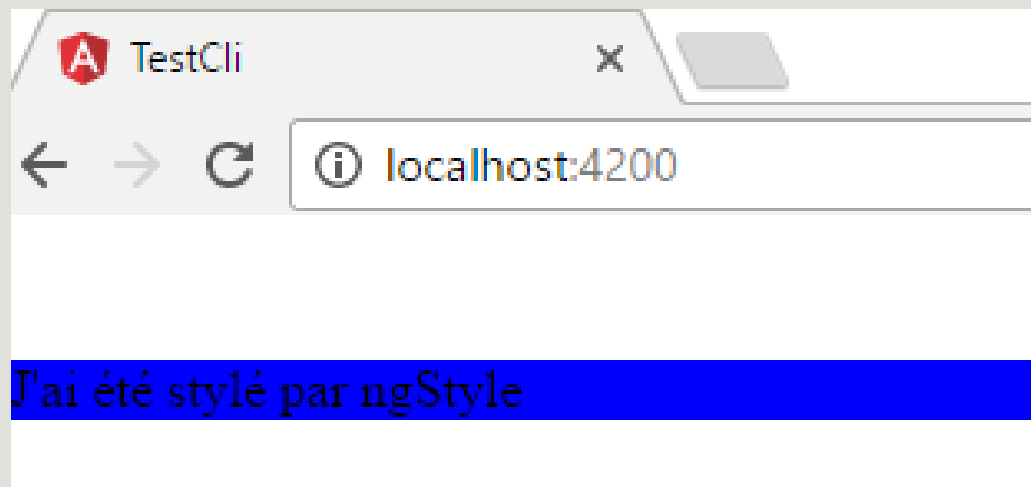
# Les directives d'attribut (ngStyle)

```
@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"
}
```

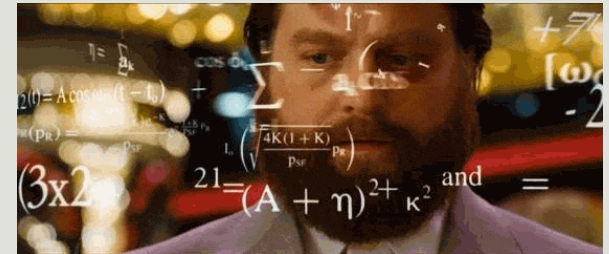
# Les directives d'attribut (ngStyle)

---



# Exercice

---



- Nous voulons simuler un Mini Word pour gérer un paragraphe en utilisant ngStyle.
- Préparer un input de type texte, un input de type number, et un select box.
- Faire 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.
- 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.

# 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: `

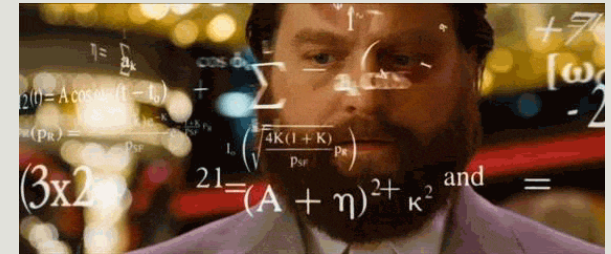
    <div ngClass="colorer arrierplan" class="encadrer">
      test ngClass
    </div>
  `,
  styles: [`
    .encadrer{ border: inset 3px black; }
    .colorer{ color: blueviolet; }
    .arrierplan{background-color: salmon; }
  `]
})
export class DirectComponent{
  private myfont:string="garamond";
  @Input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

---

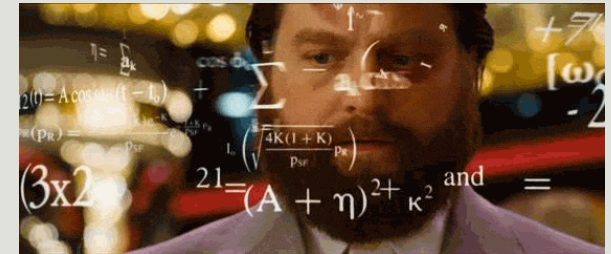


- Préparer 3 classes présentant trois thèmes différents (couleur font-size et font-police)
- Au choix du thème votre cible changera automatiquement

# 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



- Créer une directive appelée `exergue`
- Cette directive devra pouvoir faire deux choses. A l'entrée de la cible, elle devra modifier la couleur du background. A la sortie, elle devra récupérer la valeur par défaut.

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 `keydown`) 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.



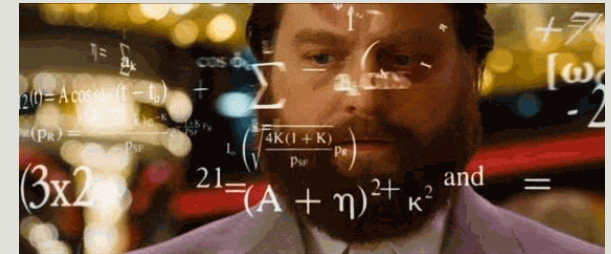
# Customiser une attribut directive

---

- 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">`

# Exercice

---



- Reprenez la directive exergue et faite en sorte qu'elle permette aussi à l'utilisateur de personnaliser ces paramètres ou d'utiliser les paramètres offerts par la directive.

# 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`
  - `[ngSwitch]`

# Les directives structurelles \*ngIf

---

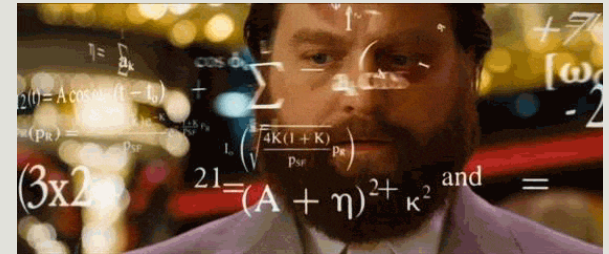
- Prend un booléen en paramètre.
- Si le booléen est true alors l'élément host est visible
- Si le booléen est false alors l'élément host est caché

Exemple

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

# Exercice

---



- Teston \*ngIf en créant un composant contenant un bouton et un paragraphe.
- Le bouton s'appellera 'Click moi'
- Un paragraphe contenant une phrase
- Au click, si le paragraphe est caché on l'affiche, s'il est affiché, on le cache

# 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

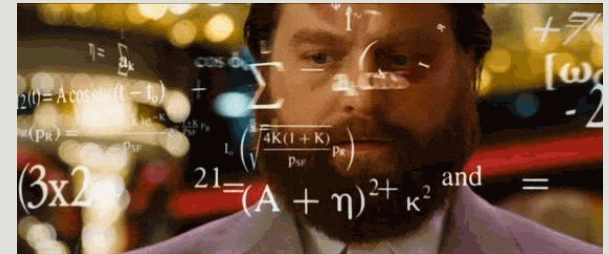
➤ odd : vrai si l'indice est impaire

```
<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>
```

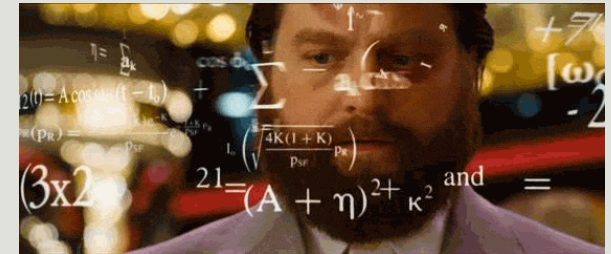
# Exercice

---

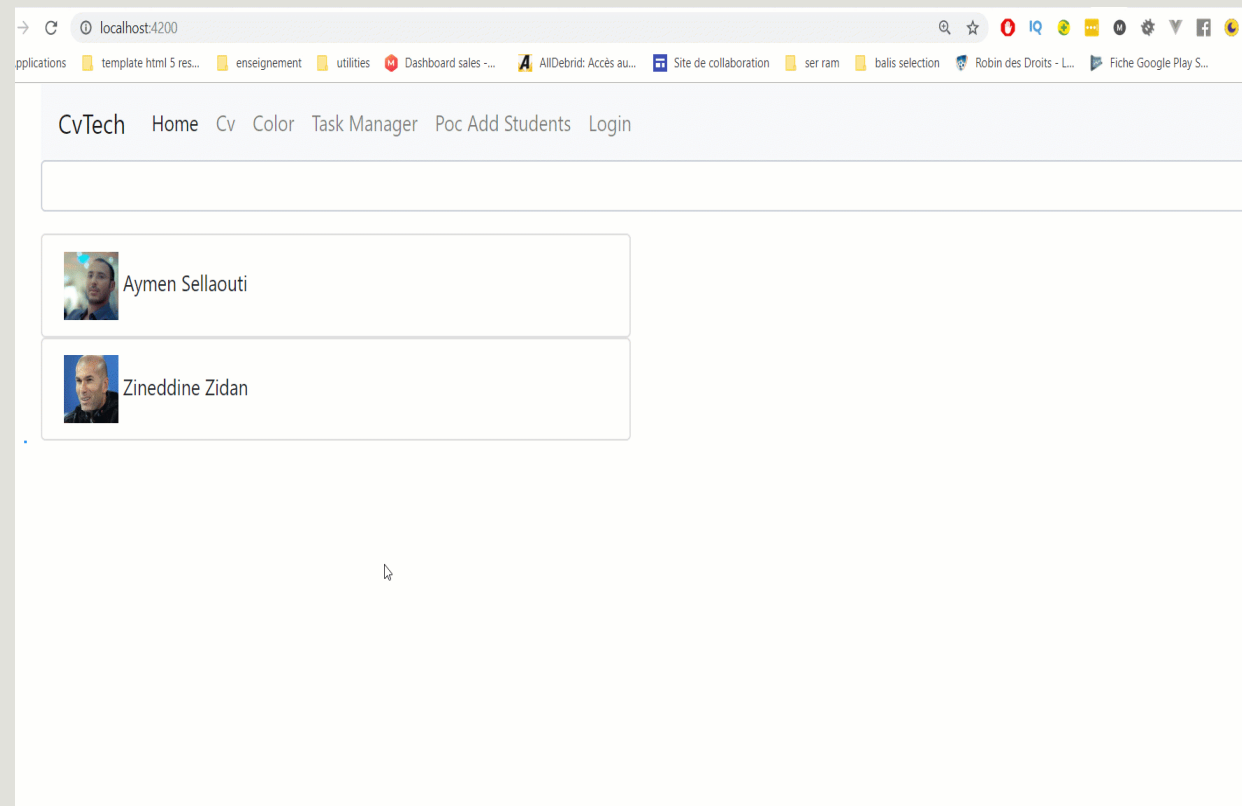


- Teston \*ngFor en créant un composant contenant un tableau d'objets personnes. Chaque personne est caractérisée par son nom, prénom, âge et métier.
- Utiliser \*ngFor pour afficher la liste de ces personnes.

# 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.





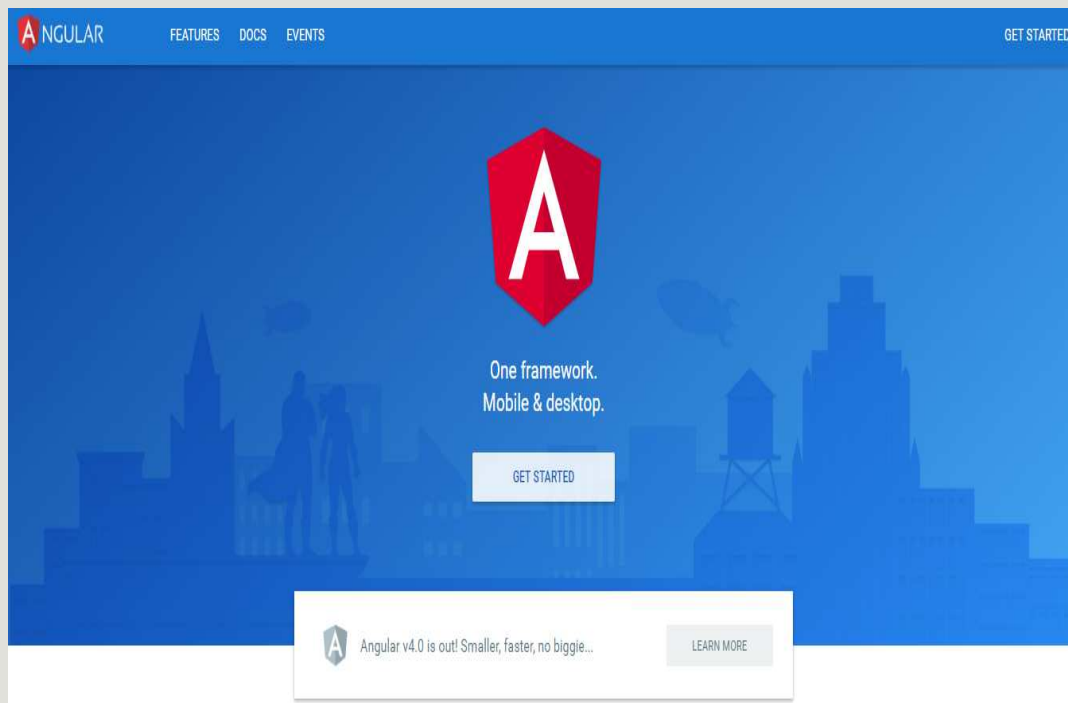
# Angular

# Les pipes

---

AYMEN SELLAOUTI

# 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
8. Les modules
9. Les tests unitaires

# 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 {

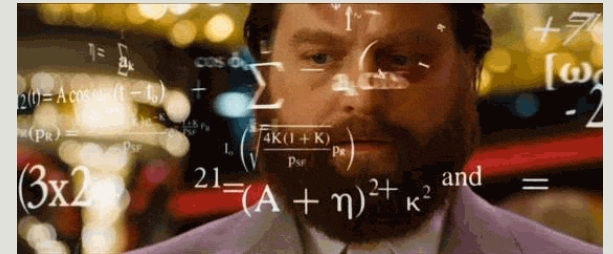
  transform(value: any, args?: any): any {
    switch (value) {
      case 'barca' : return ' blaugrana';
      case 'roma' : return ' giallorossa';
      case 'milan' : return ' rossoneri';
    }
  }
}
```

```
<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.

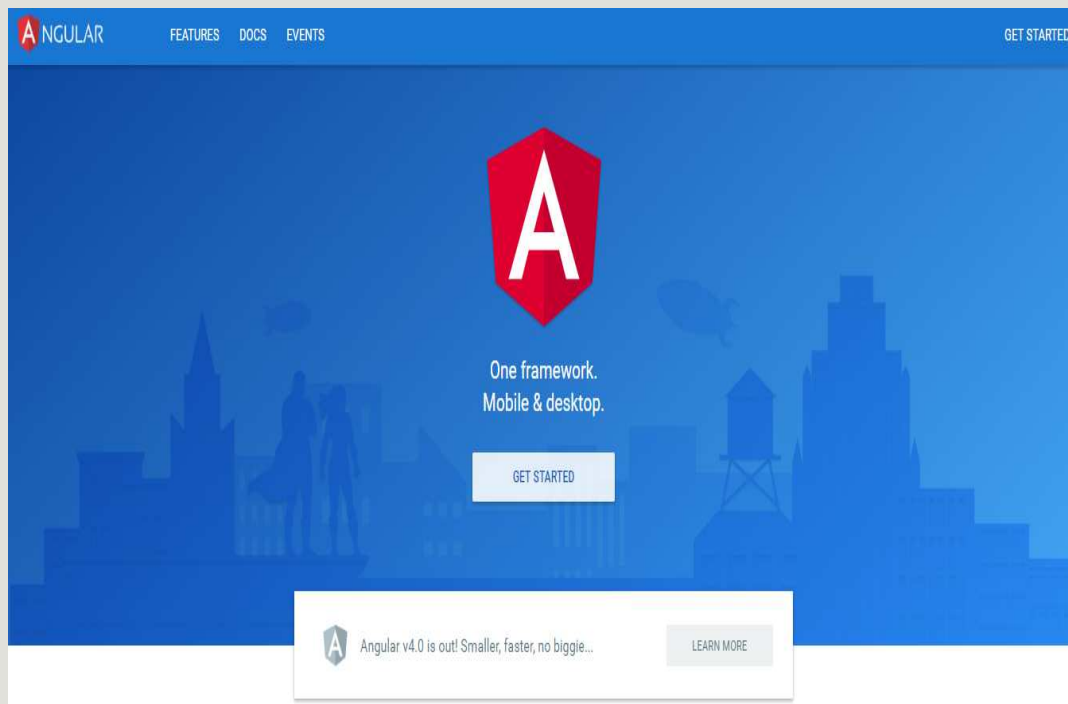
# Angular Service et injection de dépendances

---

AYMEN SELLAOUTI



# Références



# Plan du Cours

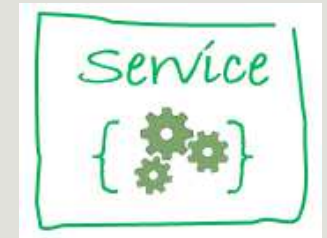
---

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
8. Les modules
9. Les tests unitaires

# 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.
- 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

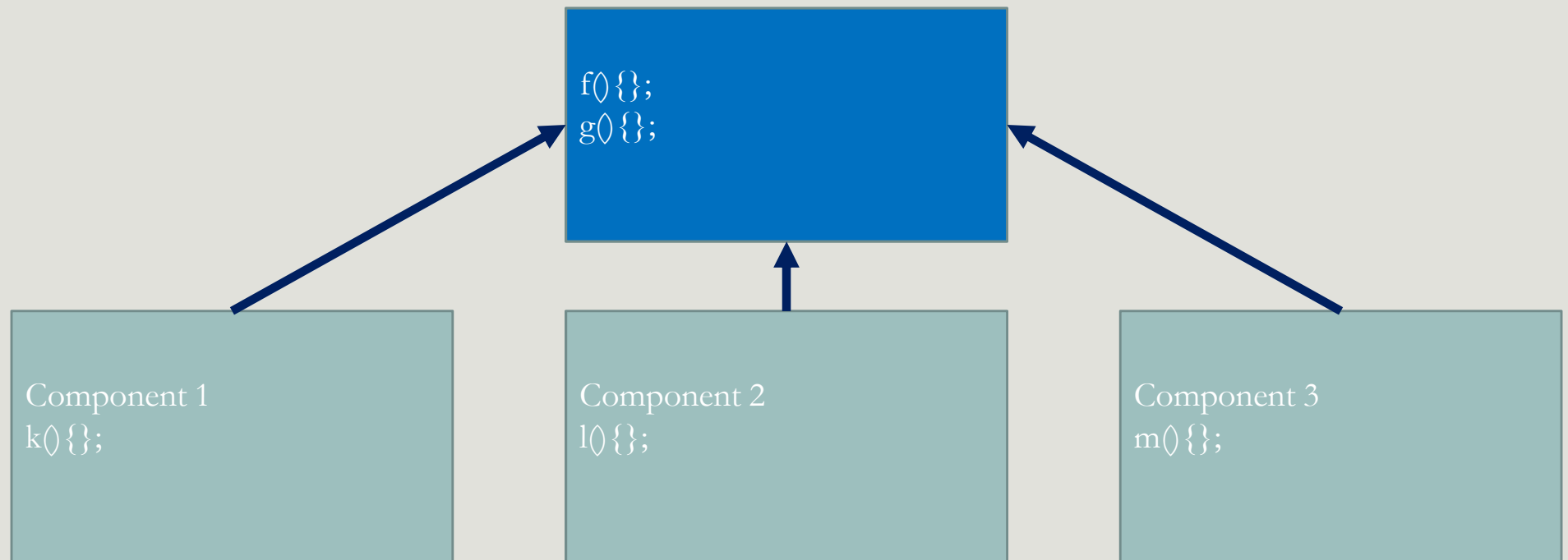
```
f(){};  
g(){};  
m(){};
```

**Redondance de code**

**Maintenabilité difficile**



# Qu'est ce qu'un service ?



# Qu'est ce qu'un service ?

---



- Un service est un médiateur 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

# Qu'est ce qu'un service ?

---



- Un service peut donc :
  - 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()
export class FirstService {

  constructor() { }

}
```

```
//...Other import
import {FirstService} from
  './first.service';

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

# 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.

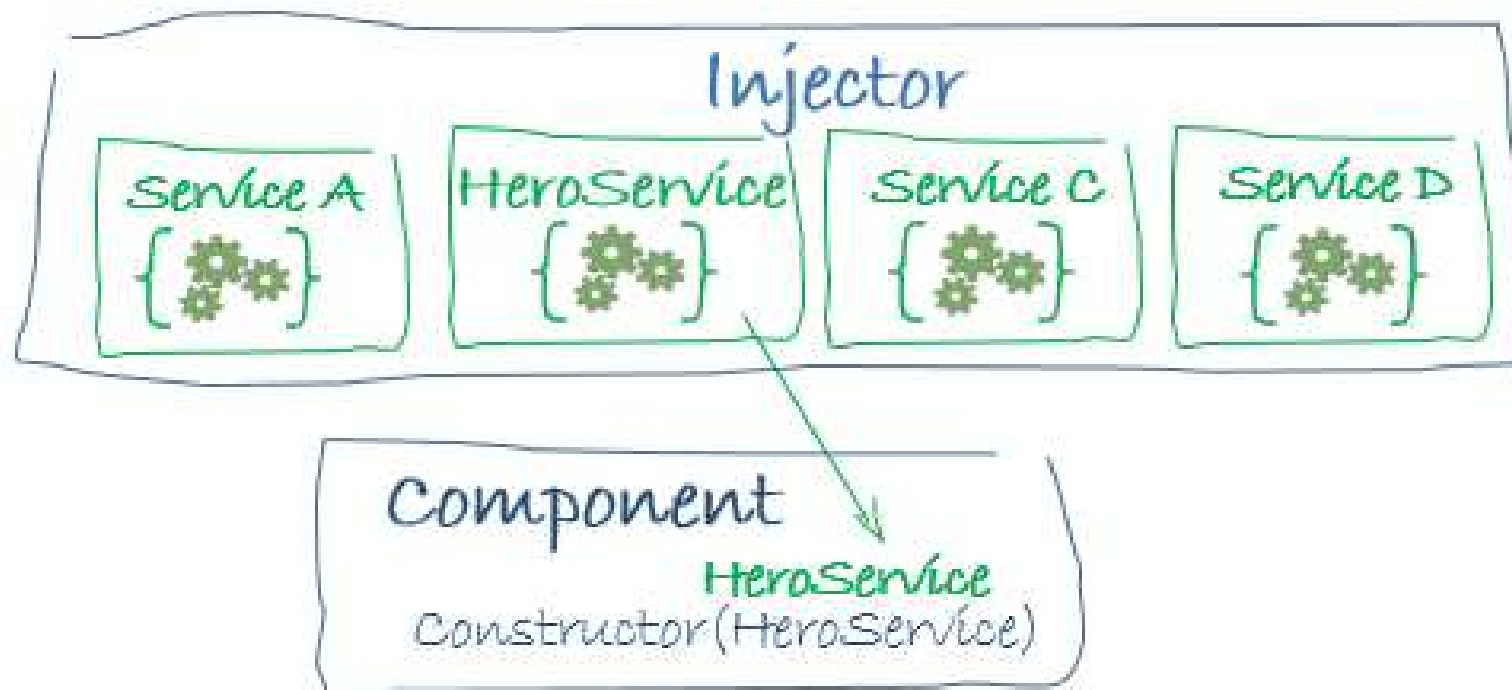
```
Classe A1 {  
  Constructor(B b, C c)  
  ...  
}
```

```
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 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() { }
}
```

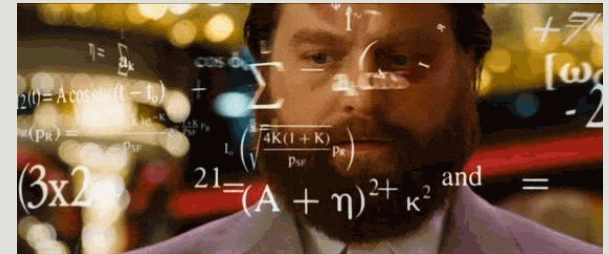
# Avantage de l'utilisation du providedIn

---

- 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.

# 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 un Todo
  - Ajouter un Todo
  - Afficher la liste des Todos
  - Supprimer un Todo

# @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.

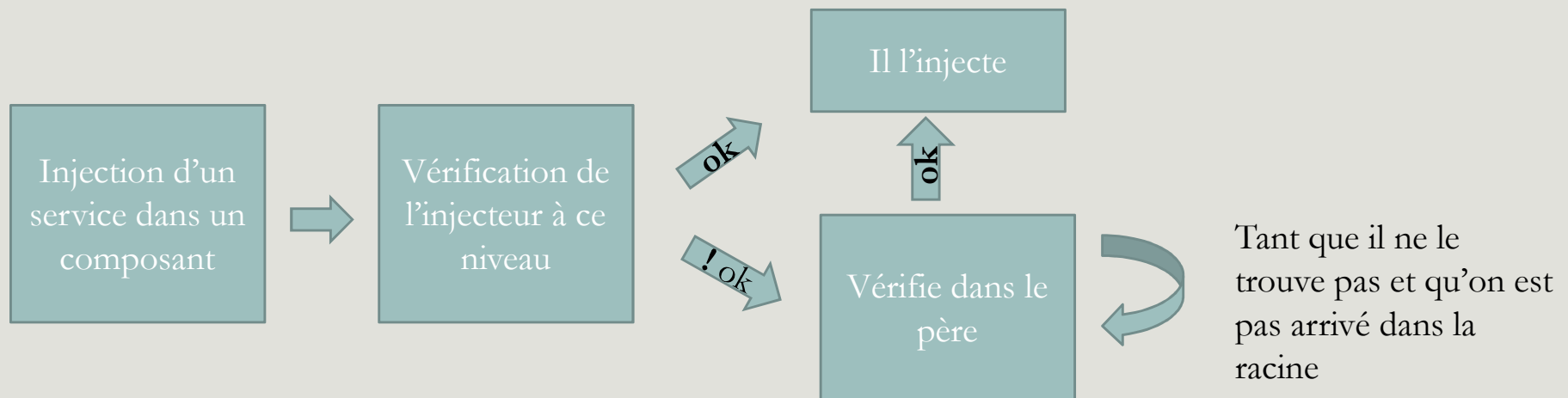


# Exemple

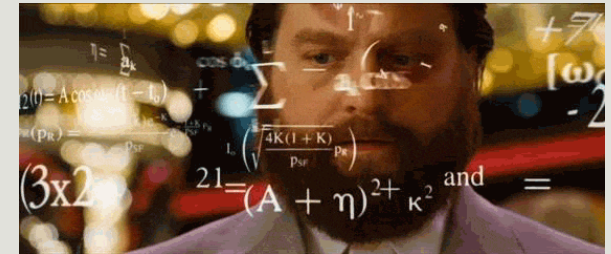
```
import { Injectable } from '@angular/core';
@Injectable()
export class LoggerService {
  constructor() { }
  Logs: string[]=[];
  logger(message:string) {
    this.Logs.push(message); console.log(message);
  }
  info(message:string) {
    this.Logs.push(message); console.info(message);
  }
  debugger(message:string)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



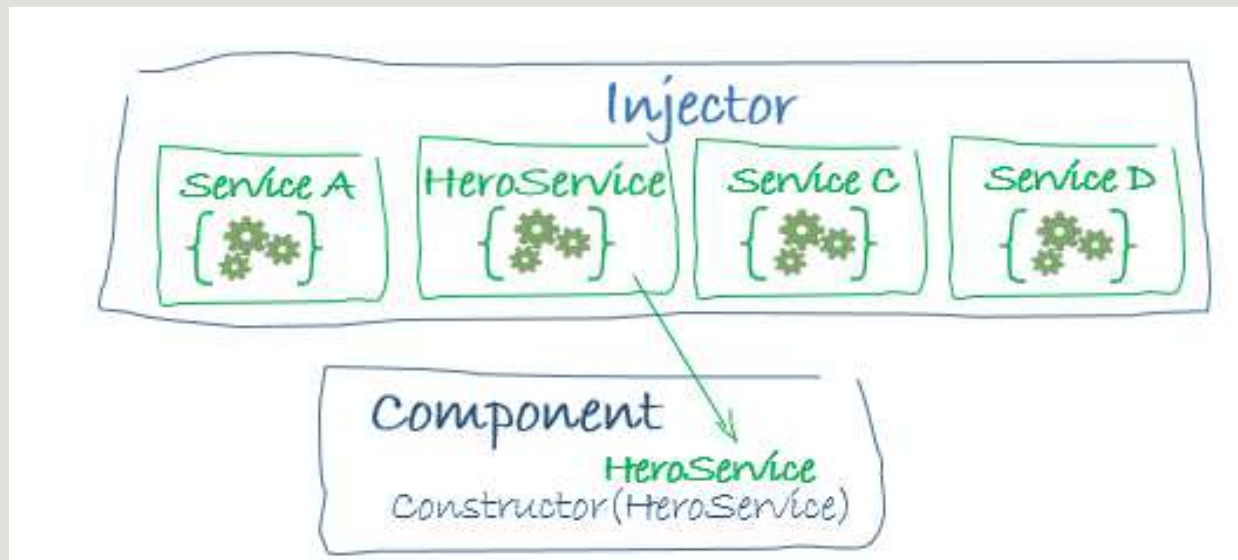
# Exercice



- Reprenez le service Logger en y ajoutant un getter qui retourne le tableau de logs
- Créer deux composants compo1 et compo2
- Injecter y le service logger
- Dans les deux composants créer un input texte et un bouton. Au click sur le bouton il va logger le contenu de l'input.
- Déclarer le directement dans le provider des deux composants.
- Vérifier est ce qu'il travaille sur la même instance du service ou non.

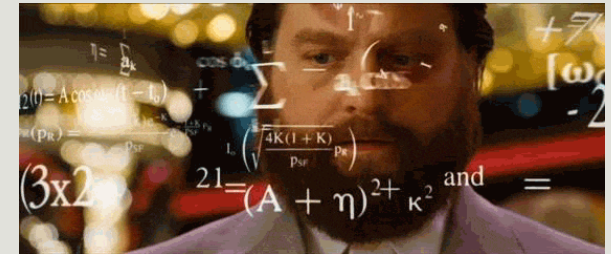
# DI Hiérarchique

- 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.



# Exercice

---



- Ajouter un troisième composant.
- Déclarer le service dans le module
- Enlever le provider du service dans le composant 1
- Garder le dans le composant 2
- Reprenez les mêmes fonctionnalités du composant 1 dans le composant 3
- Testez. Que remarquez vous?

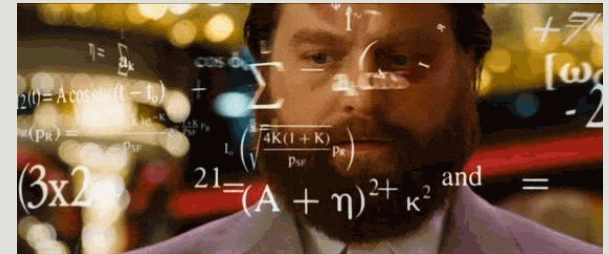
# Injecter un service dans un autre

---

- L'injection d'un service dans un autre est la même que pour un composant.
- Les seules différences sont :
  - Le service à injecter doit être visible pour le service cible.
  - Le service cible doit obligatoirement avoir la décoration `@Injectable`.

# Exercice

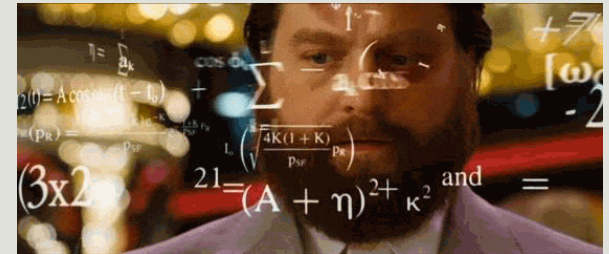
---



- Créer un nouveau service et utiliser le LoggerService dedans

# Exercice

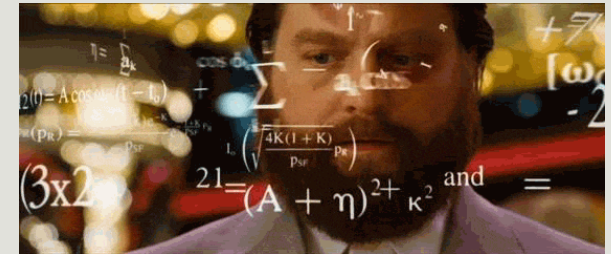
---



- 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



sellaouti aymen



sellaouti 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

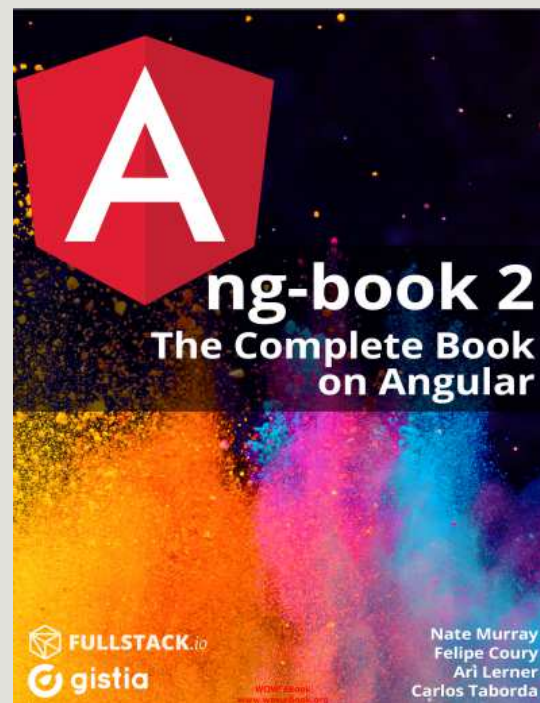
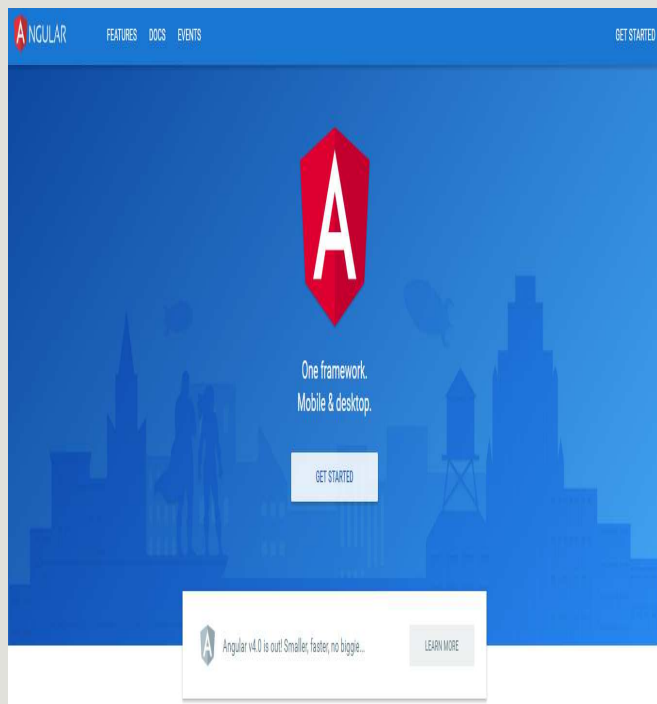


# Angular Routing

---

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
8. Les modules
9. Les tests unitaires

# 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 innexistantes

# 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

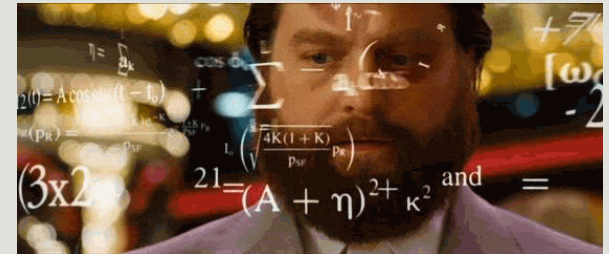
---

- Une route est un objet.
- Les deux propriétés essentielles sont `path` et `component`.
- `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 }
```

# 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

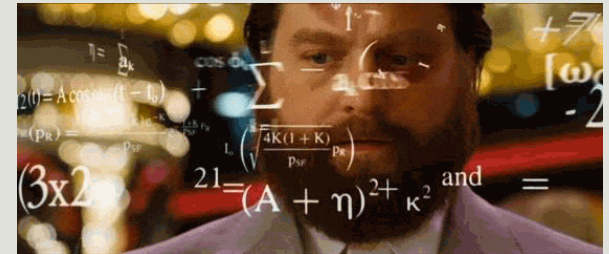
---

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

# Exercice

---



- Faites en sorte d'avoir un composant 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 l'objet **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.

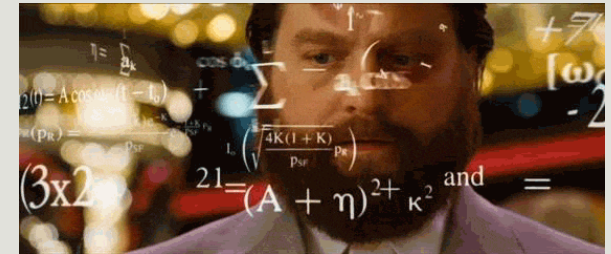
# Déclencher une route à partir du composant

---

```
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']);
  }
}
```

# Exercice

---



- 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
  - /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. Affecter le paramètre à une variable du composant en s'inscrivant avec la méthode **subscribe** à l'observable **params** de notre **ActivatedRoute**. Cette variable retourne un tableau de l'ensemble des paramètres.

Syntaxe :

```
activatedRoute.params.subscribe(params=> {this.monParam=params['param']});
```

# Récupérer les paramètres d'une route

```
import {RouterModule, Routes} from
'@angular/router';
import {HomeComponent} from
"./app/home/home.component";
import {AboutComponent} from
"./app/about/about.component";

const APP_Routes:Routes = [
  {path: '', component: HomeComponent},

  {path: 'about/:param', component: AboutCo
mponent},
]
;

export const routing =
RouterModule.forRoot (APP_Routes);
```

4

App.routing.ts

```
import { Component, OnInit } from '@angular/core';
import {ActivatedRoute} from '@angular/router';
@Component ({
  selector: 'app-about',
  templateUrl: './about.component.html',
  styleUrls: ['./about.component.css']
})
export class AboutComponent {

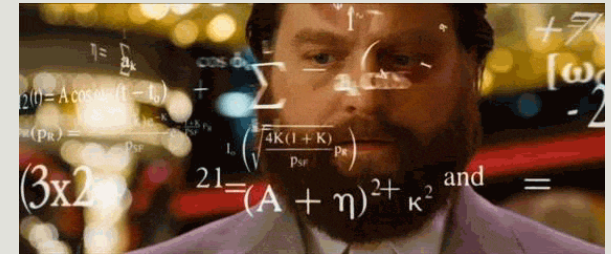
  monParam:any;
  constructor(private router:ActivatedRoute) {
    router.params.subscribe (params=>{this.monParam=par
ams['param']});
  }
}
```

1

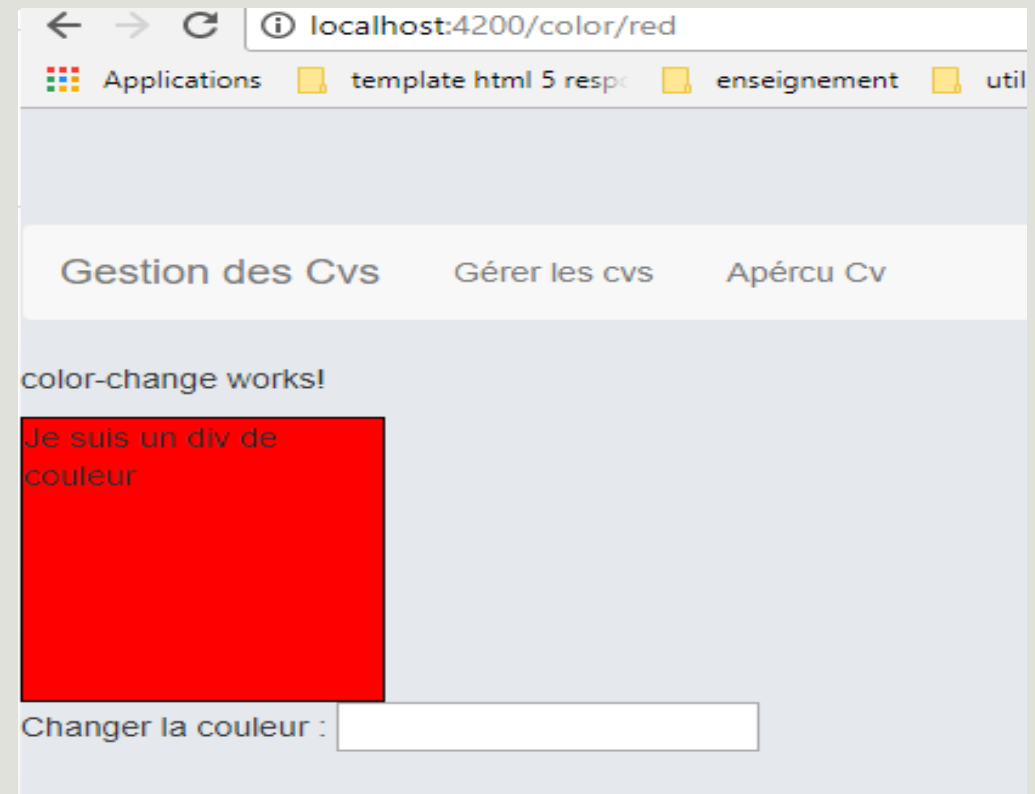
2

3

# Exercice



- Reprendre le composant qui permet de changer la couleur de la DIV
- Ajouter lui une route
- Faire en sorte que cette route soit de cette forme `/color/:couleur` et qui permettra d'affecter la couleur récupérée par la route comme couleur par défaut du DIV.



# 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;
  onNavigator() {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.
- Afin de récupérer les paramètres d'une root au niveau d'un composant on doit procéder comme suit :
  1. **Importer `ActivatedRoute`** qui nous permettra de récupérer les paramètres de la root.
  2. **Injecter `ActivatedRoute`** au niveau du composant.
  3. 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 (  
    (queryParams:any) => (this.monQp=queryParams[ 'qpVar' ] )  
);
```

# Parenthèse

---

- La méthode **subscribe** permet de s'inscrire à un **observable**.
- **Problème :** Cette souscription reste 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()**.

# Route Fils

---

- Certains composants ne sont visible 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 aimerions 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.

# Route Fils

---

- Afin de mettre en place ce processus nous procédons 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**.

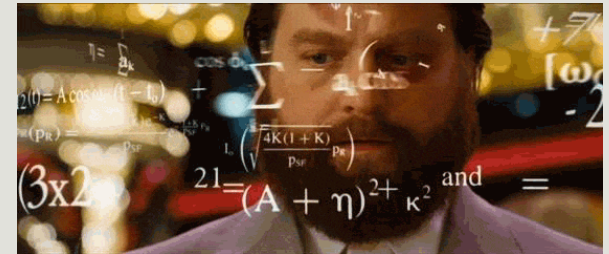
# Route Fils

---

```
const CV_ROUTE: Routes = [  
  {  
    path: 'cv',  
    children: [  
      {path: '', component: CvComponent },  
      {path: 'detail/:id', component: DetailCvComponent },  
      {path: 'addPersonne', component: FormPersonneComponent },  
    ],  
  },  
];
```

# Exercice

- Modéliser un système de routage qui utilise ses propriétés.





# Route fils / définition dans un parent

---

- Supposons que nous voulons avoir un Template central avec des données fixe 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.

# Route Fils

---

- Afin de mettre en place ce processus nous procédons comme suit :
  - Nous définissons le préfixe avec la propriété **path**. On lui associe le composant Père.
  - 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.

# Route Fils

---

```
const CV_ROUTE: Routes = [
  {
    path: 'cv',
    component: CvComponent,
    children: [
      {path: 'detail/:id', component: DetailCvComponent },
      {path: 'addPersonne', component: FormPersonneComponent },
    ]
  }
];
```

# Redirection

---

- 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.

# Redirection : exemple

---

```
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},
]
```

# Redirection : gestion d'erreurs de routing

---

- 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 '\*\*'**.

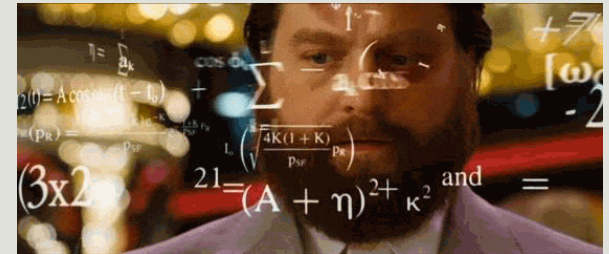
# Exemple

---

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

# Exercice

---



- Créer un composant HeaderComponent contenant votre navbar.
- Ajouter les fonctionnalités suivante à votre cvTech:
  - Une page détail qui va afficher les détails d'un cv.
  - Un bouton dans chaque cv qui au click vous envoi 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.

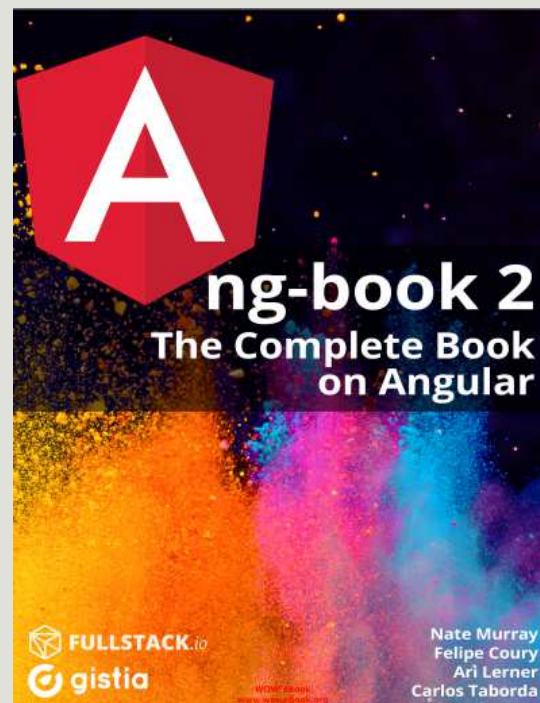
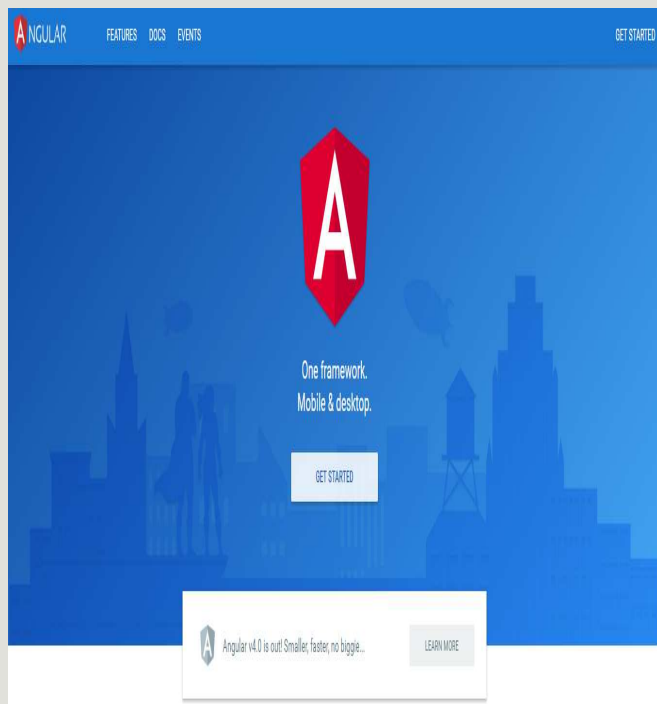


# Angular Form

---

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
8. Les modules
9. Les tests unitaires

# 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 formuliare

# 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 valid ou non en se basant sur les validateurs qu'ils contient.

[https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint\\_validation](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation)



# Approche basée Template

## NgForm

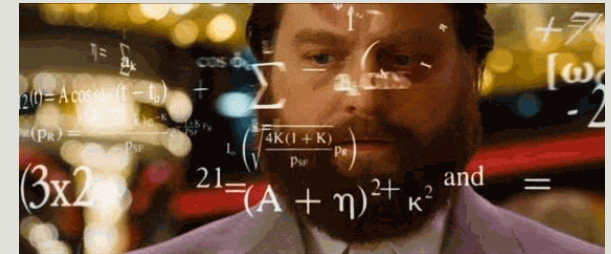
---

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` : informe si le formulaire est valide ou non
- `untouched` : 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.
- 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.

# Approche basée Template

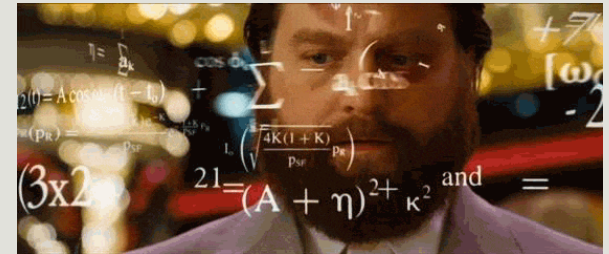
## Accéder aux propriétés d'un champ (contrôle) du formulaire

---

- 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.
- Le password devra avoir au moins 6 caractères. Ajouter un champ d'erreur pour signaler l'erreur à l'utilisateur.

# Approche basée Template

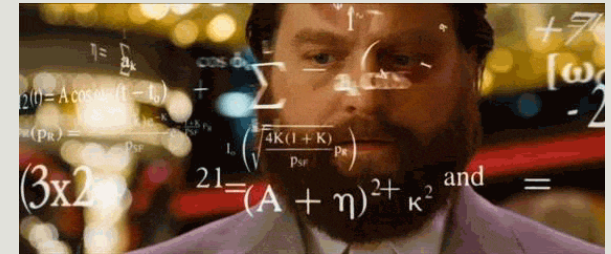
## Associer des valeurs par défaut aux champs

---

- 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 valeurs du formulaire à partir du composant il faut du binding.
- Au lieu d'avoir juste la primitive `ngModel` associé au contrôle d'un élément on ajoute le **property binding** avec `[ngModel]`

# Exercice

---



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

# Approche basée Template

## Grouping form

---

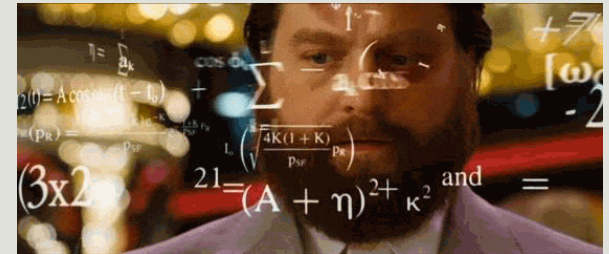
- Afin de grouper l'ensemble des contrôles (propriétés/champs) d'un formulaire, on peut utiliser la technique du « grouping form controls ».
- Il suffit d'ajouter la directive `ngModelGroup` dans la `div` qui englobe les propriétés à grouper.

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

- Afin d'accéder à cet objet vous pouvez le référencer localement en utilisant le mot clé `ngModelGroup`

# Exercice

---

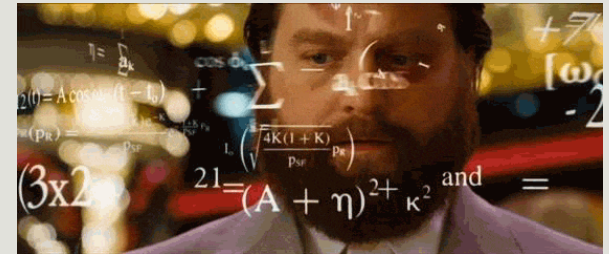


- Grouper les données de votre utilisateur dans un `ngModelGroup`
- Tester l'objet généré
- Essaye de voir s'il contient les mêmes classes qu'un contrôle simple, e.g. `ng-dirty`, `ng-valid`.
- Ajouter un message d'erreur qui apparait si votre groupe n'est pas valide.



# Exercice

---



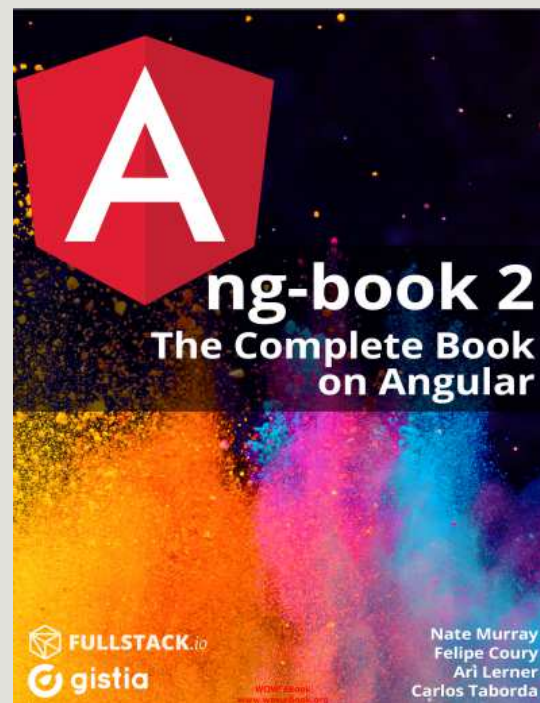
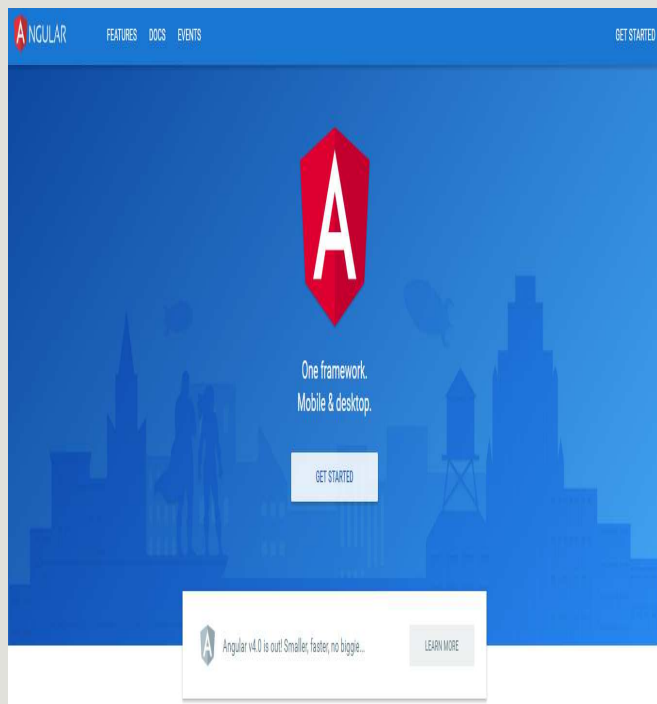
- Ajouter dans votre cvTech un composant contenant un formulaire. Ce formulaire devra vous permettre d'ajouter un utilisateur.
- Après l'ajout forwarder le user vers la liste des cvs.

# Angular HTTP et Déploiement

---

AYMEN SELLAOUTI

# 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
8. Les modules
9. Les tests unitaires

# 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 Moule 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).

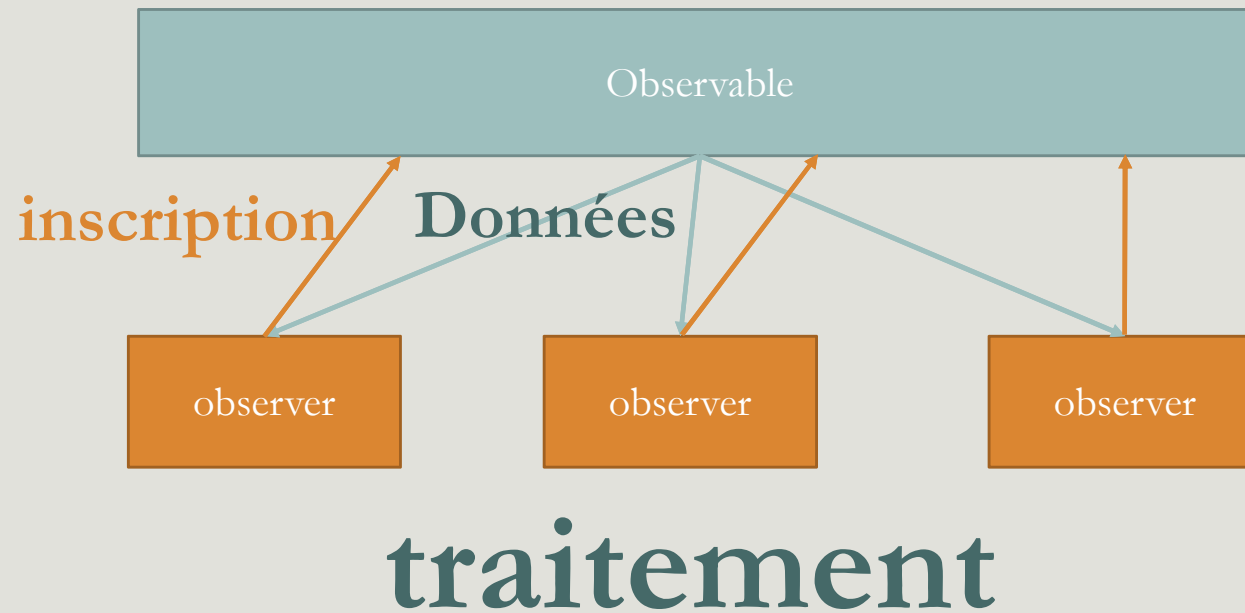
# Le pattern « Observable »

---

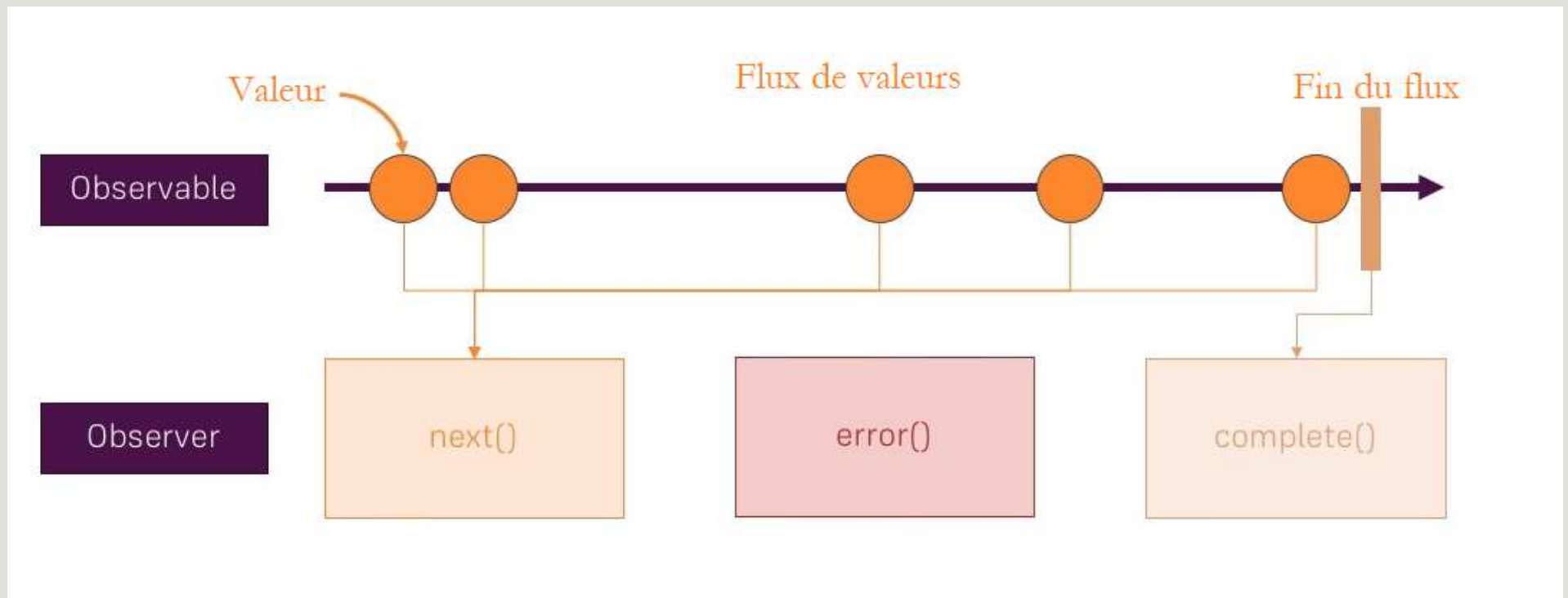
- Le patron de conception **Observable** permet à un objet de garder la trace d'autres objets, intéressés par l'état de ce dernier.
- Il définit une relation entre objets de type un-à-plusieurs.
- Lorsque l'état de cet objet **change**, il **notifie** ces **observateurs**.

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

# asyncPipe

---

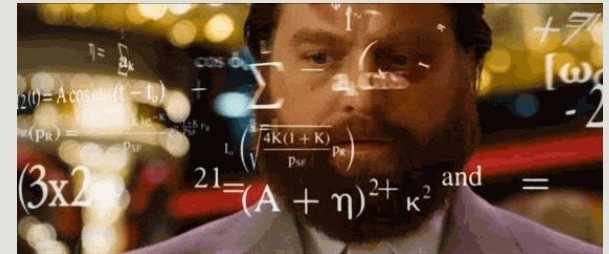
- asyncPipe est un pipe qui permet d'afficher directement un observable.
- `{{ valeurSourceAsynchrone | async }}`
- L'asyncPipe s'inscrit automatiquement à l'observable et affiche le dernier résultat envoyé.
- Quand le composant est détruit l'asyncPipe se désinscrit automatiquement de l'observable.



# Exercice

---

- Ecrire un composant qui affiche une suite d'images non stop.
- Utiliser un observable comme la source des images.



# 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**.
- 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 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.

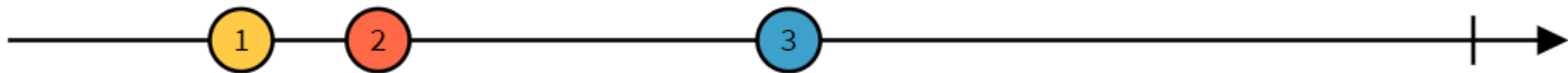
# Interagir avec une API POST Request

---

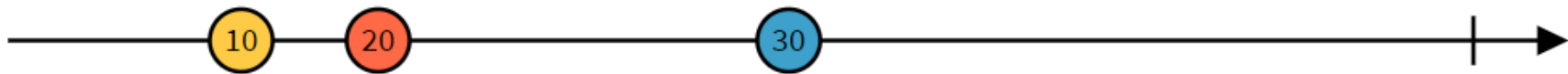
```
this.http.post(API_URL, dataToSend).subscribe(  
  (response:Response)=>{  
    //ToDo with response  
  },  
  (err:Error)=>{  
    //ToDo with error  
  },  
  () => {  
    console.log('complete');  
  }  
);
```

# Quelques opérateurs utiles de l'Observable

## map



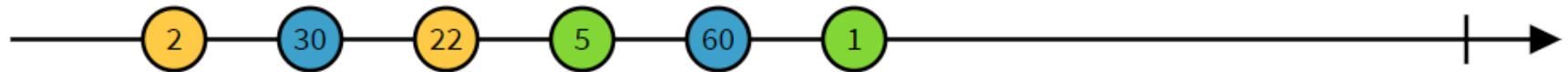
`map(x => 10 * x)`



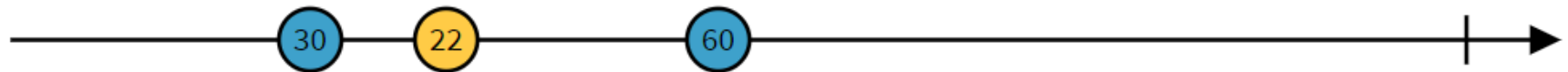


# Quelques opérateurs utiles de l'Observable

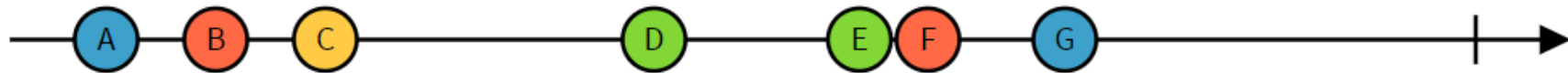
## filter



`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>

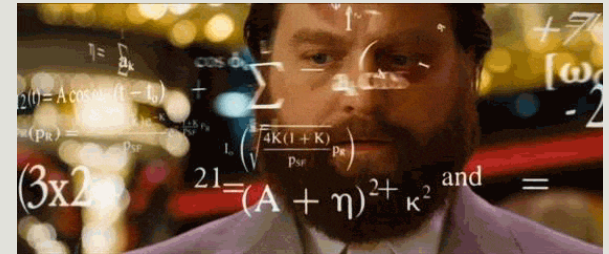
<http://rxmarbles.com/>

# 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

---

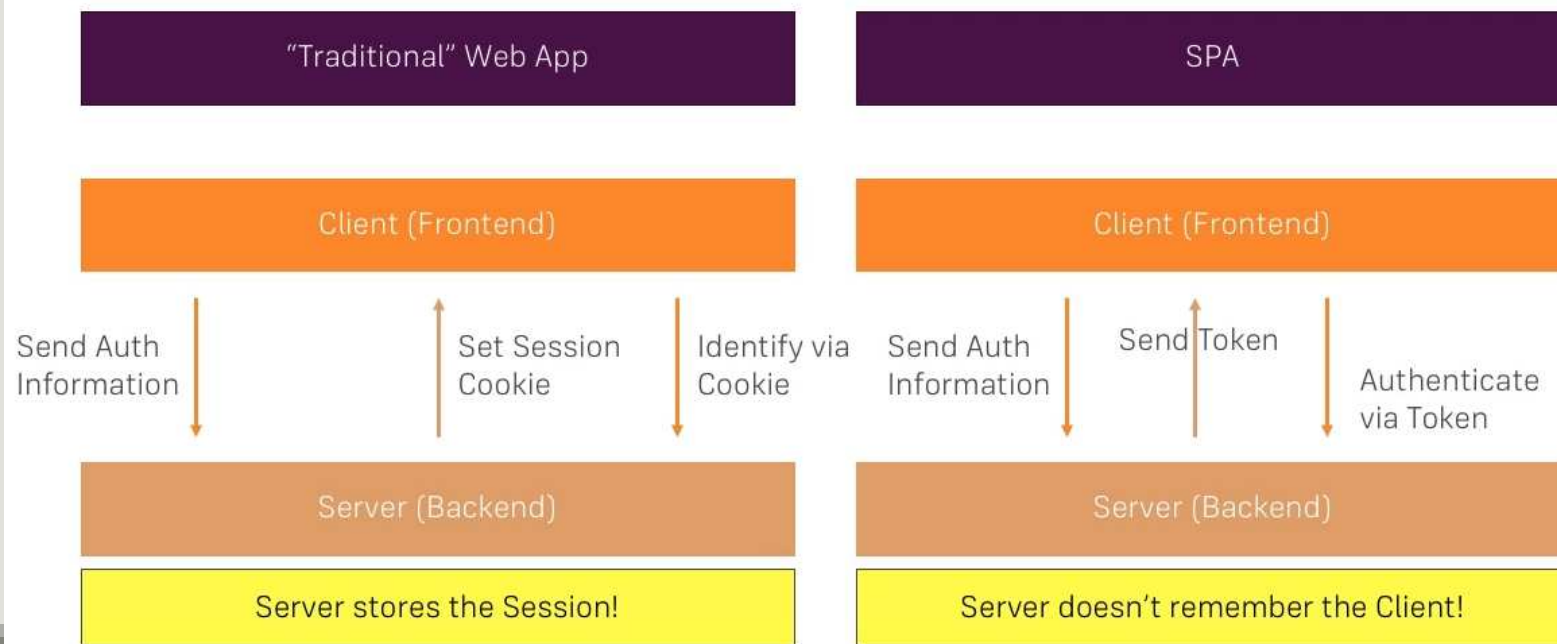
- Afin d'ajouter des paramètres à vos requêtes, le HttpClient vous offre la classe HttpParams.
- Cette classe est une classe immutable (read Only).
- Elle propose une panoplie de méthode helpers permettant de la manipuler.
- `set(clé,valeur)` permet d'ajouter des headers. Elle écrase les anciennes valeurs.
- `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?





# S'authentifier avec Loopback

---

- Loopback offre un mécanisme d'authentification prêt à l'emploi.
- Avec l'api de la classe user il vous permet d'ajouter des users et de vous connectez. Il permet aussi avec son module Loopback acl de créer des restrictions sur l'api.
- Une fois vos uri protégée, vous devez vous connecter pour pouvoir les utiliser.
- En vous connectant, il vous offrira un token. Vous devez l'utiliser à chaque appel de votre api.

# Installer loopback

---

- Nous allons installer la version lts :
- **npm install -g loopback-cli**
- Vérifier votre version avec **lb -v**
- Créer votre première application avec la commande **lb**
- Renseigner le nom de votre projet
- Renseigner le type de votre projet
- C'est fait

# Loopback : création du modèle

---

- Créer votre modèle avec **lb model**
- Suivez les étapes, ajouter votre modèle et ajouter les champs qui le compose.
- Dans notre exemple nous allons utiliser un model « Personne »

# Loopback : création du modèle

---

```
PS E:\FormaAngular\gestionPersonnes> lb model
? Entrer le nom du modèle : personne
? Sélectionner la source de données à laquelle associer personne : (n
o datasource)
? Sélectionner la classe de base du modèle PersistedModel
? Exposer personne via l'API REST ? Yes
? Forme plurielle personnalisée (utilisée pour générer l'URL REST) :
? Modèle commun ou serveur uniquement ? commun
```

# Modèle généré

- Dans ce cas nous allons utiliser une base de données MySQL avec un id en auto-increment.

```
{
  "name": "personne",
  "base": "PersistedModel",
  "idInjection": true,
  "options": {
    "validateUpsert": true
  },
  "properties": {
    "cin": {
      "type": "number",
      "required": true
    },
    "name": {
      "type": "string",
      "required": true
    },
    "firstname": {
      "type": "string",
      "required": true
    },
    "age": {
      "type": "number",
      "required": true
    },
    "path": {
      "type": "string",
      "required": true
    },
    "job": {
      "type": "string",
      "required": true
    }
  }
},
```

# Associer votre modèle à une base de données MySQL

---

- Installer le mysql-connector

```
npm install loopback-connector-mysql --save
```

- Configurer votre [datasource](#) dans le fichier [datasource.json](#) et ajouter la configuration de votre base de données mySql.
- Associer la [datasource](#) à votre [modèle](#) dans le fichier [model-config.json](#).

# Associer votre modèle à une base de données MySql

```
{
  "db": {
    "name": "db",
    "connector": "memory"
  },
  "personneDb": {
    "host": "localhost",
    "port": 3306,
    "url": "",
    "database": "test_pdo",
    "password": "",
    "name": "personneDb",
    "user": "root",
    "connector": "mysql"
  }
}
```

datasource.json

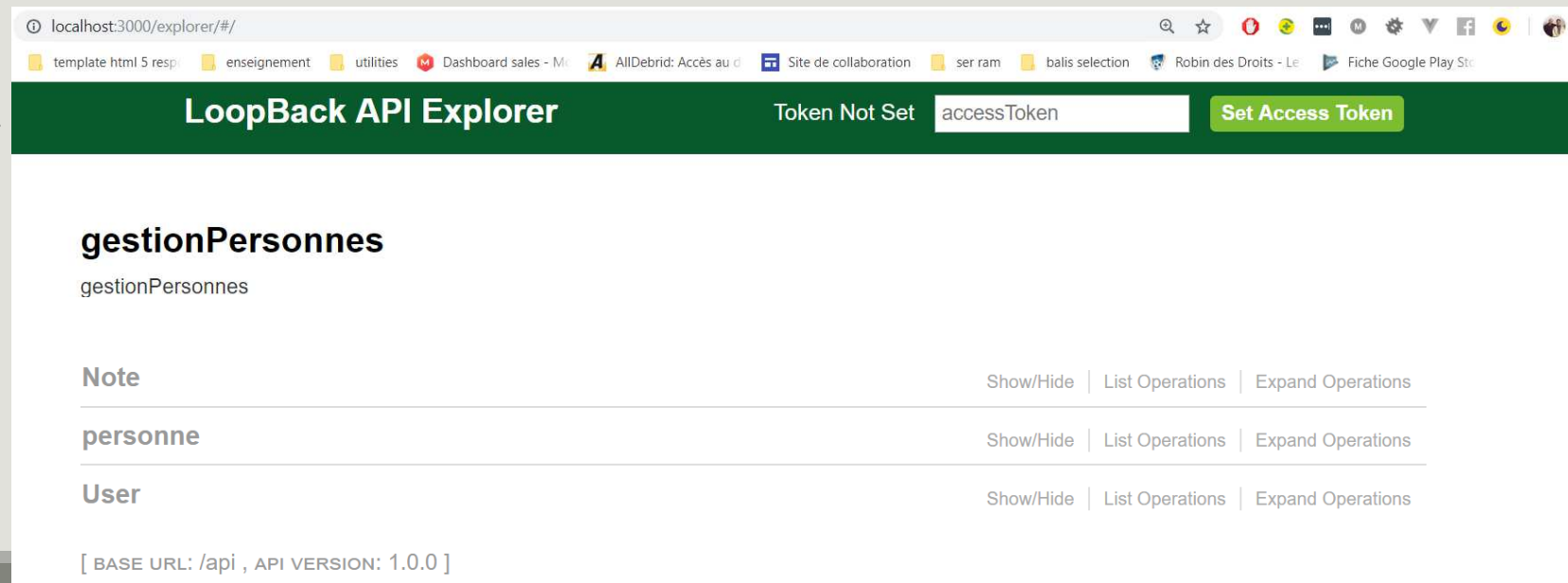
```
personneDb  
.  
.  
.  
  
"Note": {  
    "dataSource": "db"  
},  
"personne": {  
    "dataSource": "personneDb",  
    "public": true  
}
```

## Model-config.json

# Lancer votre application

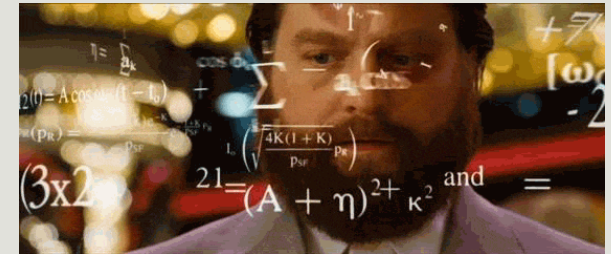
- Pour lancer votre application exécuter la commande **node** .
- Accéder à votre swagger en utilisant l'adresse <http://localhost:3000/explorer>

- Tester votre api.





# Exercice



- Associer votre application à votre API Loopback. Tester les fonctionnalités d'ajout de suppression et de modification et de sélection de l'ensemble des personnes.
- Sachant que pour sélectionner une personne dont le nom contient une chaîne donnée, loopback utilise la syntaxe suivante :  

```
{"where": {"name": {"like": "%${name}%"}}}
```
- Ceci doit être fourni dans les paramètres de votre requête avec la clé [filter](#). Tester le sur votre swagger.
- Ajouter un champ input. A chaque caractère saisie, la liste des choix doit automatiquement changer et n'afficher que les cvs qui contiennent la chaîne saisie. En sélectionnant un des choix, rediriger l'utilisateur vers les détails du cv sélectionné.

# Ajouter une acl avec loopback

---

- Nous voulons ajouter des contraintes d'accès à notre application.
- Nous voulons que seul les utilisateurs connectés puissent ajouter et supprimer des personnes.
- Ajouter les ACL en utilisant la commande `lb acl`.

```
PS E:\FormaAngular\gestionPersonnes> lb acl
? sélectionner le modèle auquel a
ppliquer l'entrée ACL : personne
? sélectionner la portée ACL : To
utes les méthodes et propriétés
? sélectionner le type d'accès :
Ecrire
? sélectionner le rôle Tous les u
tilisateurs non authentifiés
? sélectionner le droit à applicu
er Refuser explicitement l'accès
PS E:\FormaAngular\gestionPersonnes>
```

# Ajouter une acl avec loopback

---

➤ Vérifier que votre modèle a été modifié :

```
"acls": [  
  {  
    "accessType": "WRITE",  
    "principalType": "ROLE",  
    "principalId": "$unauthenticated",  
    "permission": "DENY"  
  }  
]
```

# Ajouter une acl avec loopback

---

- Essayer d'ajouter une personne avec une requête de type POST. Utiliser directement votre swagger.
- Que remarquer vous ?

# 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 qu'esi la totalité des guard retourne true. 3

# 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

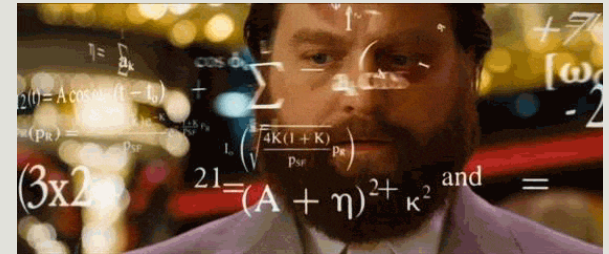
---

3

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

# Exercice

---



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

# 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 :
- Vérifier que vous avez déjà effectuer le build de votre application avec `ng build --prod`
- 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)
- <https://aymensellaouti.github.io/cvTech/> ( Cet exemple utilise un backoffice à base de loopback qu'il faudra lancer)
- En cas de mise à jour, relancer le même code.

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



---

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/",  
  }  
}
```

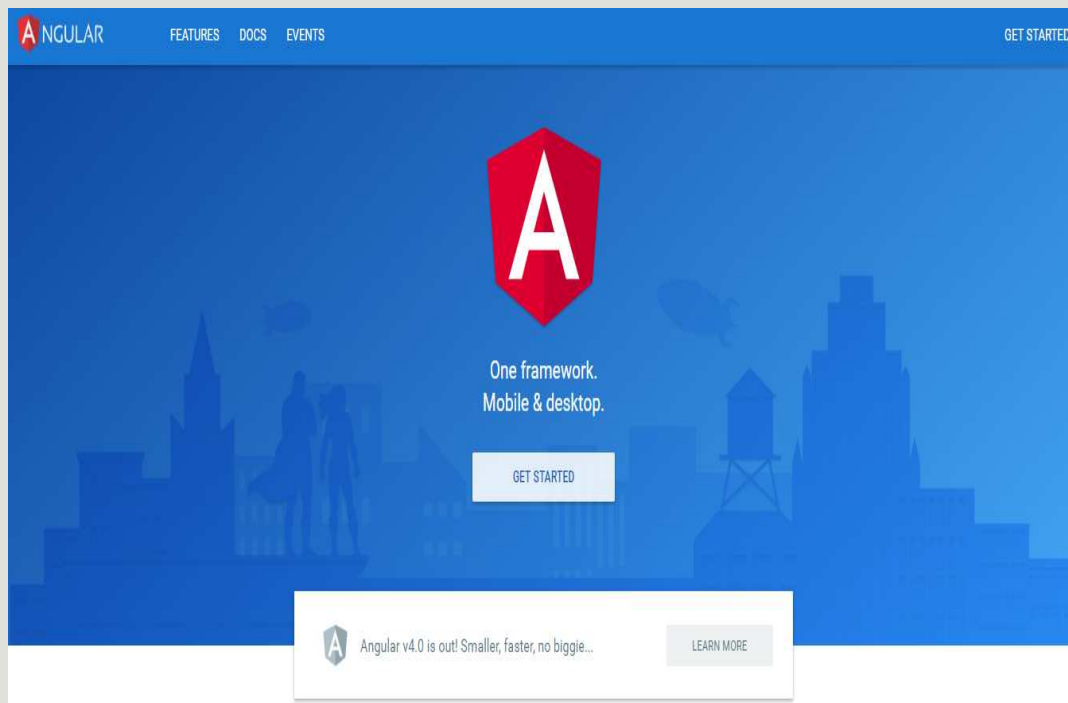
# Angular

# Les modules

---

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
8. Les modules
9. Les tests unitaires

# Qu'est ce qu'un module

---

- C'est une **classe** avec une décoration **NgModule**
- Un module est un **conteneur** qui **englobe** un **ensemble de fonctionnalités** liées.
- Les applications Angular sont **modulaire**.
- Un application Angular contient **au moins un Module** : AppModule.
- Un Module Angular peut contenir des composants, des provider de service...
- Une application simple est généralement composé d'un seul module. Par contre dès que votre application grandit penser à la séparer en Modules.

<https://angular.io/guide/architecture-modules>

# Rôle d'un module

---

- Organise votre application en des briques fonctionnelles
- Etend votre application avec des librairies externes
- Permet d'agréger et d'exporter des briques fonctionnels
- Faciliter le développement et la maintenance de votre application

# Définition d'un module

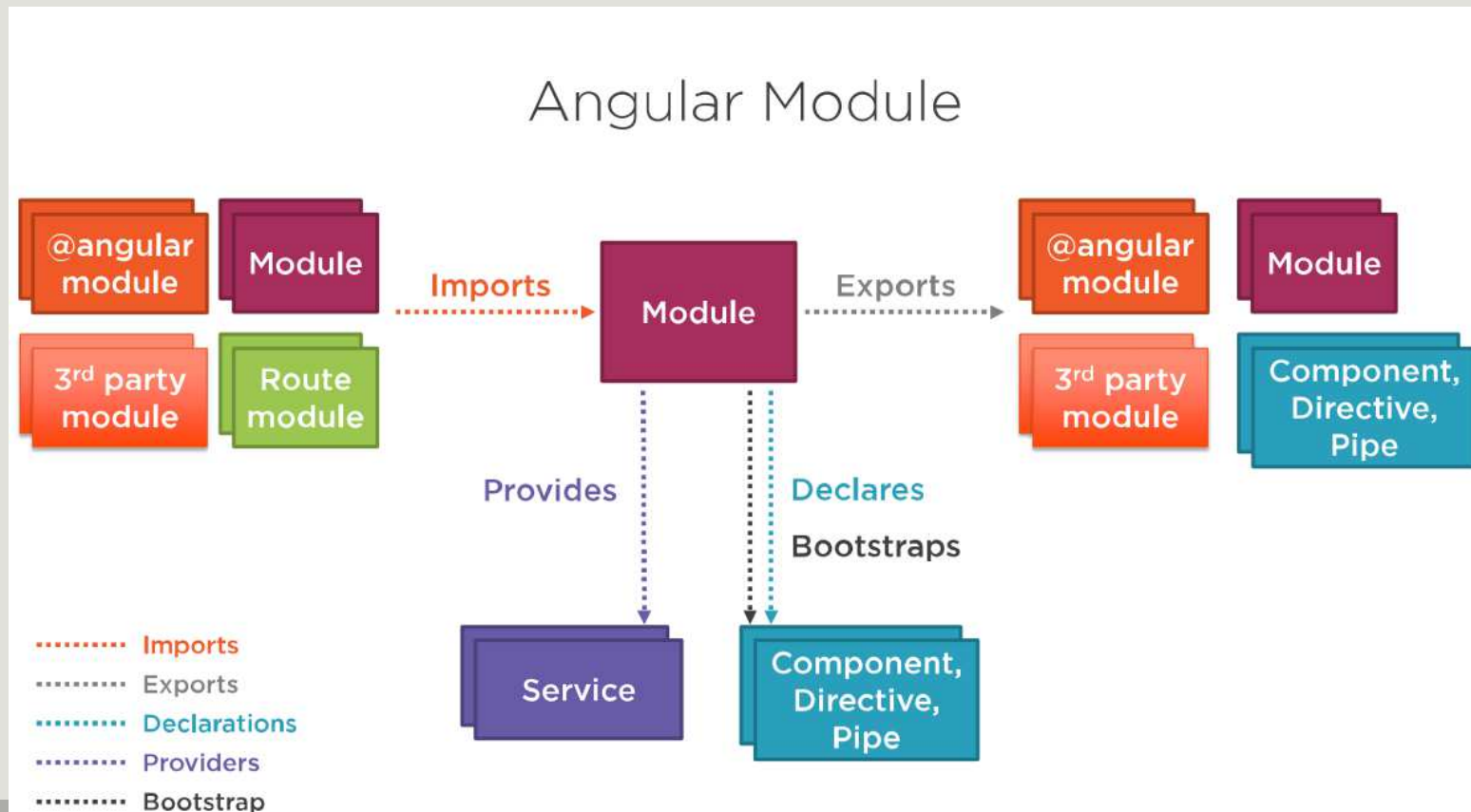
---

- L'annotation (decorator) `@NgModule` identifie 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 classes de vue appartenant à ce module, i.e. composants, directives et pipes de l'application.
  - `exports` : tableau des classes de vues à exporter.
  - `providers` : déclaration des services
  - `bootstrap` : indique le composant exécuter au lancement de l'application et elle ne concerne que le module racine.

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

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

# Définition d'un module





# declarations

---

- Dans la partie **declarations**, faite en sorte que chaque composant, directive ou pipe soit associé à **un et un seul module**. **Ne déclarer pas un même composant dans deux modules différents.**
- Ne déclarer que les composants, directives et les pipes dans cette partie.
- Tous les composants, directives et les pipes déclarés sont **privés par défaut**. Ils ne sont accessibles que pour les composants, directives et les pipes déclarés dans le même module.
- Pour utiliser un de ces éléments il faudra penser à les exporter.

# Routing

---

- Vous pouvez créer les routes de vos modules de la même manière que pour le routing de l'AppModule.
- Cependant, au lieu d'utiliser RouterModule.forRoot vous devez utiliser la méthode **forChild** du RouterModule.

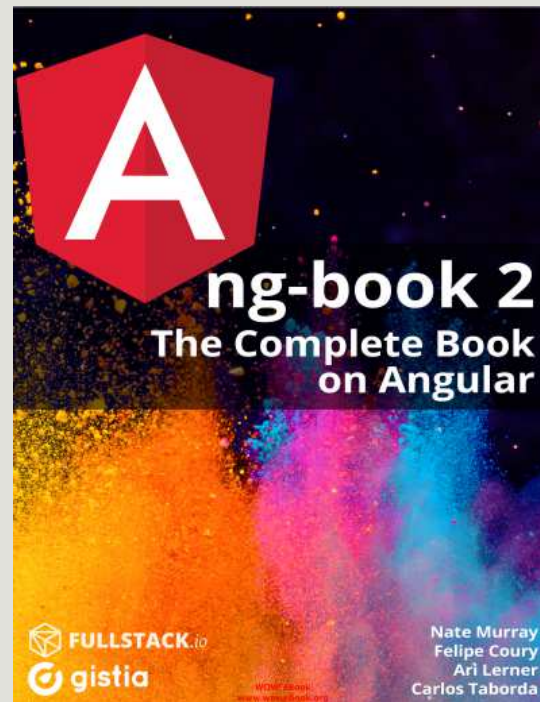
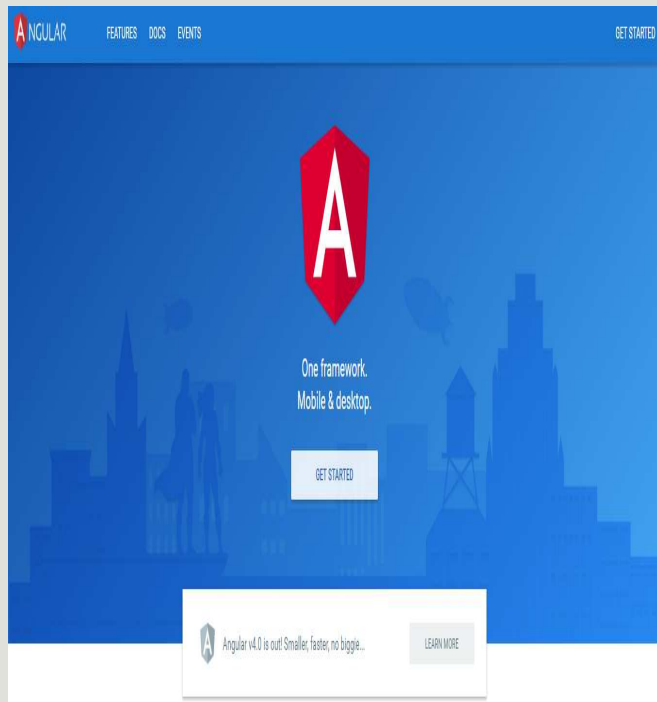
# Angular Débogage et Tests Unitaires

---

AYMEN SELLAOUTI



# Références



# Plan du Cours

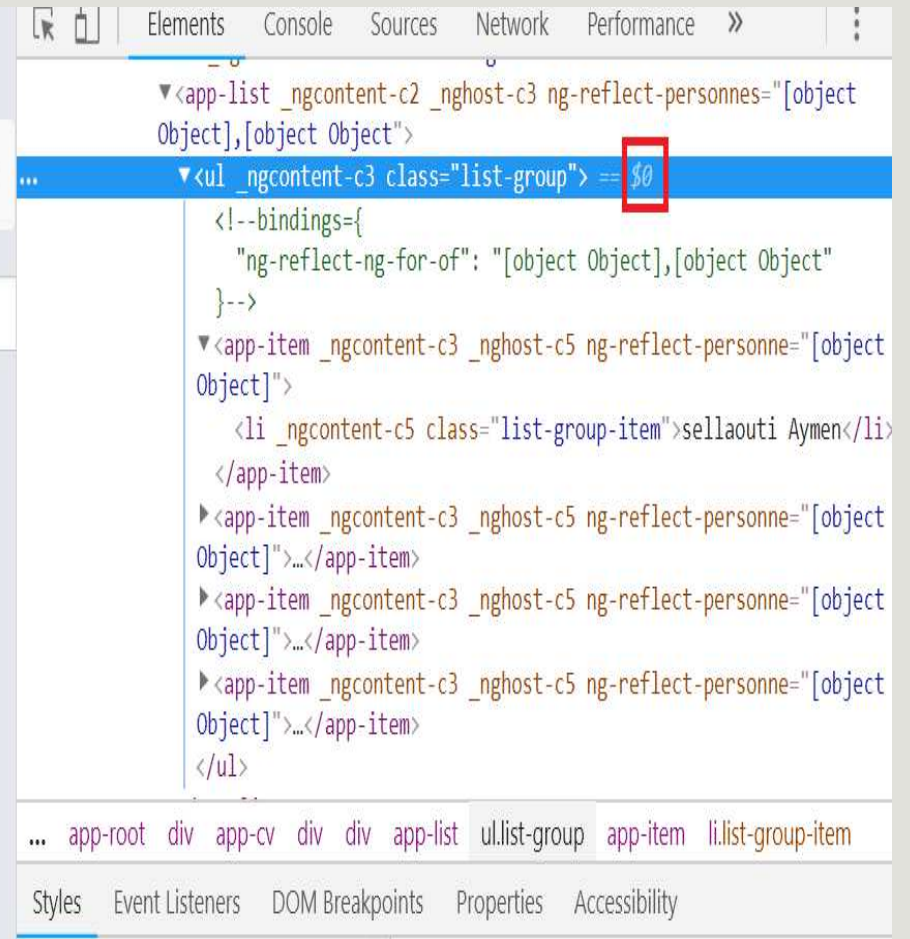
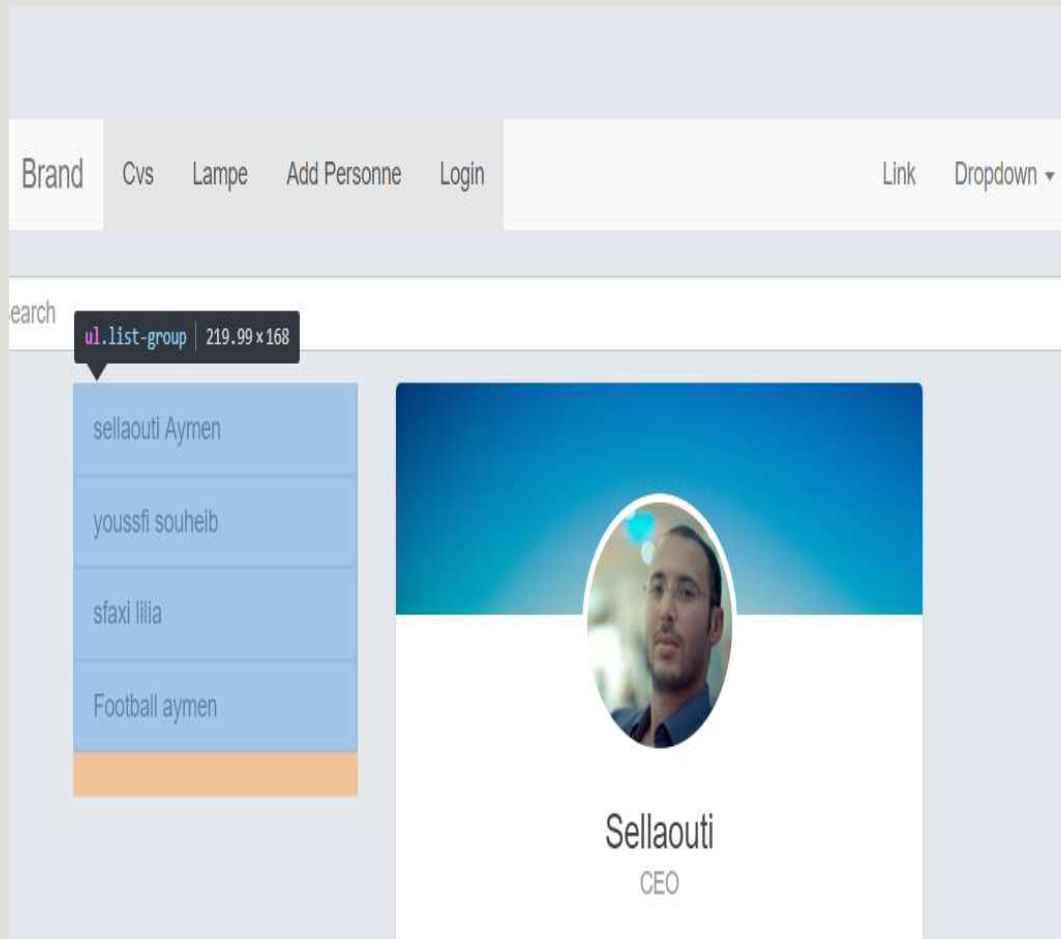
---

1. Introduction
2. Les composants
3. Les directives
4. Service et injection de dépendances
5. Le routage
6. Forms
7. HTTP Module
8. **Tests unitaires**

# Débogage

---

- Il existe plusieurs méthodes permettant de déboguer votre code Angular.
- La méthode manuelle consiste à utiliser le [DevTools](#) de votre navigateur Chrome. En [sélectionnant un élément](#), il sera [automatiquement récupérable](#) dans votre console à l'aide de la variable `$0`.
- En utilisant la méthode `ng.probe($0)` vous récupérez l'objet de type `DebugElement` de votre élément. Vous pouvez donc récupérer toutes les propriétés de [DebugElement](#) telles que l'instance de votre composant avec la propriété `componentInstance` ou l'élément du DOM correspondant avec la propriété native `Element`.
- Les variables `$1`, `$2`, `$3` et `$4` permettent de récupérer les 4 dernières variables inspectées.



\$0

► <ul\_ngcontent-c3 class="list-group">...</ul>

let de =ng.probe(\$0)  
undefined

de

▼ DebugElement {\_debugContext: DebugContext\_, nativeNode: *ul.list-group*, parent: DebugElement, listeners: Array(0), properties: {...}, ...} *i*

► attributes: {class: "list-group"}

► childNodes: (5) [DebugNode, DebugElement, DebugElement, DebugElement]

► children: (...)

► classes: {}

► componentInstance: (...)

► context: (...)

► injector: (...)

► listeners: []

► name: "ul"

► nativeElement: *ul.list-group*

► nativeNode: *ul.list-group*

► parent: DebugElement {\_debugContext: DebugContext\_, nativeNode: *app-list*, parent: DebugElement, listeners: Array(1), properties: {...}, ...}

► properties: {}

► providerTokens: (...)

► references: (...)

► styles: {}

► \_debugContext: DebugContext\_ {view: {...}, nodeId: 0, nodeDef: {...}, elDef: {...}, elView: {...}}

► \_\_proto\_\_: DebugNode

de.nativeElement

► <ul\_ngcontent-c3 class="list-group">...</ul>

de.componentInstance

▼ ListComponent {sendPersonne: EventEmitter, personnes: Array(4)} *i*

► personnes: (4) [...], {...}, {...}

► sendPersonne: EventEmitter {\_isScalar: false, observers: Array(1), closed: false, isStopped: false, hasError: false, ...}

► \_\_proto\_\_: Object



# Augury

---

- Augury est une extension de Chrome qui vous offre un ensemble d'information visuelle sur votre composant.
- Elle vous permet d'avoir des information sur les dépendance d'un composant, sur le routeur, sur l'état du composant...

# Augury : l'arbre de composant

➤ Augury vous donne un aperçu sur votre arbre de composants à travers l'onglet Component Tree

The screenshot displays an Angular application interface on the left and the Augury extension's Component Tree on the right. The application features a navigation bar with links: Brand, Cvs (highlighted), Lampe, Add Personne, Login, Link, and a Dropdown menu. Below the navigation bar is a search input field. The main content area shows a list of search results: 'sellaouti Aymen', 'youssefi souheib', 'sfaxi lilia', and 'Football aymen'. A profile card for 'Sellaouti CEO' is also visible, including a profile picture and a quote: "I'm the new Sinatra, and since I made it here I can make it anywhere, yeah, they love me everywhere".

The Augury extension's Component Tree is shown on the right, with the 'Component Tree' tab selected. The tree structure is as follows:

- AppComponent
  - HeaderComponent
    - a
    - a
    - a
    - a
  - router-outlet
  - CvComponent
    - ListComponent
      - ItemComponent
      - ItemComponent
      - ItemComponent
      - ItemComponent
      - CvCardComponent

The 'Properties' and 'Injector Graph' tabs are also visible, but the 'Properties' tab is currently selected, showing 'No component selected'.

# Augury : Etat d'un composant

- En sélectionnant un composant, Augury vous affiche son état.
- Vous pouvez changer les propriétés de l'élément directement et voir les changements

The screenshot displays the Augury Chrome extension interface, which is used to inspect the state of an Angular application. The interface is divided into several panels:

- Top Panel:** Contains navigation links: "d", "Cvs", "Lampe", "Add Personne", "Login", "Link", and "Dropdown".
- Left Panel:** A list of names: "sellaouti Aymen", "youssefi souheib", "sfaxi lilia", and "Football aymen".
- Center Panel:** Displays a profile card for "Sellaouti Aymen", CEO. The card includes a circular profile picture and a quote: "I'm the new Sinatra, and since I made it here I can make it anywhere, yeah, they love me everywhere". The card is labeled "CvCardComponent".
- Right Panel:** Contains the "Component Tree" and "Properties" panels. The "Component Tree" shows the hierarchy of components: AppComponent, HeaderComponent, router-outlet, CvComponent, ListComponent, ItemComponent, and CvCardComponent. The "Properties" panel shows the state of the selected component (ItemComponent), including a "selectItem" field with an "Emit" button and a "personne" object with the following properties: cin (551787), name (sellaouti), firstname (Aymen), age (59), path (as.jpg), job (nothing), and id (6).

# Augury : View Source et débogage

- La propriété view source permet d'accéder au code source du composant sélectionné.

The screenshot displays the Augury Chrome extension interface. On the left, a list of components is shown, including 'sellaouti Aymen', 'youssefi souheib', 'sfaxi lilia', 'Football aymen', and 'Football aua'. The central panel shows the details of the selected component, 'Youssfi', including a profile picture, name, title (CEO), and a quote. On the right, the component tree is visible, with 'AppComponent' expanded to show 'HeaderComponent' and 'CvComponent'. The 'View Source' button is highlighted in the 'ItemComponent' entry. The bottom right panel shows the state of the selected component, including 'personne' with fields like 'cin', 'name', 'firstname', 'age', 'path', 'job', and 'id'.

AppComponent

- HeaderComponent
  - a
  - a
  - a
  - a
- router-outlet
- CvComponent
  - ListComponent
    - ItemComponent
    - ItemComponent
    - ItemComponent
    - ItemComponent

Properties Injector Graph

ItemComponent (View Source) (\$\$el in Console)

Change Detection: Default

State

selectItem: Emit

personne: Object

- cin: 551787
- name: sellauti
- firstname: Aymen
- age: 59
- path: as.jpg
- job: nothing
- id: 6

Auto Rotation

Search components

# Augury : View Source et débogage

- La propriété view source permet d'accéder au code source du composant sélectionné.
- Vous pouvez insérer des breakpoints la vous souhaitez.

The screenshot displays the Augury Chrome extension interface. On the left, a list of components is shown, with 'sellaouti Aymen' selected. The center pane shows the view of the selected component, which includes a profile picture and the name 'Yousfi'. The right pane shows the source code of the selected component, with line numbers 20 and 25 highlighted. The source code is as follows:

```
6 templateUrl: './item.component.html',
7 styleUrls: ['./item.component.css'],
8 })
9 export class ItemComponent implements
10
11 @Input() personne : Personne;
12 constructor() { }
13
14 @Output() selectItem = new EventEmitter();
15 ngOnInit() {
16   console.log(this.personne);
17 }
18
19 selectPersonne(){
20   this.selectItem.emit(
21     {
22       selectItem : this.personne
23     }
24   );
25   console.log('event from item component');
26 }
27 }
28
```

# Augury : View Source et débogage

➤ En lançant votre application maintenant vous allez avoir la partie débogage.

The screenshot displays a web application with a navigation bar containing links: Brand, Cvs, Lampe, Add Personne, Login, Link, and a Dropdown menu. Below the navigation bar, there is a list of names: sellauti Aymen, youssfi souheib, sfaxi lilia, Football aymen, and Football aua. To the right of this list is a profile card for 'Youssfi' with a circular profile picture and the title 'CEO'. The Augury debugging tool is open on the right side of the screen, showing the component tree and the source code of the 'ItemComponent'. The component tree shows the following structure:

- top
  - localhost:
    - assets
    - (index)
    - main.js
    - polyfills
    - runtime
    - styles.js
    - vendor
  - (no domain)
  - Augury
  - ng://
  - webpack

The source code of the 'ItemComponent' is shown on the right, with line numbers 6 through 28. The code includes the component's template, styles, and logic for selecting a person. The following code is visible:

```
6 templateUrl: './item.component.html',
7 styleUrls: ['./item.component.css'],
8 })
9 export class ItemComponent implements OnInit {
10
11   @Input() personne : Personne;
12   constructor() { }
13
14   @Output() selectItem = new EventEmitter();
15   ngOnInit() {
16     console.log(this.personne);
17   }
18
19   selectPersonne(){
20     this.selectItem.emit(this.personne);
21   }
22   selectItem : Personne;
23 }
24
25 console.log('event from itemComponent');
```

# Augury : View Source et débogage

- En lançant votre application maintenant vous allez avoir la partie débogage.

The screenshot shows a web application interface with a navigation bar containing links: Brand, Cvs, Lampe, Add Personne, Login, Link, and Dropdown. Below the navigation bar is a search bar. The main content area displays a list of items, each with a name and a corresponding image placeholder. The items are: sellauti Aymen, youssfi souheib, sfaxi lilia, Football aymen, and Football aua. A red button labeled "Save Cvs Test" is positioned to the right of the list. The application is running in debug mode, as indicated by the "Paused in debugger" status bar. The Chrome DevTools console is open, showing the "Object" tab with the following properties: age: 59, cin: 551787, firstname: "Aymen", id: 6, job: "nothing", name: "sellaouti", path: "as.jpg", and \_\_proto\_\_: Object. The console also shows a log message: "console.log('event from ite...".

Paused in debugger

Brand Cvs Lampe Add Personne Login Link Dropdown

Search

sellauti Aymen

youssfi souheib

sfaxi lilia

Football aymen

Football aua

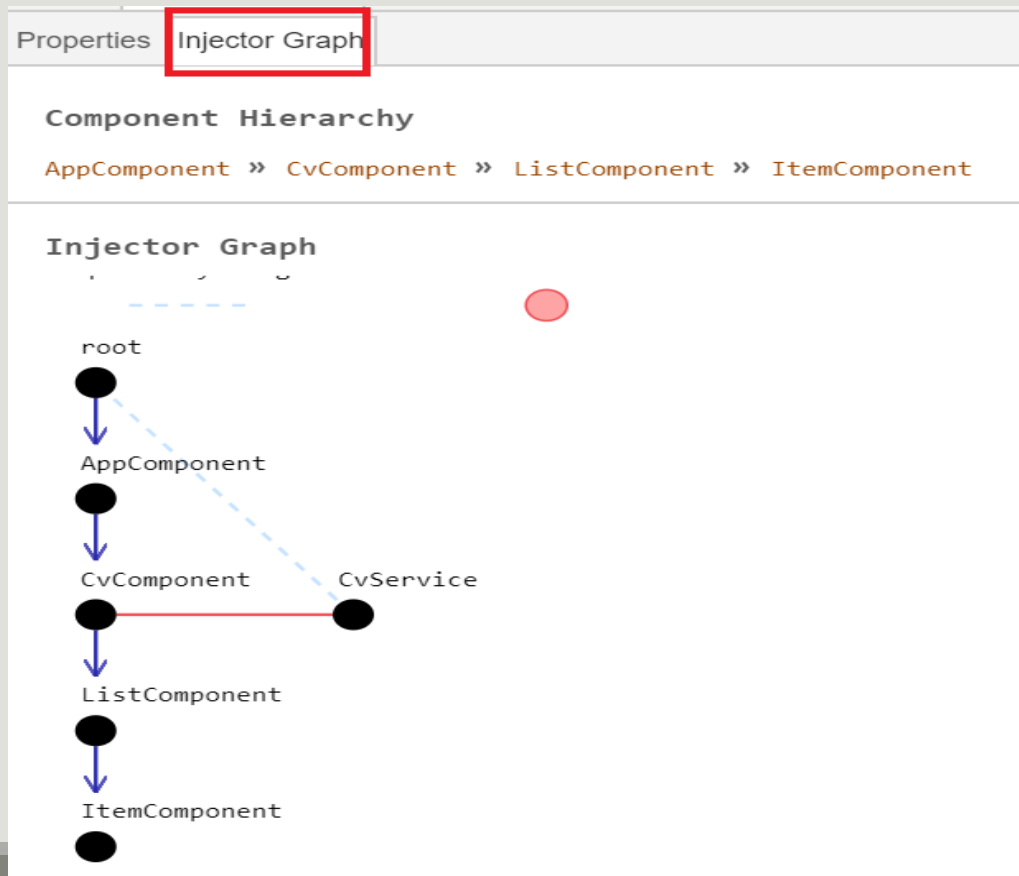
Save Cvs Test

Object

- age: 59
- cin: 551787
- firstname: "Aymen"
- id: 6
- job: "nothing"
- name: "sellaouti"
- path: "as.jpg"
- \_\_proto\_\_: Object

```
21 {  
22   selectdItem : this.person  
23 }  
24 );  
25 console.log('event from ite  
26 }  
27 }  
28 }
```

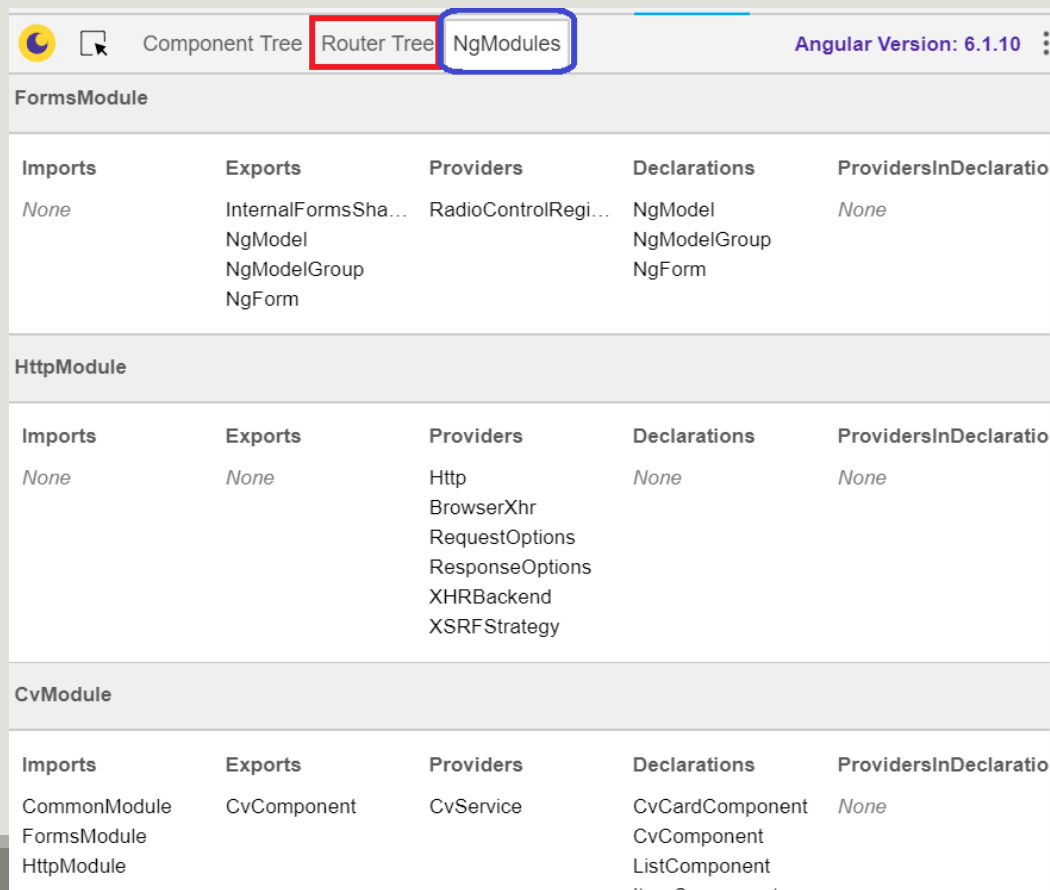
# Augury : Le graphe d'injection



- L'onglet Injector graphe permet d'afficher l'graph d'injection des différents services.



# Augury : Router tree et NgModules



Component Tree Router Tree NgModules Angular Version: 6.1.10

Imports	Exports	Providers	Declarations	ProvidersInDeclaratio
None	InternalFormsSha... NgModel NgModelGroup NgForm	RadioControlRegi...	NgModel NgModelGroup NgForm	None

HttpModule

Imports	Exports	Providers	Declarations	ProvidersInDeclaratio
None	None	Http BrowserXhr RequestOptions ResponseOptions XHRBackend XSRFStrategy	None	None

CvModule

Imports	Exports	Providers	Declarations	ProvidersInDeclaratio
CommonModule FormsModule HttpModule	CvComponent	CvService	CvCardComponent CvComponent ListComponent ItemComponent	None

- Router tree permet d'afficher les routes de l'application ainsi que la hiérarchie de ces routes.
- NgModules listes les modules et leurs dépendances.

# Tests unitaires : Introduction

---

- La plus petite unité de test possible
- Couvre une petite fonctionnalité et ne s'occupe pas de comment les différents unités testées travaillent ensemble.
- Rapide, fiable et pointe directement sur le bug en question.

# Jasmin et Karma

---

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

```
"@types/jasmine": "~2.8.6",  
"@types/jasminewd2": "~2.0.3",  
"@types/node": "~8.9.4",  
"codelyzer": "~4.2.1",  
"jasmine-core": "~2.99.1",  
"jasmine-spec-reporter": "~4.2.1",  
"karma": "~1.7.1",  
"karma-chrome-launcher": "~2.2.0",  
"karma-coverage-istanbul-reporter": "~2.0.0",  
"karma-jasmine": "~1.1.1",  
"karma-jasmine-html-reporter": "^0.2.2",
```

# Jasmin et Karma

---

➤ **Jasmin** est un framework permettant de faciliter la création de tests. Il contient un ensemble de fonctionnalités permettant d'écrire plusieurs types de test.



**karma** est un task runner pour vos tests. Il utilise un fichier de configuration afin de gérer le process de test en identifiant les fichiers de chargement, le framework de test, le navigateur à lancer...

# Lancement d'un 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.

ng test

# Configuration de Karma

La configuration de karma se trouve dans le fichier 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'),  
      reports: ['html', 'lcovonly'],  
      fixWebpackSourcePaths: true  
    },  
    reporters: ['progress', 'kjhtml'],  
    port: 9876,  
    colors: true,  
    logLevel: config.LOG_INFO,  
    autoWatch: true,  
    browsers: ['Chrome'],  
    singleRun: false  
  });  
};
```



Framework à lancer



Plugins à charger



Format du rapport



Lancer le test en mode watch  
Navigateur

# Concepts de base de jasmine

---

- **describe (string, function)** : fonction qui prend en paramètre un titre et une ensemble de test individuel.
- **it (string, function)** : fonction représentant un **test** individuel qui prend en paramètre un titre et une fonction définissant un test individuel.
- **expect** : fonction qui retourne un booléen et évalue une expectation un besoin à valider par le test unitaire.

Exemple **expect**(etatActuel).**toBe**(etatExpecté)

- **les matchers** : sont des helpers prédéfinis permettant différentes validations.

Exemple : toBe, ToEqual, toContain, toBeDefined, toBeTruthy, toBeLessThan, toBeGreaterThan

<https://github.com/JamieMason/Jasmine-Matchers>

# Concepts de base de jasmine

---

Jasmin offre des handlers permettant de répéter certaines fonctionnalités.

- **beforeEach** : prend en paramètre une **callback** et la **répète** avant chaque spec **it**.
- **afterEach** : prend en paramètre une **callback** et la **répète** après chaque spec **it**.
- **beforeAll** : prend en paramètre une **callback** et la **répète** avant chaque suite **describe**.
- **afterEach** : prend en paramètre une **callback** et la **répète** après chaque suite **describe**.



# Concepts de base de jasmine

---

- **x**it permet d'exclure un test individuel
- **x**describe permet d'exclure tout le bloc
- **f**it permet de spécifier le test individuel à exécuter
- **f**describe permet de spécifier le bloc à exécuter.

# Tester une classe

---

- 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.

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

@Injectable()
export class AuthenticationService {
  authenticated = false;
  constructor() { }
  isAuthenticated(): boolean {
    this.authenticated = !! localStorage.getItem('user');
    return this.authenticated;
  }
}
```

# Tester une classe

---

- La seule 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('AuthenticationService', () => {  
  it('should be created', () => {  
    const service = new AuthenticationService();  
    expect(service).toBeTruthy();  
  });  
  it('it should return true if there is a user in the localStorage ', () => {  
    const service = new AuthenticationService();  
    localStorage.setItem('user', 'aymen');  
    expect(service.isAuthenticated()).toBeTruthy();  
  })  
});
```

# Tester une classe

---

- Finaliser le test de cette classe.
- Essayer de factoriser vos tests

```

import { AuthenticationService } from './authentication.service';

let service;
describe('AuthenticationService', () => {
  beforeEach(() => {
    service = new AuthenticationService();
  });
  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', 'aymen');
    expect(service.isAuthenticated()).toBeTruthy();
  });
  it('it should return false if there is not a user in the localStorage ', () =>
  {
    expect(service.isAuthenticated()).toBeFalsy();
  });
});
}

```

# Tester un pipe

---

- Un pipe contient uniquement la méthode transform. Tester un pipe revient donc à tester la méthode transform.
- Soit le pipe suivant :

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

@Pipe({
  name: 'lower'
})
export class LowerPipe implements PipeTransform {
  transform(value: string, args?: any): any {
    return value.toLowerCase();
  }
}
```

# Tester un pipe

---

```
import { LowerPipe } from './lower.pipe';
let pipe;
describe('LowerPipe', () => {
  beforeEach(() => {
    pipe = new LowerPipe();
  });
  it('create an instance', () => {
    expect(pipe).toBeTruthy();
  });
  it('should transform input to lowercase', () => {
    expect(pipe.transform('BonJouR')).toEqual('bonjour');
  });
});
```

# Angular TestBed

---

- **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 propriétés de **@NgModule**.
- Prenant l'exemple de test d'un service, il vous faudra l'ajouter dans le provider avant de le tester.



# Angular TestBed

---

```
beforeEach(async() => {  
    TestBed.configureTestingModule({  
        declarations: [ AuthenticationComponent ],  
        providers: [ AuthenticationService ]  
    })  
});
```

# Component fixtures

---

- 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 multitude d'attributs et de méthodes très utiles pour tester un composant.

<https://angular.io/api/core/testing/ComponentFixture>

# debugElement / nativeElement

---

- 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 récupérer une référence sur l'élément DOM du template.

# componentInstance

---

- 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 :

```
fixture = TestBed.createComponent(AppComponent);  
const myCompo = fixture.componentInstance;
```

# Premier Test d'un composant

---

En installant votre projet Angular vous avez un test avec votre premier composant

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

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

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

# Récupérer un service injecté avec TestBed

---

➤ Afin de récupérer un service injecté, **TestBed** offre la méthode **get** qui prend en paramètre la classe du service injecté et retourne une instance.

➤ Exemple `TestBed.get(MonService);`

# Tester un composant

---

- Un composant est une combinaison entre un Template (votre code HTML) et le métier qui le gère (cotre classe TypeScript).
- En testant votre composant, vous devez tester l'adéquation entre le Template et sa classe TypeScript et la bonne communication entre les deux.
- Nous devons donc 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.

# Exercice

---

- Créer un composant Authentication
- Dans ce composant on va avoir deux div.
  - La première div apparaîtra si le user est authentifié.
  - La seconde sinon.

```
<div *ngIf="isLoggedIn">
  <h1>User Logged</h1>
</div>
<div *ngIf="!isLoggedIn">
  <h1>User not Logged</h1>
</div>
```

```
@Component ({
  selector: 'app-authentication',
  templateUrl: './authentication.component.html',
  styleUrls: ['./authentication.component.css']
})
export class AuthenticationComponent {
  isLoggedIn = true;
  constructor(private authenticationService:
AuthenticationService) { }
  ngOnInit() {
  }
  login() {
    this.isLoggedIn =
this.authenticationService.isAuthenticated();
  }
}
```



# Exercice

---

1. Créer les tests nécessaires à votre Composant.

```

import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { AuthenticationComponent } from '../authentication.component';
import { DebugElement } from '@angular/core';
import { AuthenticationService } from '../authentication.service';
describe('AuthenticationComponent', () => {
  let component: AuthenticationComponent;
  let fixture: ComponentFixture<AuthenticationComponent>;
  let compiled: any;
  let service: AuthenticationService;
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ AuthenticationComponent ],
      providers: [AuthenticationService]
    })
  }));
  beforeEach(() => {
    fixture = TestBed.createComponent(AuthenticationComponent);
    component = fixture.componentInstance;
    compiled = fixture.debugElement.nativeElement;
    service = TestBed.get(AuthenticationService);
  });
  afterEach(() => {
    localStorage.removeItem('user');
  });
  it('should create', () => {
    expect(component).toBeTruthy();
  });
  it('should display User logged when user is logged', () => {
    localStorage.setItem('user', 'aymen');
    component.login();
    fixture.detectChanges();
    expect(compiled.querySelector('h1').textContent).toContain('User Logged');
  });
  it('should display User not logged when user is not logged', () => {
    component.login();
    fixture.detectChanges();
    expect(compiled.querySelector('h1').textContent).toContain('User not Logged');
  });
});

```

# Isolation des tests

---

```
it('should display User logged when user is logged', () => {  
  localStorage.setItem('user', 'aymen');  
  component.login();  
  fixture.detectChanges();  
  expect(compiled.querySelector('h1').textContent).toContain('User Logged');  
});
```

Si la méthode login ne vérifie plus le localStorage ? Que doit on faire ?

On doit réécrire notre test même si la méthode retourne toujours un booléen et agit donc de la même façon

Le test n'est pas isolé, si on garde la même façon de travailler on va avoir des tests très couplés.

# Simuler (mock) vos classes

---

- Afin de remédier au problème d'isolation et de couplage, vous devez simuler vos classes (mock).
- Trois manières sont possible pour mocker vos classes :
  1. Créer une classe Mock simulons le fonctionnement de vos classes cibles.
  2. Etendre les classes cibles et redéfinir les méthodes à utiliser. Ceci est utile lorsque vous avez des classes assez complexes.
  3. Utiliser le **spyOn** de **Jamin**.

# Mock avec une fake classe

---

- L'idée est de créer une Fake classe qui simule le fonctionnement de notre service et qui retourne ce que nous attendons comme résultat.
- Nous pouvons complètement séparer notre test en supprimant le service. Notre test devient donc isolé.

# Mock avec une fake classe : Etapes

---

- Créer votre MockClass
- Surcharger le composant pour qu'il prenne en considération la nouvelle classe :

```
TestBed.overrideComponent(  
  AuthenticationComponent,  
  { set: { providers: [{ provide: AuthenticationService, useClass: MockAuthenticationService }] }  
);
```

- récupérer le nouveau service

```
serviceMock = fixture.debugElement.injector.get(AuthenticationService);
```

- Vous pouvez maintenant utiliser votre nouveau service

# Mock avec une fake classe : Exemple

```
import { async, ComponentFixture, TestBed } from
'@angular/core/testing';
import { AuthenticationComponent } from
'./authentication.component';
import { AuthenticationService } from '../authentication.service';
class MockAuthenticationService {
  auth = false;
  isAuthenticated() {
    return this.auth;
  }
}
describe('AuthenticationComponent', () => {
  let component: AuthenticationComponent;
  let fixture: ComponentFixture<AuthenticationComponent>;
  let compiled: any;
  let serviceMock: AuthenticationService;
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ AuthenticationComponent ],
      providers: [AuthenticationService]
    });
    TestBed.overrideComponent(
      AuthenticationComponent,
      { set: { providers: [{ provide: AuthenticationService, useClass:
MockAuthenticationService }] } }
    );
  }));
```

```
beforeEach(() => {
  fixture = TestBed.createComponent(AuthenticationComponent);
  component = fixture.componentInstance;
  compiled = fixture.debugElement.nativeElement;
  serviceMock = fixture.debugElement.injector.get(AuthenticationService);
});
afterEach( () => {
  serviceMock = null;
  compiled = null;
  component = null;
  fixture = null;
})
});

it('should create', () => {
  expect(component).toBeTruthy();
});
it('should display User logged when user is logged', () => {
  serviceMock.auth = true;
  component.login();
  fixture.detectChanges();
  expect(compiled.querySelector('h1').textContent).toContain('User Logged');
});
it('should display User not logged when user is not logged', () => {
  component.login();
  fixture.detectChanges();
  expect(compiled.querySelector('h1').textContent).toContain('User not Logged');
});
});
```

# Mocker un service avec spyOn

---

- Afin de simuler le fonctionnement d'une méthode, jasmine vous offre les spy et plus particulièrement spyOn.
- spyOn prend en paramètre l'instance à intercepter et le nom de la méthode à simuler.
- L'une des méthode la plus utilisées est .and.returnValue qui permet de retourner une valeur lorsque l'on intercepte la fonction.

Exemple `spy =`  
`spyOn(service, 'isAuthenticated').and.returnValue(false);`  
retourne false lorsque l'on appelle la méthode isAuthenticated du service.



# Mocker un service avec spyOn

---

```
it('should display User logged when user is logged', () => {
  // serviceMock.auth = true;
  spy = spyOn(service, 'isAuthenticated').and.returnValue(true);
  component.login();
  fixture.detectChanges();
  expect(compiled.querySelector('h1').textContent).toContain('User
Logged');
});
it('should display User not logged when user is not logged', () => {
  spy = spyOn(service, 'isAuthenticated').and.returnValue(false);
  component.login();
  fixture.detectChanges();
  expect(compiled.querySelector('h1').textContent).toContain('User not
Logged');
});
```

# Autres fonctionnalités de spyOn

---

`and.callFake(fn)` : exécute la fonction `fn` appelé

`and.throwError(e)` : throw l'erreur `e`

`calls.count()` combien de fois la méthode a été appelé

<https://jasmine.github.io/2.0/introduction.html>

# Teste Asynchrone avec `async` et `whenStable`

---

`Async` est un outil offert par Angular afin de gérer les tests asynchrone.

Il suffit d'englober votre test avec `async` et d'ajouter une méthode `fixture.whenStable` qui retourne une promesse.

En utilisant la méthode `.then` de votre promesse mettez y le code asynchrone à tester

# Teste Asynchrone avec async et whenStable

---

```
it('should have a async', async(() => {  
  let value;  
  const p = new Promise(  
    resolve => {  
      resolve(10);  
    }  
  );  
  expect(value).toEqual(undefined);  
  p.then(  
    (x) => {  
      value = x ;  
    }  
  );  
  fixture.whenStable().then(  
    () => {  
      expect(value).toEqual(10);  
    }  
  );  
}));
```

# Teste Asynchrone avec fakeAsync

---

```
it('should have a value workflow when observable  
finish', fakeAsync(() => {  
  let value;  
  setTimeout(() => value = 10, 1000);  
  expect(value).toEqual(undefined);  
  tick(1000);  
  expect(value).toEqual(10);  
}));
```

tick(x : ms) : simule le passe du temps x ms.

# Tester du HTTP

---

- Un appel HTTP est un appel asynchrone. Cependant nous n'avons aucune idée sur la durée de la réponse. Une des solutions est de simuler un appel asynchrone et de mocker le service en utilisant cet appel.

Exemple d'appel asynchrone simulé :

```
function asyncData<T>(data: T) {  
  return defer(() => Promise.resolve(data));  
}
```

Nous utilisons ici la méthode `defer` de `rxjs` qui permet de créer un `Observable` uniquement que un observateur s'inscrit. Elle crée un observable à chaque inscription. Dans notre cas on veut simuler un observable http qui retourner une réponse à un GET.

# Tester du HTTP

---

- Nous créons ensuite le mock de notre service avec un `spyOn`

```
spy = spyOn(backService, 'loadPersonnesNames').and.returnValue(asyncData([  
    'aymen',  
    'mohamed'  
]));
```

- Finalement avec un `fakeAsync` on peut tester notre réponse
- La méthode `queryAll` du `debugElement` retourne un tableau de `debugElement` répondant à la requête.

# Tester du HTTP

---

```
it('should show needed data', fakeAsync(() => {
  spy = spyOn(backService,
'loadPersonnesNames').and.returnValue(asyncData(['aymen', 'mohamed']));
  fixture.detectChanges();
  backService.loadPersonnesNames().subscribe(
    (noms) => {
      component.names = noms;
    }
  );
  tick();
  fixture.detectChanges();
  const names = fixture.debugElement.queryAll(By.css('.personne'));
  expect(names.length).toBe(2);
  expect(names[0].nativeElement.textContent.trim()).toContain('aymen');
}));
});
```



# Tester les événements.

---

➤ L'objet `debugElement` nous offre une méthode `triggerEventHandler()` qui permet de déclencher des événements.

➤ `triggerEventHandler` prend en paramètre le nom de l'événement et le second paramètre est un objet de type event. Cet objet n'est pas obligatoire pour la plupart des cas.

<https://angular.io/guide/testing#triggereventhandler>

➤ l'idée est donc de sectionner l'événement qui va écouter l'événement et d'y déclencher l'événement souhaité et enfin vérifier le bon déroulement des choses.

# Exercice

---

- Créer un composant ColorComponent.
- La couleur du background du paragraphe crée par défaut doit être rouge. En cliquant sur le paragraphe, elle doit se changer en jaune.
- Tester le fonctionnement de votre composant.

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { ColorComponent } from '../color.component';
import { By } from '@angular/platform-browser';

describe('ColorComponent', () => {
  let component: ColorComponent;
  let fixture: ComponentFixture<ColorComponent>;
  let compiled: ComponentFixture<ColorComponent>;
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ ColorComponent ]
    })
  }));
  beforeEach(() => {
    fixture = TestBed.createComponent(ColorComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
    compiled = fixture.debugElement.nativeElement;
  });
  it('should create', () => {
    expect(component).toBeTruthy();
  });
  it('p should have red color', () => {
    fixture.detectChanges();
    const h = fixture.debugElement.query(By.css('p'));
    expect(h.nativeElement.style.backgroundColor.trim()).toBe('red');
    h.triggerEventHandler('click', null);
    fixture.detectChanges();
    expect(h.nativeElement.style.backgroundColor.trim()).toBe('yellow');
  });
});
```

---

aymen.sellaouti@gmail.com