

# Formation : Angular, maîtriser le Framework Front-End de Google Introduction

---

AYMEN SELLAOUTI

# PRÉ-REQUIS

---

- ▶ HTML 5 / CSS 3 / Bootstrap



- ▶ JavaScript

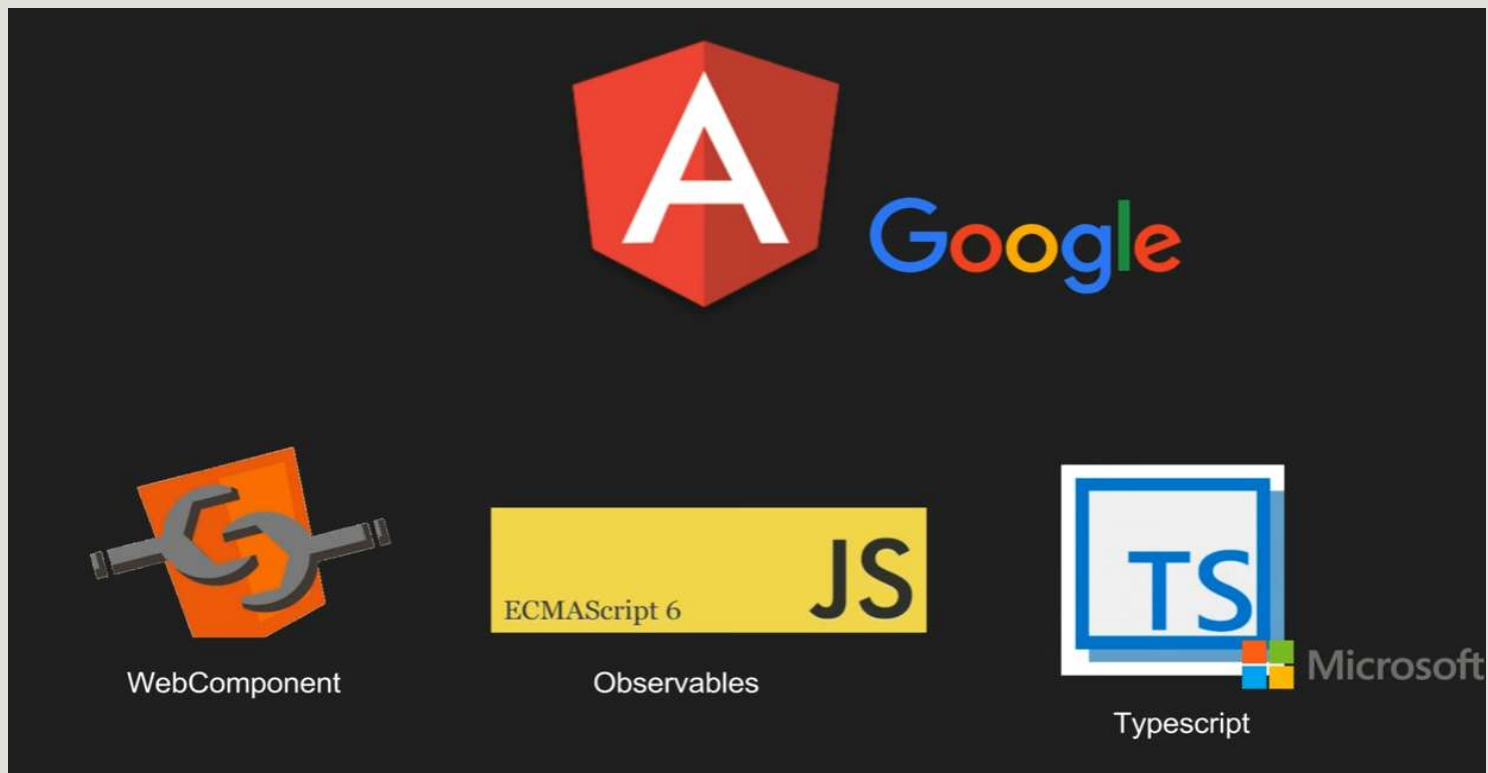


- ▶ Programmation Orientée Objet

**POO**

# C'est quoi Angular?

---

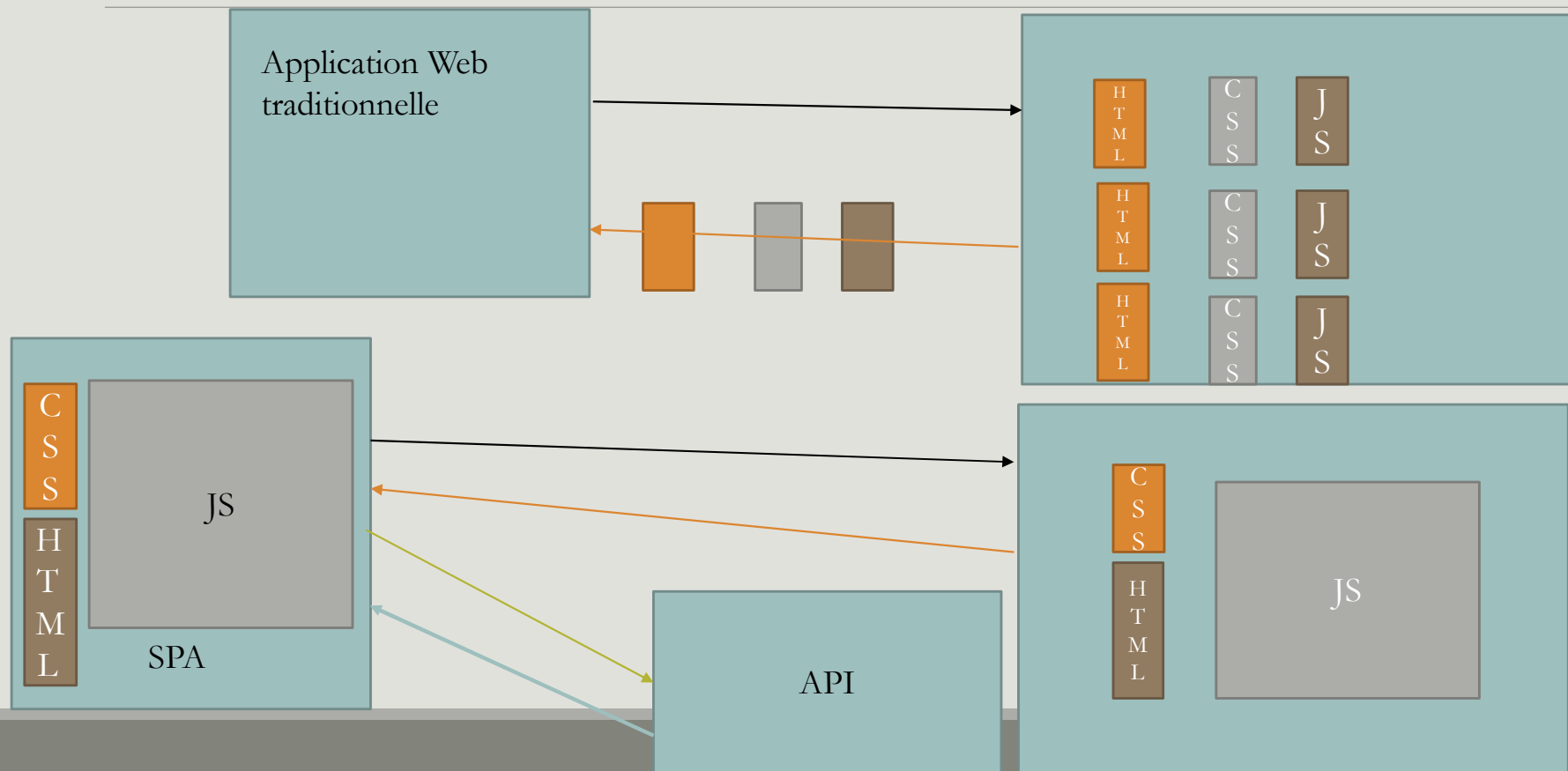


# C'est quoi Angular?

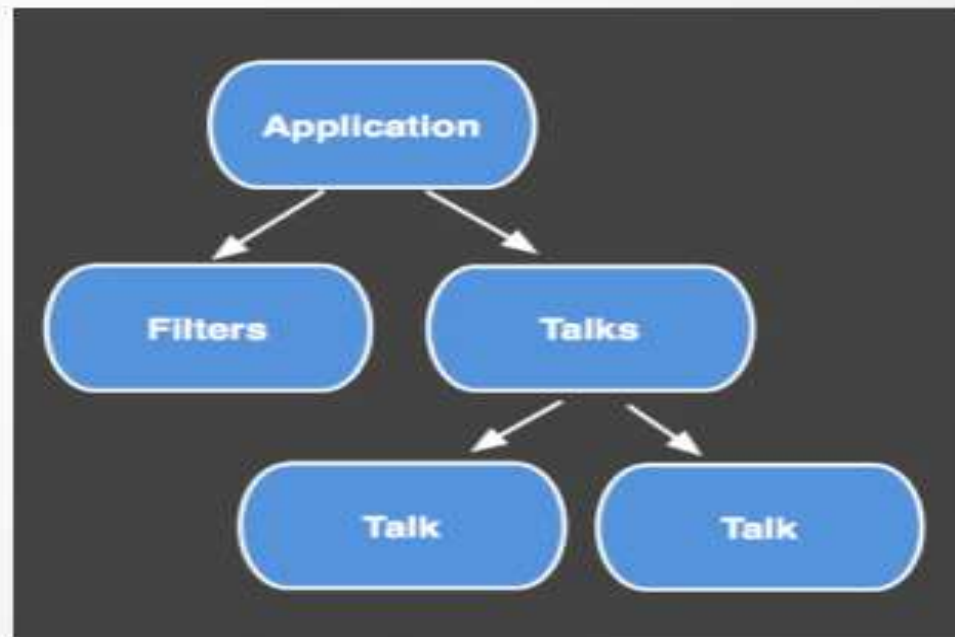
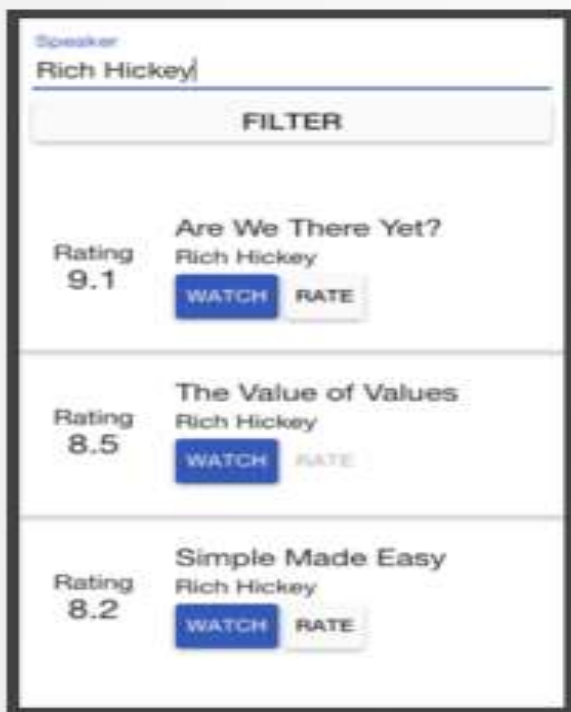
---

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

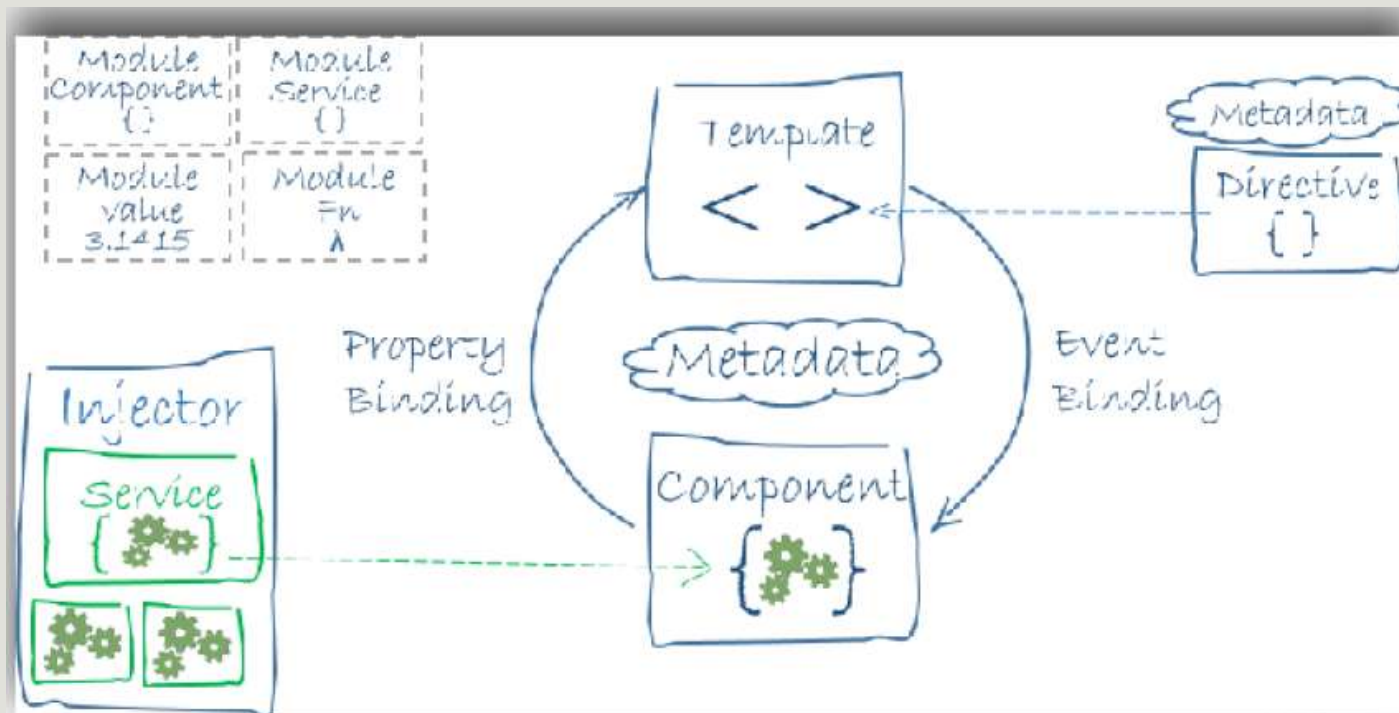
# SPA



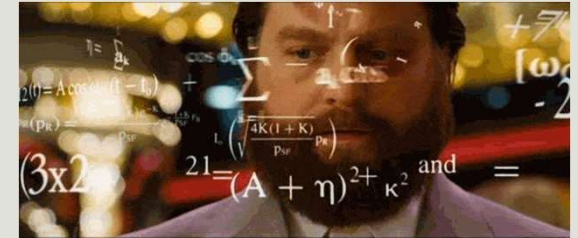
# Angular : Arbre de composants



# Architecture Angular



# Installation d'Angular Angular Cli



- Nous allons installer notre première application en utilisant **angular Cli**.
- Si vous avez Node c'est bon, sinon, installer **NodeJs** sur votre machine. Vous devez avoir une version de **node nécessaire pour la version Angular que vous installez**.
- Une fois installé vous disposez de npm qui est le **Node Package Manager**. Afin de vérifier si vous avez NodeJs installé, tapez **npm -v**.
- Installer maintenant le Cli en tapant la : **npm install -g @angular/cli**
  - **npm install -g @angular/cli@16.0.0** installe la version 16.0.0
  - **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**
- **npx @angular/cli@16.2.15 new nomProjet**
- Afin d'avoir du help pour le cli tapez **ng help**
- Lancer le projet en utilisant la commande **ng serve**

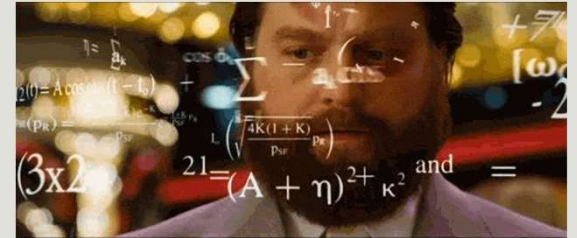


# Angular dépendances

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

<https://gist.github.com/LayZeeDK/c822cc812f75bb07b7c55d07ba2719b3>

# Installation d'Angular Angular Cli



- Vous pouvez configurer le Host ainsi que le port avec la commande suivante : `ng serve --host leHost --port lePort`
- Pour plus de détails sur le cli visitez <https://cli.angular.io/>

# Quelques commandes du Cli

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

# Ajouter Bootstrap

---

On peut ajouter Bootstrap de plusieurs façons :

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

# Ajouter Bootstrap

---

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

1- En l'ajoutant dans index.html

2- En ajoutant le [chemin](#) des dépendances dans les tableaux [styles](#) et [scripts](#) dans le fichier [angular.json](#):

```
"styles": [  
  "../node_modules/bootstrap/dist/css/bootstrap.min.css",  
  "styles.css",  
],  
"scripts": [  
  "../node_modules/jquery/dist/jquery.min.js",  
  "../node_modules/popper.js/dist/umd/popper.min.js",  
  "../node_modules/bootstrap/dist/js/bootstrap.min.js"  
],
```

# Ajouter Bootstrap

---

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

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

Essayer la même chose avec font-awesome.

# Angular

# Les composants

---

AYMEN SELLAOUTI

# Objectifs

---

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



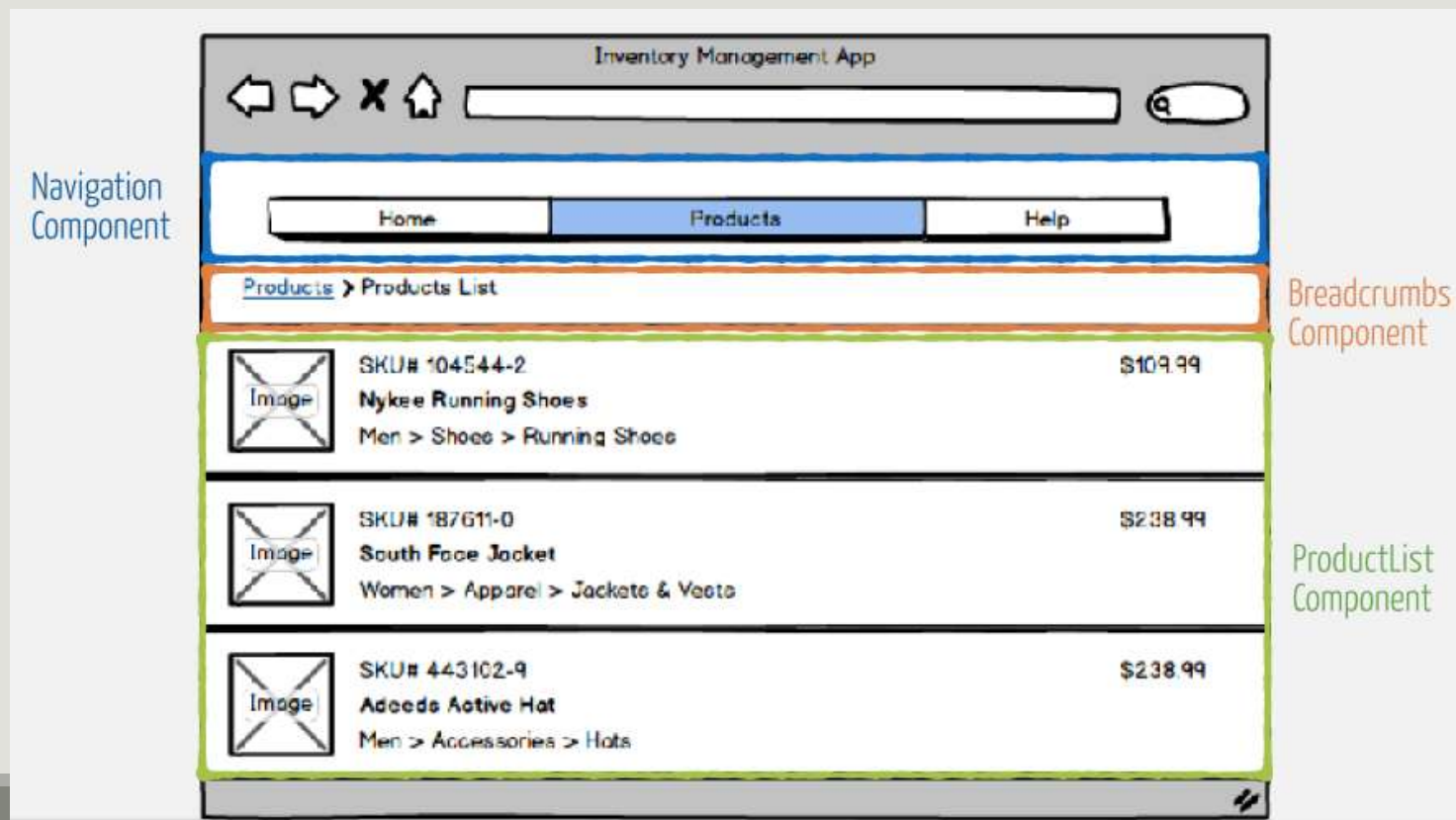
# Qu'est ce qu'un composant (Component)

---

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

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

# Quelques exemples



# Quelques exemples

Product Row  
Component

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

# Quelques exemples

---

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

# Premier Composant

---

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

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

Chargement de la classe Component

Le décorateur @Component permet d'ajouter un comportement à notre classe et de spécifier que c'est un Composant Angular.

**selector** permet de spécifier le tag (nom de la balise) associé ce composant

**templateUrl**: spécifie l'url du template associé au composant

**styleUrls**: tableau des feuilles de styles associé à ce composant

Export de la classe afin de pouvoir l'utiliser

# Création d'un composant

---

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

# Création d'un composant

---

- La commande `generate` possède plusieurs options

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

# Que contient le composant ?

---

- Le composant dispose de deux parties :
  - La partie HTML qui représente la Vue
  - La partie TS décrivant l'état et le comportement du composant



# L'état et le comportement d'un composant

```
export class FirstComponent {  
  //Propriétés : état State  
  name = 'First Component';  
  isHidden = false;  
  message = '';  
  //Méthodes : comportement Behaviour  
  showHide() {  
    this.isHidden = !this.isHidden;  
  }  
  changeMessage(newMessage: string) {  
    this.message = newMessage;  
  }  
}
```

Classe

```
<!-- attributs: l'état de la balise -->  
<p hidden> First Component works!</p>  
<div class="alert alert-info">  
  Je test le binding  
</div>  
<p>Le contenu de l'input est : </p>  
<input  
  type="text"  
  class="form-control"  
>  

```

Template

First Component works!

Je test le binding

Le contenu de l'input est :



- Et si on voulait afficher quelque chose dans notre template ?
- Et si on voulait contrôler un attribut de notre template ?
- Et si on voulait réagir à un événement qui survient dans notre template ?

# Afficher des propriétés dans le Template Interpolation

- L'interpolation permet de projeter des valeurs de propriétés dans votre template.
- Angular utilise la syntaxe "double accolades `{{ }}`" pour l'interpolation

```
export class FirstComponent {  
  //Propriétés : état State  
  
  name = 'First Component';  
  isHidden = false;  
  message = '';  
  //Méthodes : comportement Behaviour  
  showHide() {  
    this.isHidden = !this.isHidden;  
  }  
  changeMessage(newMessage: string) {  
    this.message = newMessage;  
  }  
}
```

```
<!-- attributs: l'état de la balise -->  
<p hidden>First Component works!</p>  
<p hidden {{name}} works!</p>  
<div class="alert alert-info">  
  Je test le binding  
</div>  
<p>Le contenu de l'input est : </p>  
<input  
  type="text"  
  class="form-control"  
>  

```

First Component works!

Je test le binding

Le contenu de l'input est :



# Contrôler un attribut de notre template

## Property Binding

---

- Afin de **contrôler un attribut d'une des balises de notre Template**, angular nous fournit le concept de **Binding de propriété (Property Binding)**.
- C'est un Binding unidirectionnel.
- La propriété liée au composant est interprétée avant d'être ajoutée au Template.
- syntaxe: **[propriété]="varOuCte"**

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

# Contrôler un attribut de notre template Property Binding

```
export class FirstComponent {  
  //Propriétés : état State  
  name = 'First Component';  
  isHidden = false;  
  message = '';  
  imagePath = 'assets/images/as.jpg';  
  //Méthodes : comportement Behaviour  
  showHide() {  
    this.isHidden = !this.isHidden;  
  }  
  changeMessage(newMessage: string) {  
    this.message = newMessage;  
  }  
}
```

```
<!-- attributs: l'état de la balise -->  
<p hidden> {{name}} works!</p>  
<p [hidden]="isHidden"> {{name}} works!</p>  
  
<div class="alert alert-info">  
  Je test le binding  
</div>  
<p>Le contenu de l'input est : {{ message}}</p>  
<input  
  type="text"  
  class="form-control"  
>  
  
<img [src]="imagePath" alt="aymen">
```

First Component works!

Je test le binding

Le contenu de l'input est :



# Réagir à un événement qui survient dans notre template

## Event Binding

---

- Afin **d'écouter et de réagir** à **un événement déclenché** dans notre **Template**, angular nous fournit le concept de **Binding d'événement** (**Event Binding**).
- C'est un **binding unidirectionnel**, Il permet d'interagir du DOM vers le composant. L'interaction se fait à travers les **événements**.
- Syntaxe : **(evenement)**="methodeAExecuter()">

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

# Réagir à un événement qui survient dans notre template

## Event Binding

---

```
export class FirstComponent {  
  //Propriétés : état State  
  name = 'First Component';  
  isHidden = false;  
  message = '';  
  //Méthodes : comportement Behaviour  
  showHide() {  
    this.isHidden = !this.isHidden;  
  }  
  changeMessage(newMessage: string) {  
    this.message = newMessage;  
  }  
}
```

```
<!-- attributs: l'état de la balise -->  
<p [hidden]="isHidden"> {{name}} works!</p>  
<!-- Au click appelle la fonction showHide -->  
<div (click)="showHide()" class="alert alert-info">  
  Je test le binding  
</div>  
<p>Le contenu de l'input est : {{ message}}</p>  
<input  
  type="text"  
  class="form-control"  
>
```

# TEMPLATE REFERENCE

- Dans certains cas d'utilisation, vous avez besoin de récupérer **la référence d'un objet du DOM dans votre template**. Par exemple la référence d'un input pour accéder à sa valeur.
- Pour ce faire, on utilise le symbole # pour la création d'une variable de référence.

```
export class FirstComponent {  
  //Propriétés : état State  
  name = 'First Component';  
  isHidden = false;  
  message = '';  
  //Méthodes : comportement Behaviour  
  showHide() {  
    this.isHidden = !this.isHidden;  
  }  
  changeMessage(newMessage: string) {  
    this.message = newMessage;  
  }  
}
```

Classe

```
<!-- attributs: l'état de la balise -->  
<p hidden> First Component works!</p>  
<div class="alert alert-info">  
  Je test le binding  
</div>  
<p>Le contenu de l'input est : </p>  
<input  
  #myInput  
  (change)="changeMessage(myInput.value)"  
  type="text"  
  class="form-control"  
>  

```

Template

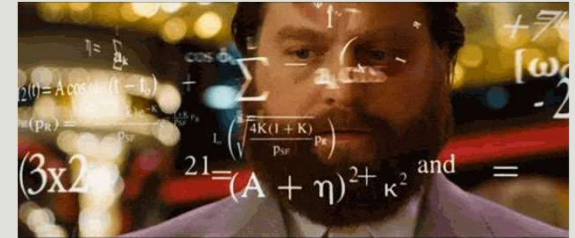
First Component works!

Je test le binding

Le contenu de l'input est :



# 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, ça 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]`





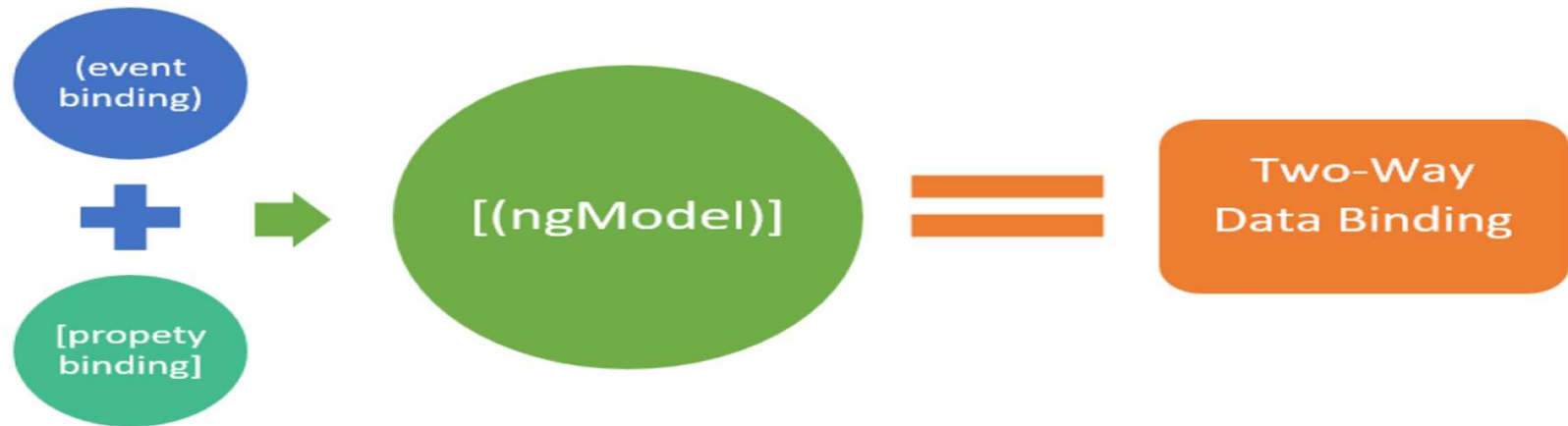
# Two way Binding

---

- 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

# Two way Binding

## PROPERTY BINDING + EVENT BINDING



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

Template

# Property Binding et Event Binding

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

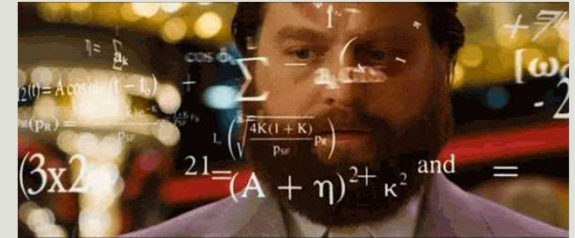
@Component({
  selector: 'app-two-way',
  templateUrl: './two-way.component.html',
  styleUrls: ['./two-way.component.css']
})
export class TwoWayComponent {
  two: any = "myInitValue";
}
```

Component

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

Template


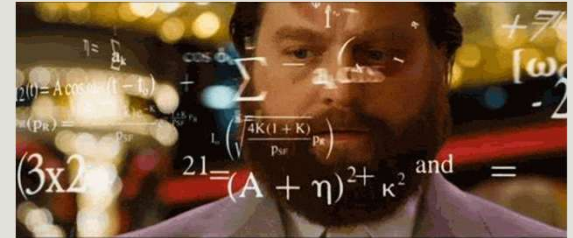
# Exercice



- Le but de cet exercice est de créer un aperçu de carte visite.
- Créer un composant
- Préparer une interface permettant de saisir d'un côté les données à insérer dans une carte visite. De l'autre côté 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>

# Exercise



Aymen Sellaouti  
trainer

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

Auto Rotation

name :

sellaouti

firstname :

aymen

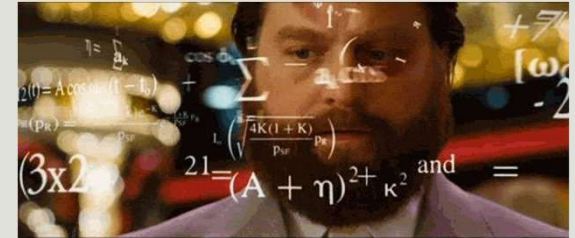
job :

trainer

path :

rotating\_card\_profile3.png

# Exercice



## Two Way Binding



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

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

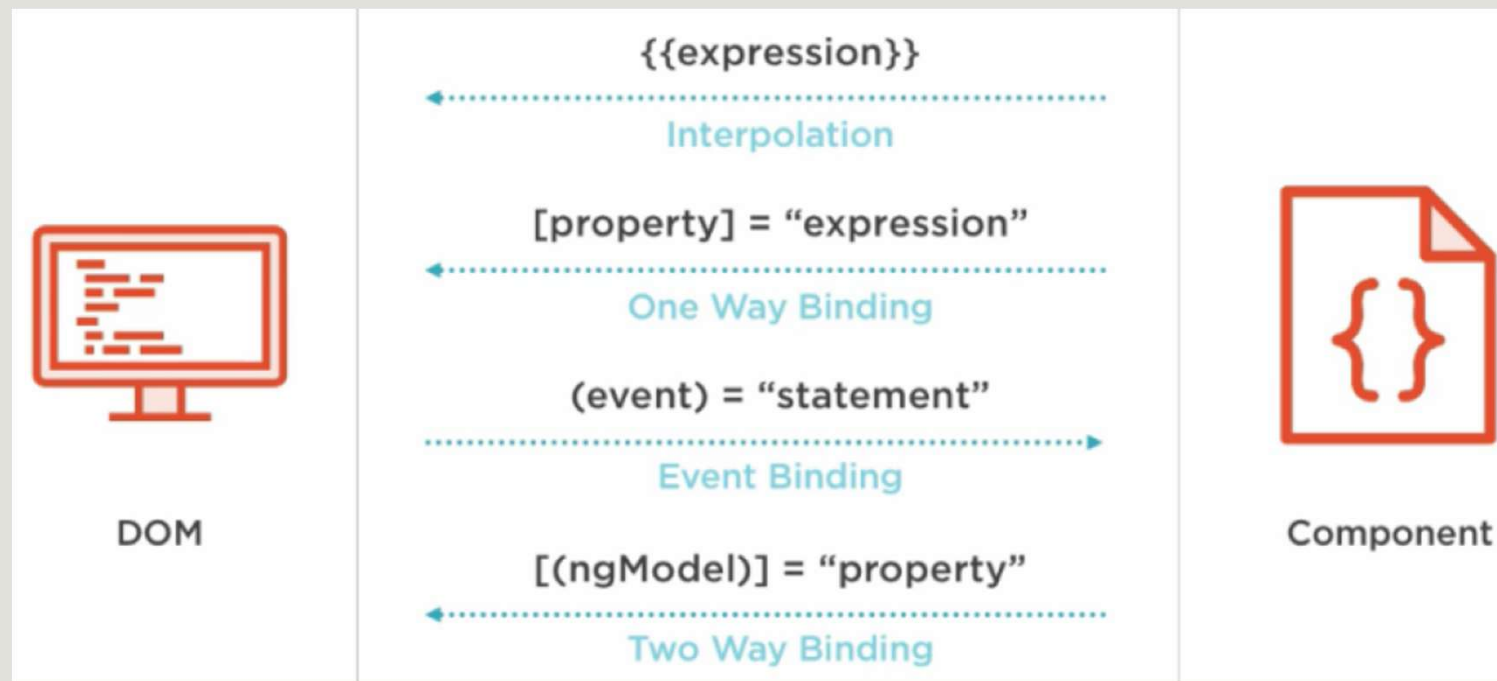
image :  
as.jpg

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

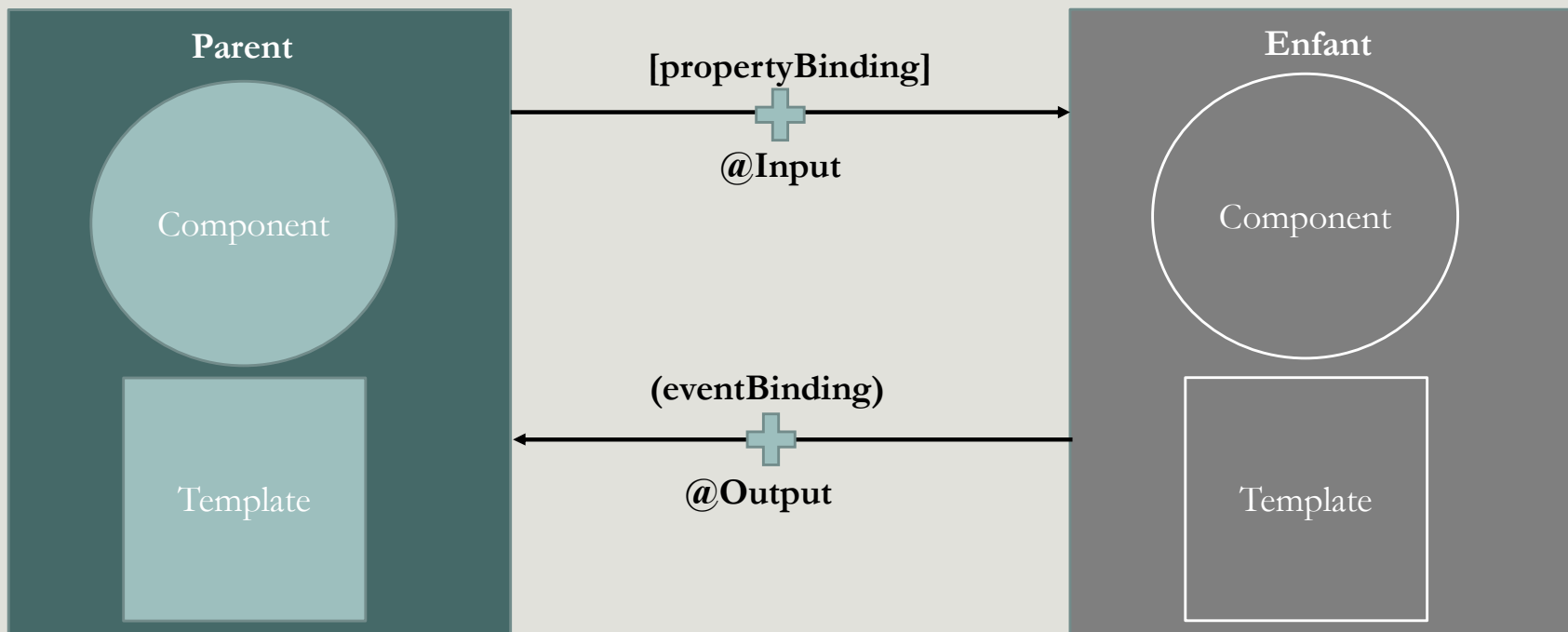
Mots clé de votre travail :  
HTML CSS JS PHP Symfony Angular

# Résumé : Property Binding



# Interaction entre composants

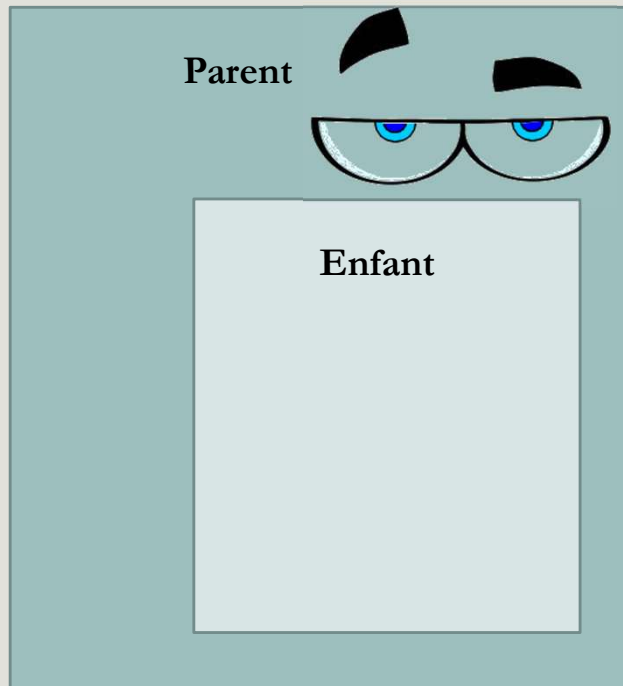
---





# Pourquoi ?

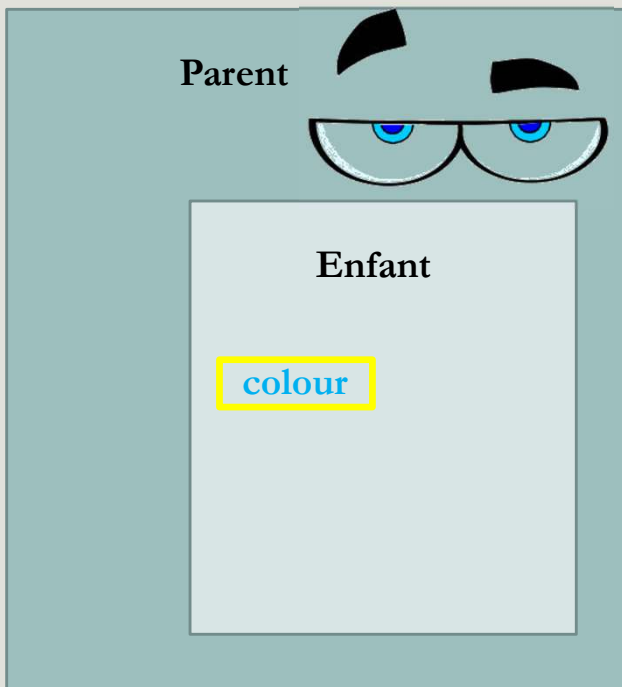
Le père voit le fils, le fils ne voit pas le père



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

@Component({
  selector: 'app-pere',
  template: `
    <p>Je suis le composant père </p>
    <forma-fils></forma-fils>
  `,
  styles:
})
export class PereComponent {
  color = 'green';
}
```

# Interaction du père vers le fils



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

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

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

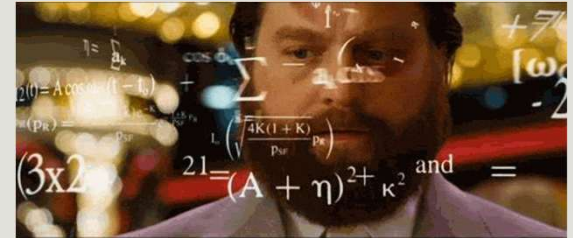
```
<app-fils [colour]="color"/>
```

Template du pere

Ts du fils

# Exercice

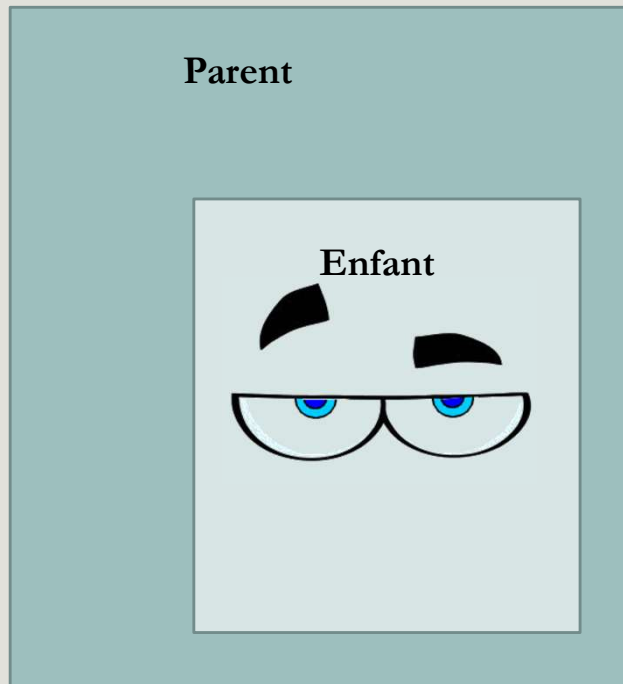
---



- 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

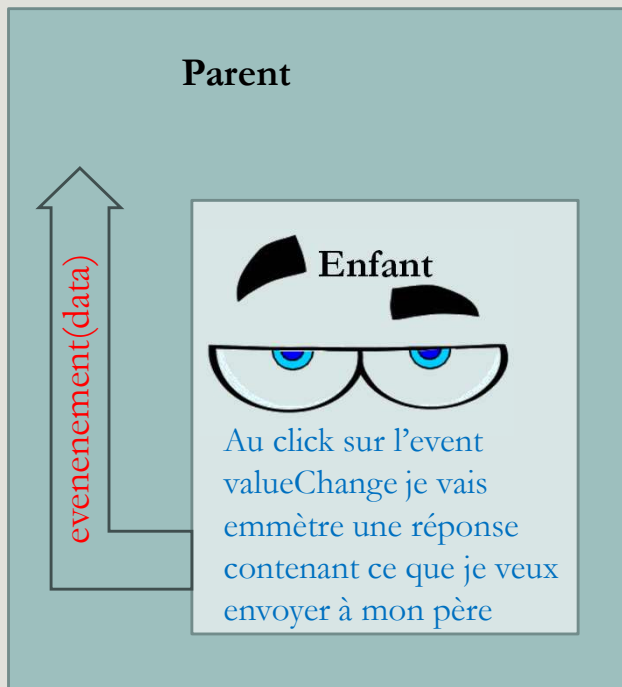
# Interaction du fils vers le père

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



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

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

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

Template du fils

<app-fils [colour]="color"/>

Template du père

# Interaction du fils vers le père

---

## 2. Configurer l'événement

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

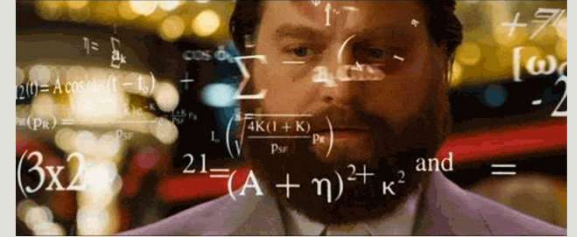
## 3. Récupérer l'évènement dans le composant parent

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

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

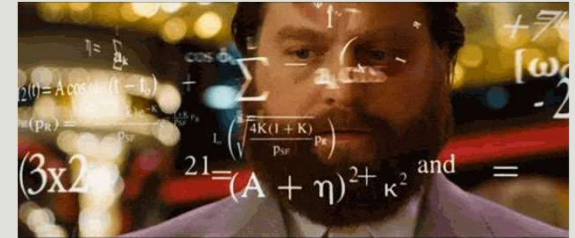
```
ReceivedEvent(msg : any)  
{  
  alert(msg);  
}
```

# Exercice

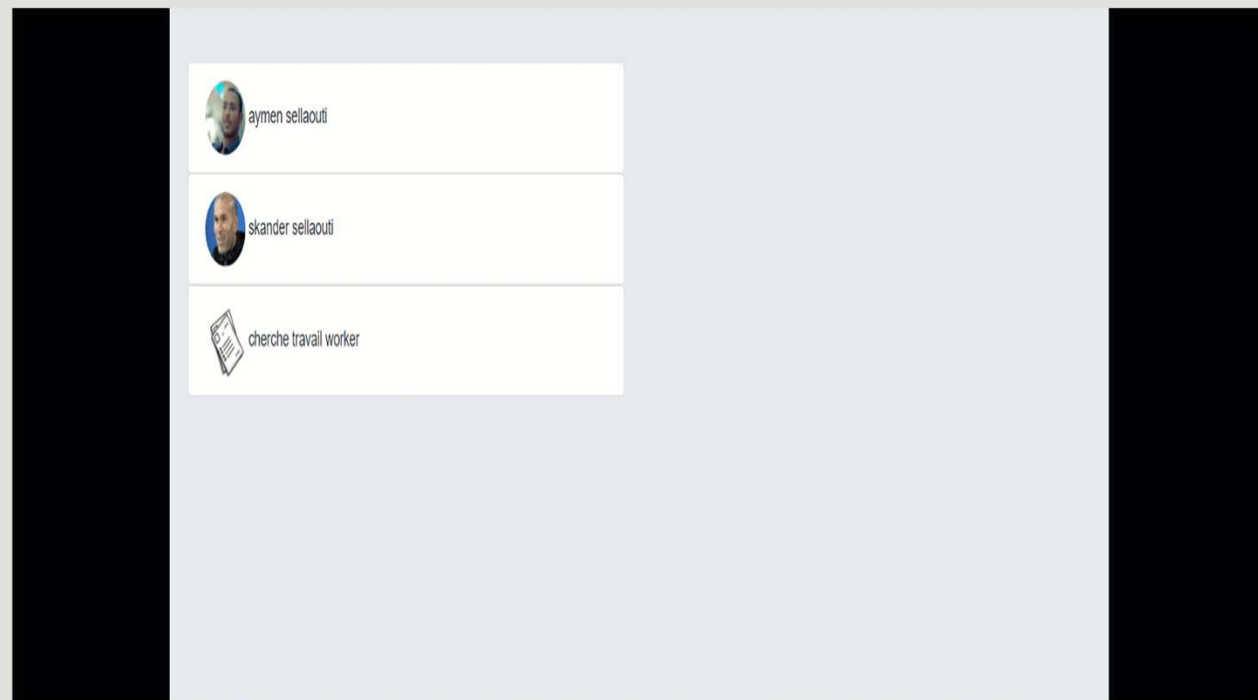


- Ajouter une variable `myFavoriteColor` dans le composant du fils.
- Ajouter un bouton dans le composant Fils
- Au click sur ce bouton, la couleur de background du Div du père doit prendre la couleur favorite du fils.

# Exercice

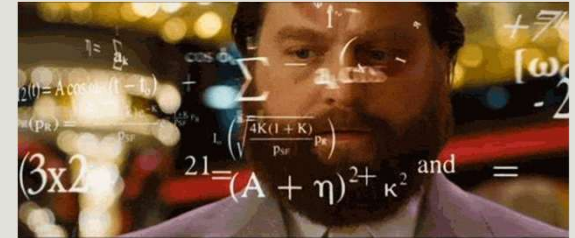


- Le but de cet exercice est de créer une mini plateforme de recrutement.
- La première étape est de créer la structure suivante avec une vue contenant deux parties :
  - Liste des Cvs inscrits
  - Détail du Cv qui apparaîtra au click
- Il vous est demandé juste d'afficher un seul Cv et de lui afficher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





# Exercice



- Voici la décomposition de l'interface en composant

Un cv est caractérisé par :

id

name

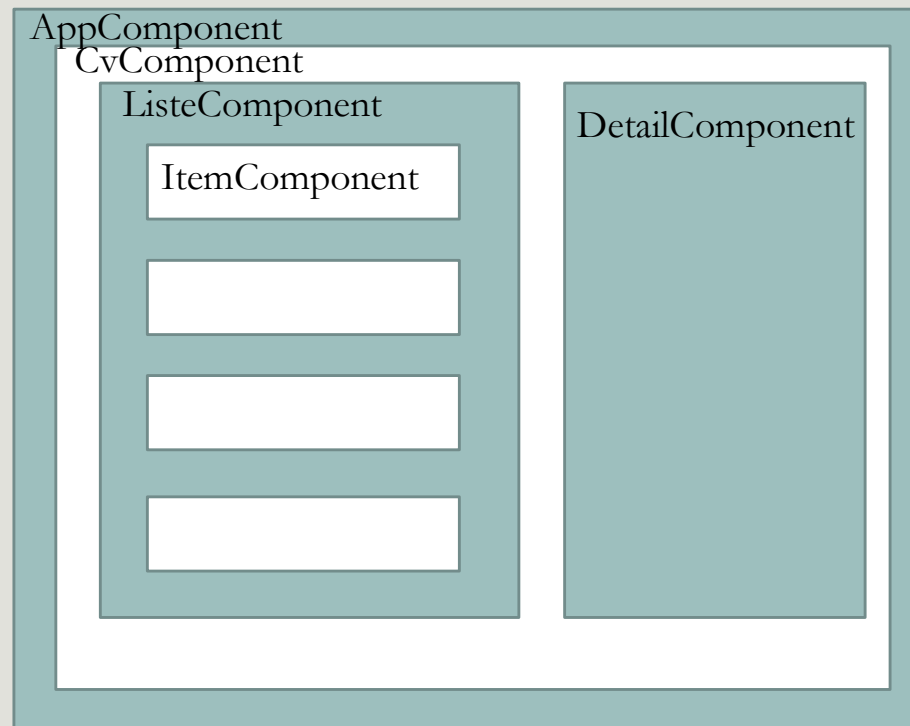
firstname

Age

Cin

Job

path



# Angular

# Les directives

---

AYMEN SELLAOUTI

# Objectifs

---

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

# Qu'est ce qu'une directive

---

- Une directive est une **classe annotée avec le décorateur @Directive()** qui permet à Angular **d'attacher un comportement spécifique à un élément HTML**.
- Les directives permettent :
  - **Réutilisabilité du code : en centralisant** un comportement ou un style **réutilisable** dans toute l'application.
  - **Séparation des préoccupations (Separation of concerns) :**  
La logique **métier est séparée du design** ou de l'affichage.
  - **Clarté et lisibilité :**  
Plutôt qu'un code lourd dans le composant, la directive gère une fonctionnalité spécifique (exemple : gestion de permissions d'un bouton).
  - **Personnalisation facile :**  
Tu peux créer des directives sur mesure pour enrichir le comportement de tes composants

# 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**.
- Il existe des **directives fournies par Angular** mais vous pouvez **créer vos propres directives**

# Les directives structurelles

---

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

# Les directives structurelles \*ngIf

---

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

Exemple

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

# Les directives structurelles \*ngFor

➤ Permet de répéter un élément plusieurs fois dans le DOM.

➤ Prend en paramètre les entités à reproduire.

➤ Fournit certaines valeurs :

➤ index : position de l'élément courant

➤ first : vrai si premier élément

➤ last vrai si dernier élément

➤ even : vrai si l'indice est paire

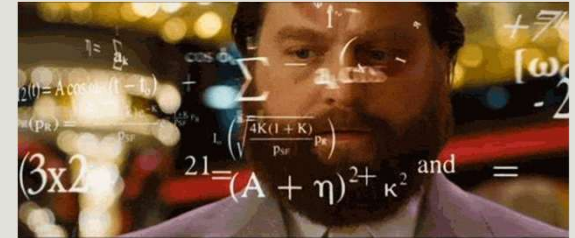
➤ odd : vrai si l'indice est impaire

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

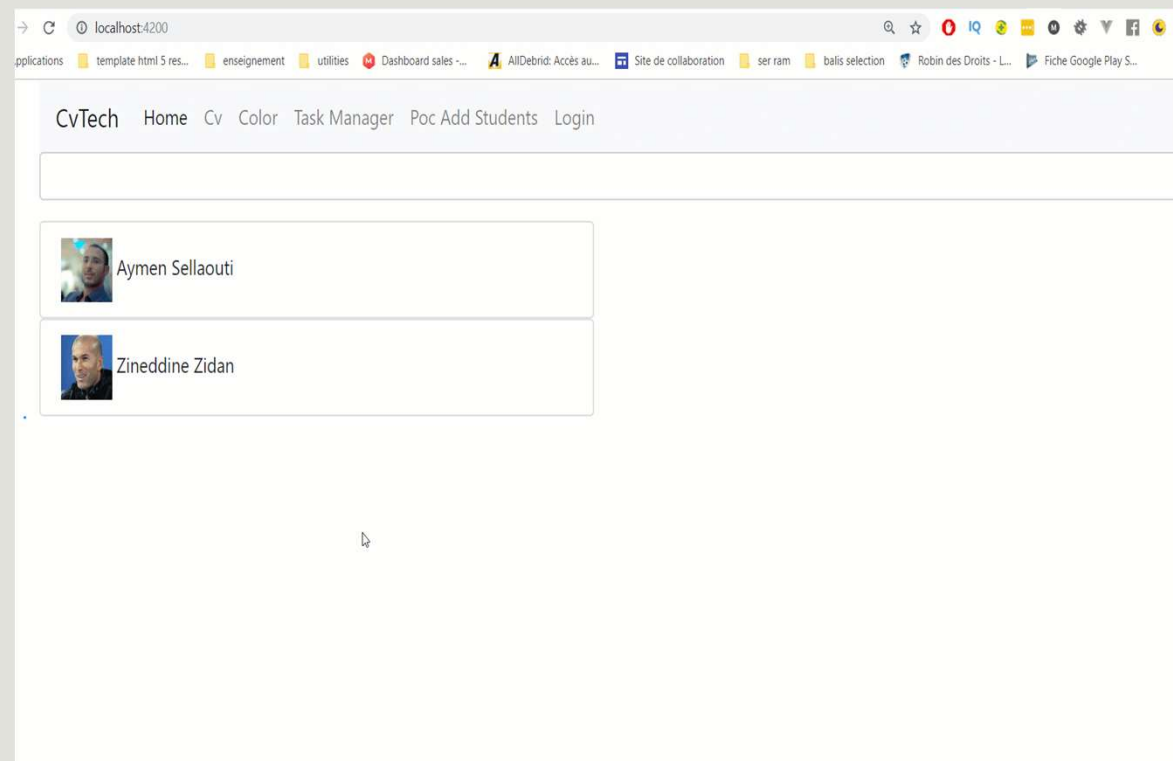
```
<ul>
  <li *ngFor="let episode of episodes; let i = index;">
    Episode {{i+1}}{{episode.title}}
  </li>
</ul>
```



# Exercice (Notre Projet)



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



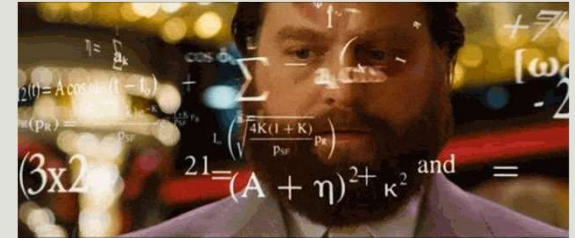
# Les directives d'attribut (ngStyle)

- Cette directive permet de modifier l'apparence de l'élément cible.
- Elle est placée entre [ ] **[ngStyle]** qu'on ajoute dans la balise cible.
- Elle prend en paramètre un **attribut** représentant un **objet** décrivant le **style** à appliquer.
- Dans cet objet, la **clé** va représenter l'attribut de style que vous voulez gérer, e.g. color, backgroundColor, fontSize, fontFamily, border.
- La **valeur** représente la valeur associée à cette propriété de style et qui peut être une constante, dans ce cas elle ne change pas ou une **variable** et donc elle suivra toujours la valeur de cette variable.
- Dans cet exemple, l'attribut de style **color** prend sa valeur de la variable color, il sera donc 'lightblue'.
- Pour l'attribut de style fontFamily on lui a associé une constante qui est 'garamond'.

```
@Component({
  selector: 'app-ngstyle-exemple',
  templateUrl: './ngstyle-exemple.component.html',
  styleUrls: ['./ngstyle-exemple.component.css']
})
export class NgstyleExempleComponent {
  color = 'lightblue';
}
```

```
<p
  [ngStyle]="{
    backgroundColor: 'red',
    color: color,
    fontFamily: 'garamond'
  }"
>ngstyle-exemple works!</p>
```

# Exercice



- Nous voulons simuler un Mini Word pour gérer un paragraphe en utilisant ngStyle.
- Préparer un input de type color, un input de type number, et un select box.
- Faire en sorte que lorsqu'on change la couleur du color 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.

Test

30

arial

# Les directives d'attribut (ngClass)

- Cette directive permet de modifier l'attribut **class**.
- Elle cohabite avec l'attribut **class**.
- Elle prend en paramètre
  - Une chaîne (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**.
- Dans cet exemple la classe CSS off sera appliqué à la Div

```
@Component({
  selector: 'app-ngclass-exemple',
  templateUrl: './ngclass-exemple.component.html',
  styleUrls: ['./ngclass-exemple.component.css'],
})
export class NgclassExempleComponent {
  isOn = false;
}
```

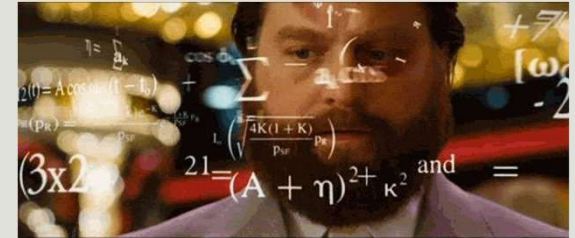
```
<div
  [ngClass]="{
    on: isOn,
    off: !isOn
  }"
  class="ampoule">lampe</div>
```

# Les directives structurelles \*ngFor

- Pour rappel \*ngFor fournit certaines valeurs que nous pouvons combiner avec ngClass entre autres:
  - 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; let i = index;
    let isOdd = odd; let isFirst=first"
    [ngClass]="{ odd: isOdd , bgfonce: isFirst}"
  >
    Episode {{i+1}}{{episode.title}}
  </li>
</ul>
```

# Exercice



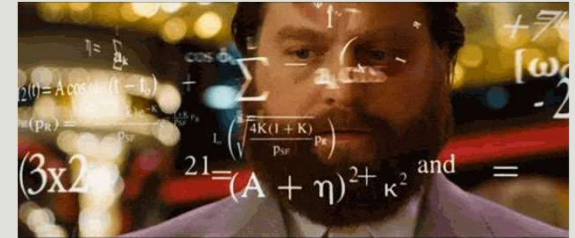
- Afin d'améliorer l'affichage de la liste des cvs dans votre CvTech, reprenez le composant cvList et faites-en sortes d'avoir les éléments successifs colorés d'une manière permutée entre deux couleurs.



# Customiser une directive d'attribut

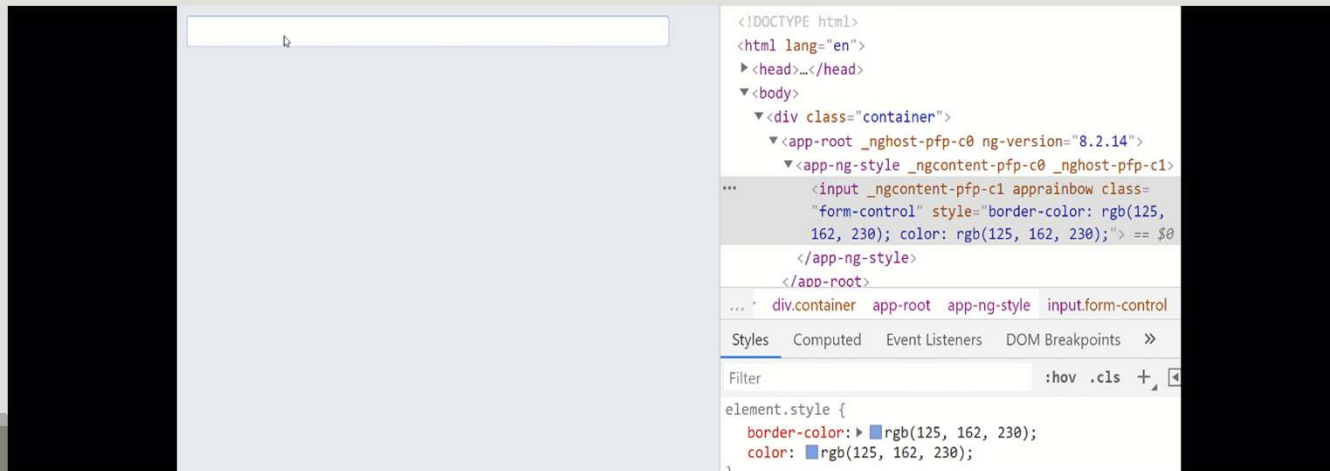
- Afin de créer sa propre « attribut directive » il faut utiliser un `HostBinding` sur la **propriété** que vous voulez **binder**.
  - **Exemple :** `@HostBinding('style.backgroundColor')`  
`bg:string="red";`
- Dans cet exemple on lie (bind) l'attribut **bg** de la directive à l'attribut **style.backgroundColor** de l'élément hôte (celui qui est tagué) de la directive. A chaque fois qu'on change **bg** le **style.backgroundColor** de l'élément hôte change.
- 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 `HostListner` il faut les importer du `core d'angular`

# Exercice



Un truc plus sympa, 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.
- Préparer une méthode qui permet de générer aléatoirement une couleur dans votre directive.
- Faite en sorte qu'en appliquant votre directive à un input, à chaque écriture d'une lettre (event keyup) la couleur change en prenant aléatoirement l'une des couleurs de votre tableau. Pensez à utiliser `Math.random()` qui vous retourne une valeur entre 0 et 1.





# 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">`

# Angular

# Les pipes

---

AYMEN SELLAOUTI

# Plan du Cours

---

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

# Objectifs

---

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

# Qu'est ce qu'un pipe

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

Avec le pipe uppercase :

Sans aucun pipe :

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

# Syntaxe

---

- Afin d'utiliser un pipe vous utilisez la syntaxe suivante :
  - `{{ variable | nomDuPipe }}`
- Exemple : `{{ maDate | date }}`
- Afin d'utiliser plusieurs pipes combinés vous utilisez la syntaxe suivante :
  - `{{ variable | nomDuPipe1 | nomDuPipe2 | nomDuPipe3 }}`
- Exemple : `{{ maDate | date | uppercase }}`

# 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 }}`

# 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
- ...



# 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';
    }
  }
}
```

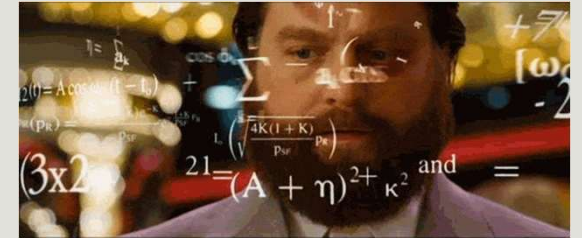
```
<li>
  <ol *ngFor="let team of
teams">
    {{team | team}}
  </ol>
</li>
```

```
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 ou la valeur fournie au pipe est une chaine vide ou ne contient que des espaces.



# Angular Service et injection de dépendances



---

AYMEN SELLAOUTI

# Objectifs

---

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

# Qu'est ce qu'un service ?



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

Component 1

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

Component 2

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

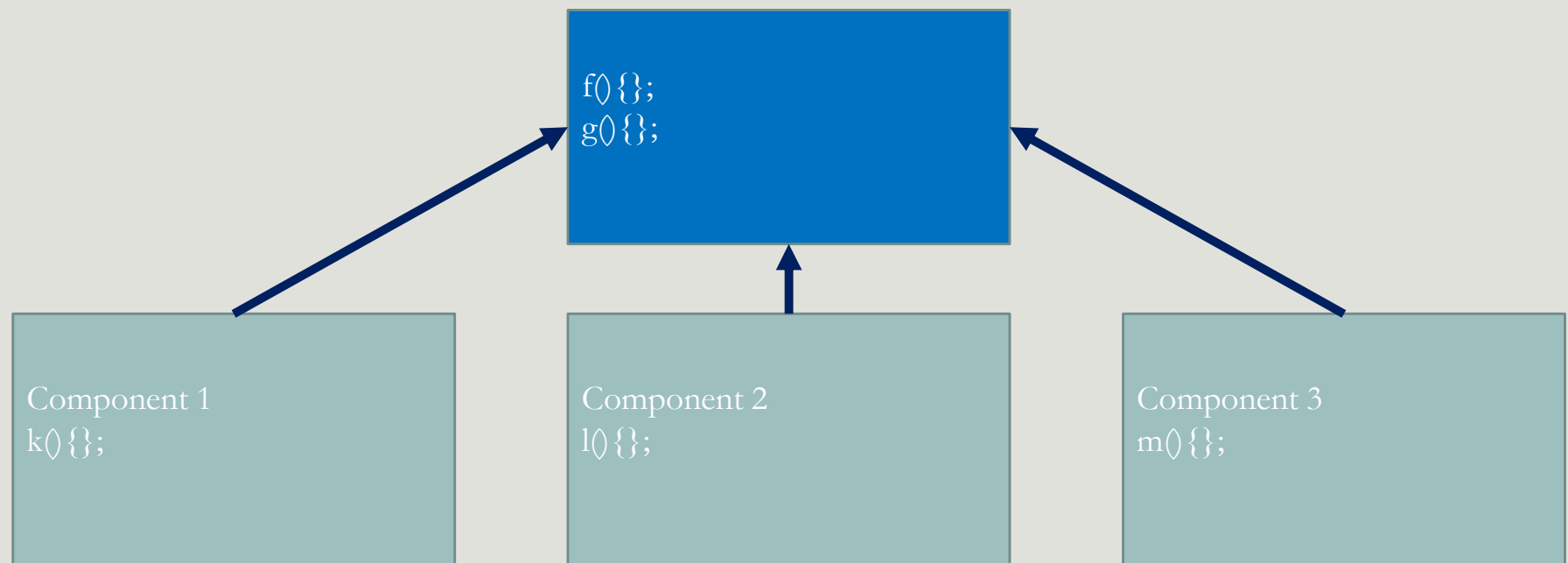
Component 3

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

**Redondance de code**

**Maintenabilité difficile**

# Qu'est ce qu'un service ?



# Qu'est ce qu'un service ?

---



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



# Création d'un service

---

- Via CLI

- `ng generate service nomDuService`

- `ng g s nomDuService`

# Premier Service

---

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

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

  constructor() { }

}
```

# Comment fonctionne un restaurant ?



# 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.

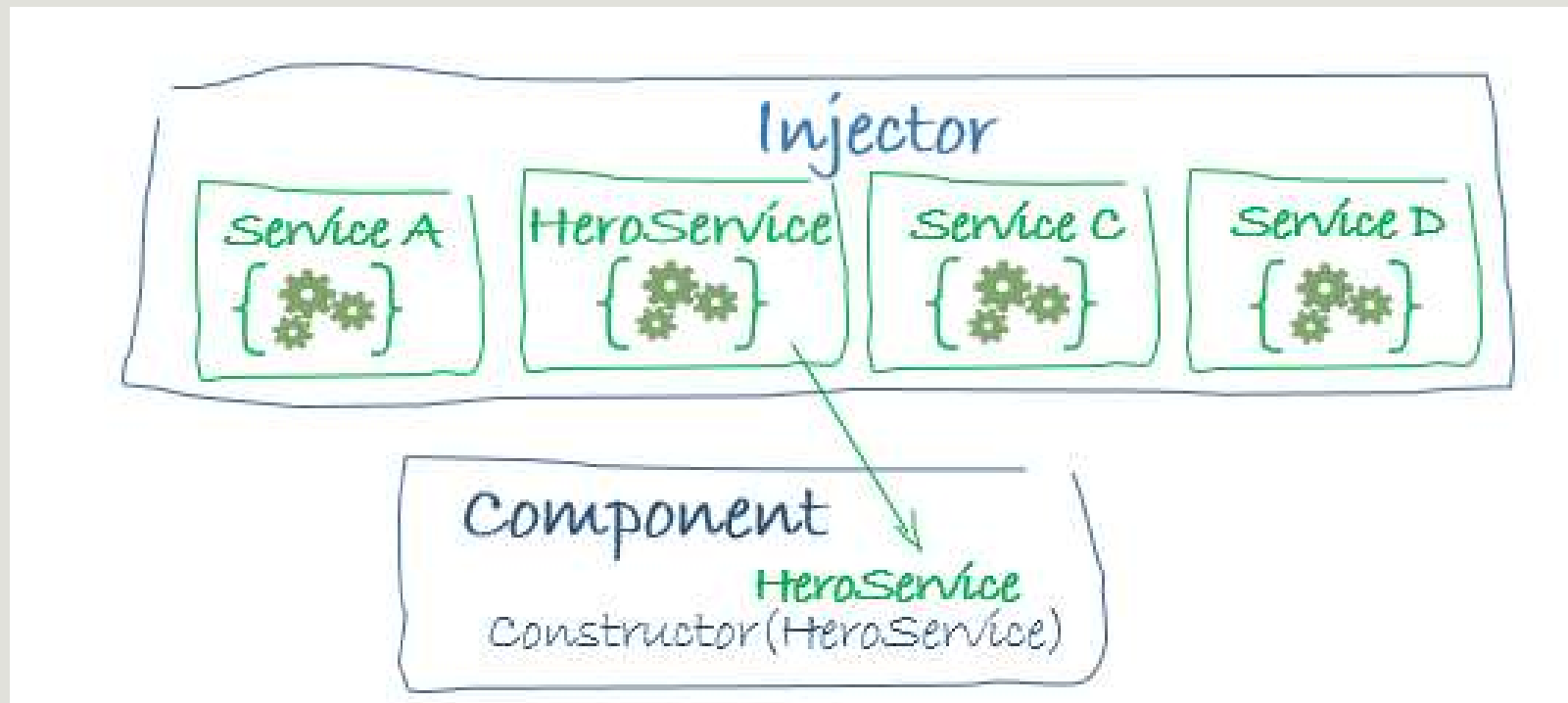
```
Classe A1 {  
  Constructor(B b, C c)  
  ...  
}
```

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

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

INJECTOR

# Injection de dépendance (DI)



# Comment fonctionne un restaurant (système d'injection de dépendances) ?

Providers

Injector

IOC

Injection



# Injection de dépendance (DI)

---

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



# Injection de dépendance (DI)

---

- L'injection de dépendance utilise les étapes suivantes :
  - **Provider les dépendances** (**Préparer notre menu, définir quels plats nous pouvons fournir**) via **l'annotation @Injectable et son attribut providedIn** (ici les services), dans le **provider du module ou du composant** (**C'est la préparation de votre menu**).
  - **Injecter le service ( commander le plat souhaité )** en le passant comme **paramètre du constructeur** de la **classe (composant ou service) qui en a besoin**.

# Injection de dépendance (DI)

## Provider les dépendances

```
@NgModule({  
  providers: [CvService],  
  bootstrap: [AppComponent],  
})  
export class AppModule {}
```

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

Provider

Injector

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

```
export class CvComponent {  
  constructor(  
    private cvService: CvService  
  ) {}  
}
```

# 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**. Ceci permet **d'optimiser votre code** via :
- **Lazy loading** : N'instancie un service que s'il est injecté au moins une fois
- Permettre le **Tree-Shaking** des services non utilisés : Si le service n'est jamais utilisé, son **code sera entièrement retiré du build final**.

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

# @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.

# Les providers personnalisés

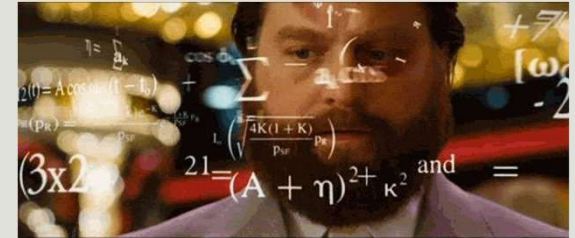
## la fonction inject (après Angular 14)

---

- La fonction inject vous permet d'injecter un injectable.
- **Avant Angular 14** et à partir **d'Angular 9**, la fonction **inject** pouvait être utilisé uniquement dans la **factory** de **l'InjectionToken** ou dans le factory du **@Injectable**.
- A partir d'Angular 14, vous pouvez **l'utiliser dans tout le context d'injection de dépendances comme vos composants, directives et pipes**.
- Le premier intérêt est le **type safety** avec le décorateur @Inject.
- Il **facilite aussi l'héritage** en externalisant la dépendance du composant.

```
export class CvComponent {  
  // Donne moi le sayHelloService  
  sayHelloService = inject(SayHelloService);  
  cvService = inject(CvService);  
}
```

# Exercice



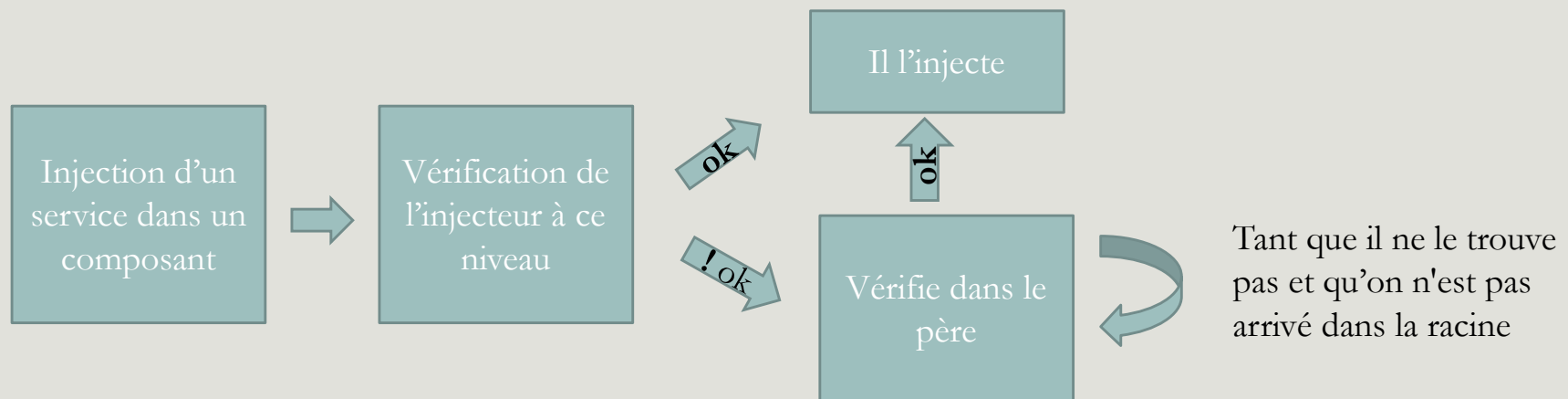
- Créons un service de Todo, le Model et le composant qui va avec. Un Todo est caractérisé par un nom et un contenu.
- Ce service permettra de faire les fonctionnalités suivantes :
  - Logger les todos
  - Ajouter un Todo
  - Récupérer la liste des Todos
  - Supprimer un Todo

Name:

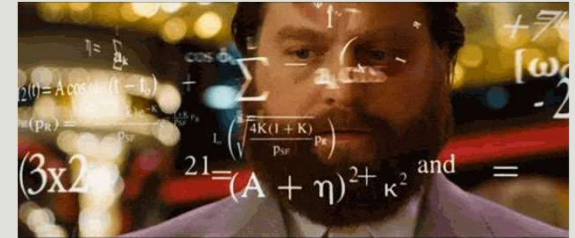
Content:

# DI Hiérarchique

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



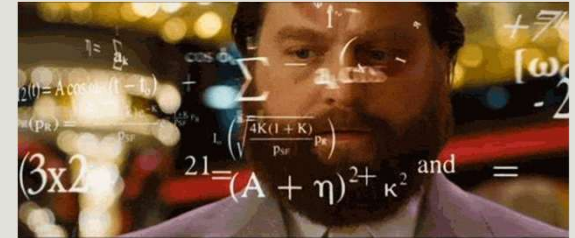
# Exercice



- Ajouter les services suivants afin d'améliorer la gestion de notre plateforme d'embauche.
  - Un premier **service CvService** qui gérera les Cvs. Pour le moment c'est lui qui contiendra la liste des cvs que nous avons.
  - Ajouter aussi un **composant** pour afficher la liste des cvs embauchées ainsi qu'un **service EmbaucheService** qui gère les embauches.
  - Au click sur le bouton embaucher d'un Cv, le cv est ajouté à la liste des personnes embauchées et une liste des embauchées apparait.
  - Si une personne est déjà embauchée, affiché un toast avertissant que la personne ne peut être sélectionnée qu'une fois. Vous pouvez utiliser la bibliothèque ngx-toastr : <https://www.npmjs.com/package/ngx-toastr>




# Exercice





CvTech

Home Cv Add Cv Todo Mini word Color Rôles Logout

FrancaisEnglish

Mandi Chaabane

Mendi Buh

Aymen Sellout

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

Job Description

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

235

Followers

114

Following

35

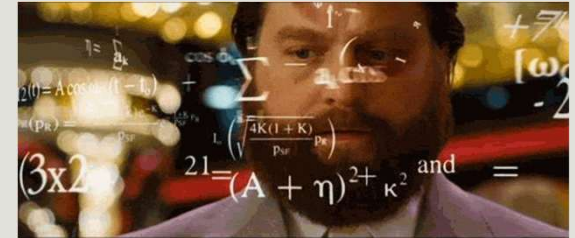
Projects

embaucher

détails

Footer

# Exercice



sellauti aymen



sellauti 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

aymen sellauti



# Angular Routing

---

AYMEN SELLAOUTI

# Objectifs

---

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

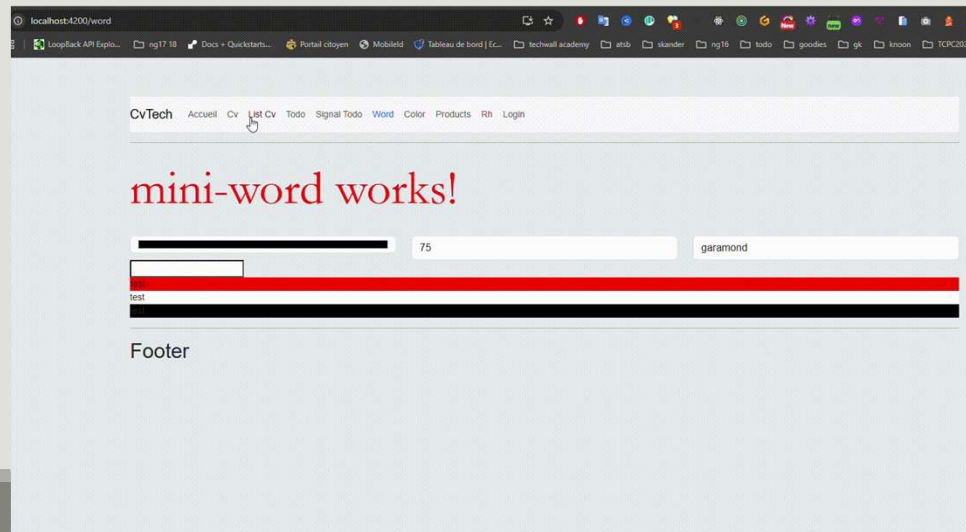
# Qu'est ce que le routing

---

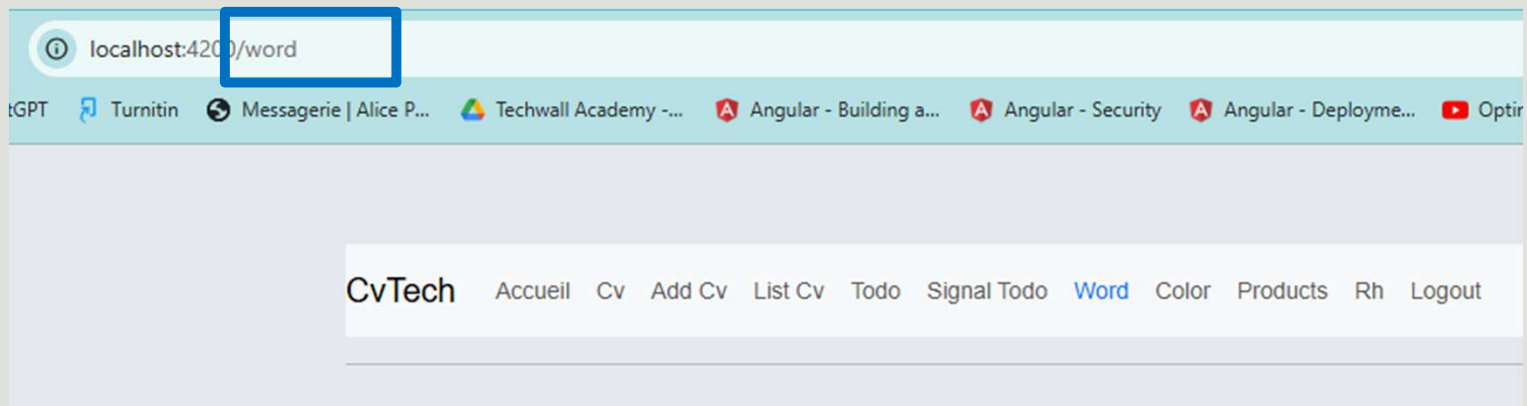
- Tout système de routing permet d'associer une URI à un traitement
- Angular est une SPA. Pourquoi parle-on de route ??
  - Séparer différentes fonctionnalités du système
  - Maintenir l'état de l'application
  - Ajouter des règles de protection
- Que risque t-on d'avoir si on n'utilise pas un système de routing ?
  - On ne peut plus rafraichir notre page
  - Plus de Favoris ☹
  - Comment partager vos pages ????

# Comment fonctionne le système de routing d'Angular et quels sont ses composantes

- Quand vous **naviguez dans un site web**, c'est votre URI qui vous permet de demander **quelle page (ressource) afficher**.
- A chaque fois que vous changez d'URI, Angular détecte ce changement et essaye de détecter à quoi correspond l'URI demandée. Pour ce faire nous avons besoin de deux choses :
  - Un module **responsable** de cette tâche : c'est le **RouterModule**
  - De définir cette **relation URI, Composant** pour que le RouterModule sache quoi faire : c'est votre **route**.



# Création d'un système de Routing



```
@NgModule({  
  imports: [  
    RouterModule.forRoot(routes),  
  ],  
  exports: [RouterModule],  
})  
export class AppRoutingModule {}
```

```
const routes: Routes = [  
  { path: word, component: FirstComponent },  
  { path: 'word', component: MiniWordComponent },  
  { path: 'color', component: ColorComponent },  
];
```

# Syntaxe minimaliste d'une route

---

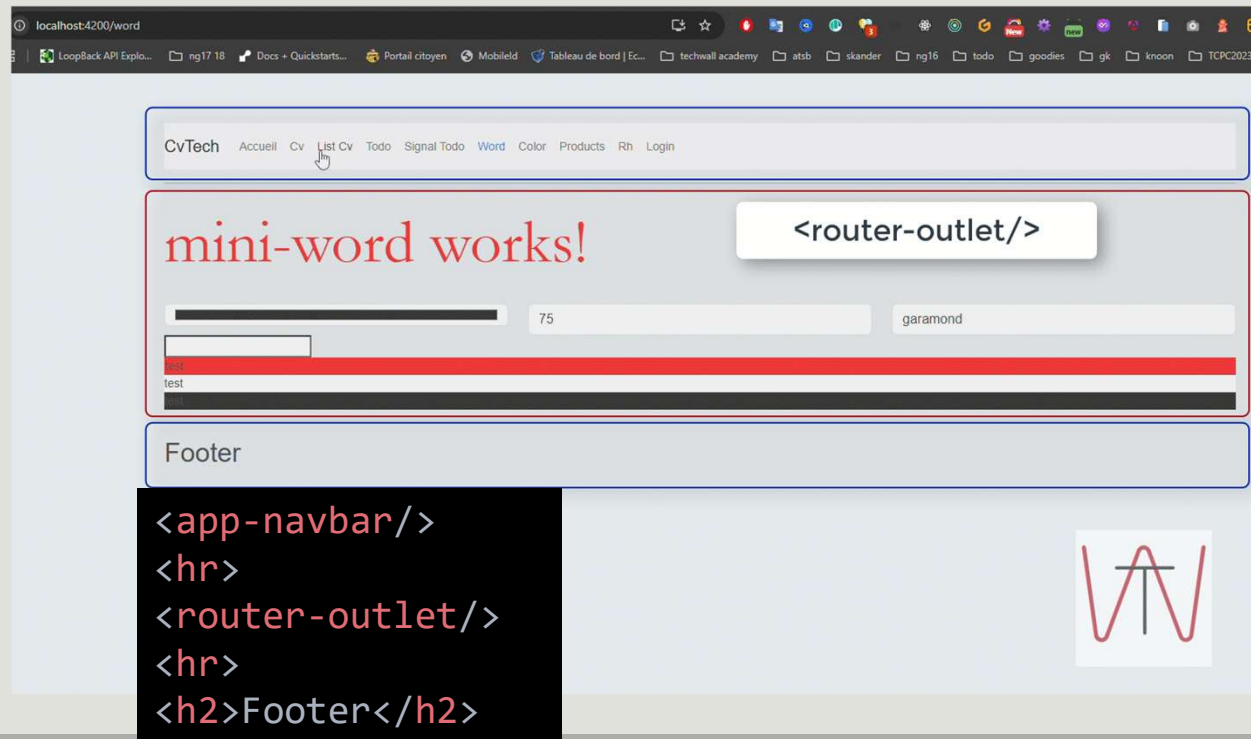
- Une route est un objet.
- Les deux propriétés essentielles sont `path` et `component`.
- `Path` représente l'URI
- `component` permet de spécifier le composant à exécuter.

```
const routes: Routes = [  
  { path: '', component: FirstComponent },  
  { path: 'word', component: MiniWordComponent },  
  { path: 'color', component: ColorComponent },  
];
```



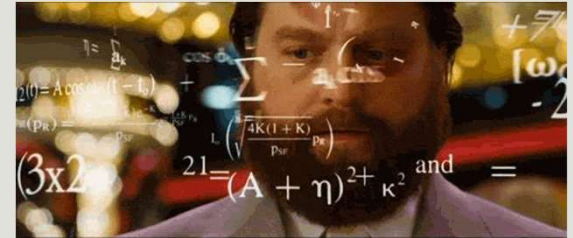
# Préparer l'emplacement d'affichage des vues correspondantes aux routes

- Généralement dans vos sites ou applications Web vous avez un template que vous suivez.
- Ce template contient une **partie statique** et une **partie variable**.
- La partie variable sera celle ou Angular devra affiché le composant associé à la route.
- Pour indiquer cette partie variable vous devez utiliser la directive `<router-outlet/>`.



# 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**.

```
<a [routerLink]="['todo']">Todo</a>
```

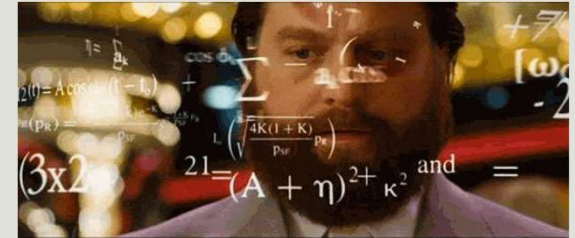
# Déclencher une route routerLink

---

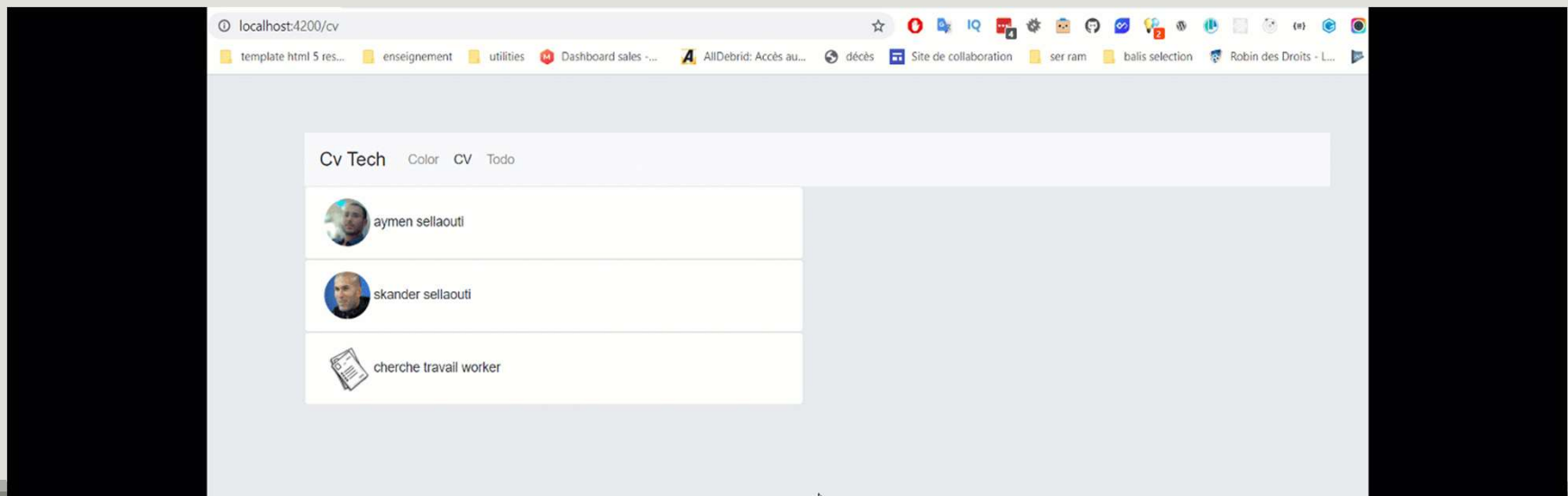
- Afin de mettre en avant la route sélectionnée par l'utilisateur dans votre menu, vous pouvez utiliser la directive **routerLinkActive**
- **routerLinkActive** prend en paramètre la **liste des classes CSS** que vous voulez appliquer à la **route sélectionnée ainsi qu'à tous ses ancêtres**.
- Par exemple si on a l'URI **/cv/liste** les classes de **routerLinkActive** seront ajoutées à cet URI ainsi qu'à l'URI **/cv** et **/**.
- Pour identifier uniquement l'uri cible, ajouter la directive **routerLinkActive** avec la valeur **{exact: true}**

```
<a class="nav-item nav-link"
  routerLinkActive="active text-primary"
  [routerLinkActiveOptions]="{exact: true}"
  [routerLink]="['']">Accueil</a>
```

# Exercice



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



# Déclencher une route à partir du composant

---

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

```
import { ActivatedRoute, Router } from "@angular/router";
router = inject(Router);
onLoadCv(id: number) {
  // méthode conseillée
  this.router.navigate(['cv', id]);
  //alternative
  this.router.navigate(['cv/${id}']);
}
```

# 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.

```
const routes: Routes = [  
  { path: '', component: FirstComponent },  
  { path: 'todo/:id', component: TodoDetailsComponent },  
  { path: 'color', component: ColorComponent },  
];
```

# Création d'un système de Routing

localhost:4200/todo/5

La troisième route ne contient qu'un seul segment et moi j'en cherche 2 ça ne peut pas être ça

Je cherche une route avec deux segments

- 1- Le premier segment si c'est une constante il doit être todo
- 2- Le deuxième segment c'est une constante égale à 5 ou il peut contenir n'importe quoi si c'est une variable

```
@NgModule({
  imports: [
    RouterModule.forRoot(routes),
  ],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

```
const routes: Routes = [
  { path: 'todo/5', component: FirstComponent },
  { path: 'color', component: ColorComponent },
  { path: 'todo/:id', component: TodoDetailsComponent },
];
```



# Déclencher une route routerLink

## routerLink : DIRECTIVE VS PROPERTY

---

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

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

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

# 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 de **"@angular/router"**.
  2. **Injecter ActivatedRoute** au niveau du composant.
  3. Utilisez l'objet **snapshot**

```
import { ActivatedRoute } from "@angular/router";

export class DetailsCvComponent {
  acr = inject(ActivatedRoute);
  constructor() {
    this.acr.snapshot.params;
  }
}
```

# Récupérer les paramètres d'une route

## ActivatedRoute / snapshot

---

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

# Récupérer les paramètres d'une route ActivatedRoute / snapshot

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

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

# Récupérer les paramètres d'une route

## ActivatedRoute / snapshot

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

```
▼ snapshot.params:  
  id: "662"
```

```
import { ActivatedRoute } from "@angular/router";  
  
export class DetailsCvComponent {  
  acr = inject(ActivatedRoute);  
  constructor() {  
    console.log({'snapshot.params': this.acr.snapshot.params});  
  }  
}
```

# Passer le paramètre à travers le tableau de routerLink

---

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

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

# Déclencher une route routerLink

## Route relative Vs Route absolue

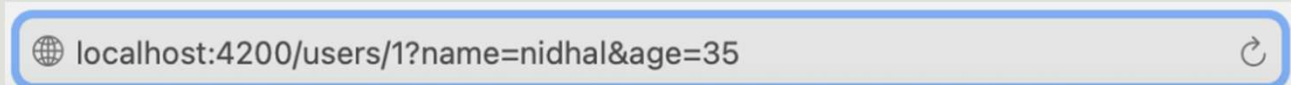
---

- Quand vous définissez la route vers laquelle vous voulez naviguer, si on **préfixe la valeur passé au routerLink** avec :
  - **'/'**, la route sera **absolue**.
  - **Rien ou './'**, le Router **regardera dans les enfants du ActivatedRoute**.
  - **'../'**, le Router regardera dans le niveau au-dessus dans l'arbre de routes.
- Quand on désire passer un path relativement à la route dans laquelle on est, il est possible de passer à la méthode `navigate`, en argument, un objet dont la clé `relativeTo` avec comme valeur est une instance de `ActivatedRoute`, informe sur le fait qu'on veut une route relative à la route active.

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

# 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** content les identifiants des queryParams et leurs valeurs.



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



# Les queryParameters

---

- La deuxième méthode est en l'intégrant à notre routerLink de la manière suivante :

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

# Récupérer Les queryParameters

---

- Les **queryParameters** sont récupérable de la même façon que les paramètres. Soit d'une façon statique avec **snapshot via la propriété queryParams** ou sa propriété **queryParamMap** et sa méthode **get**.
- Soit dynamiquement via l'observable **queryParams**

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

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

# La route joker

---

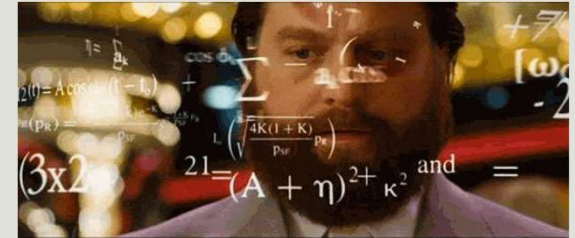
- Il existe une route **joker** qui **matche n'importe quelle autre route**.  
C'est la route **'\*\*'**.

# Exemple

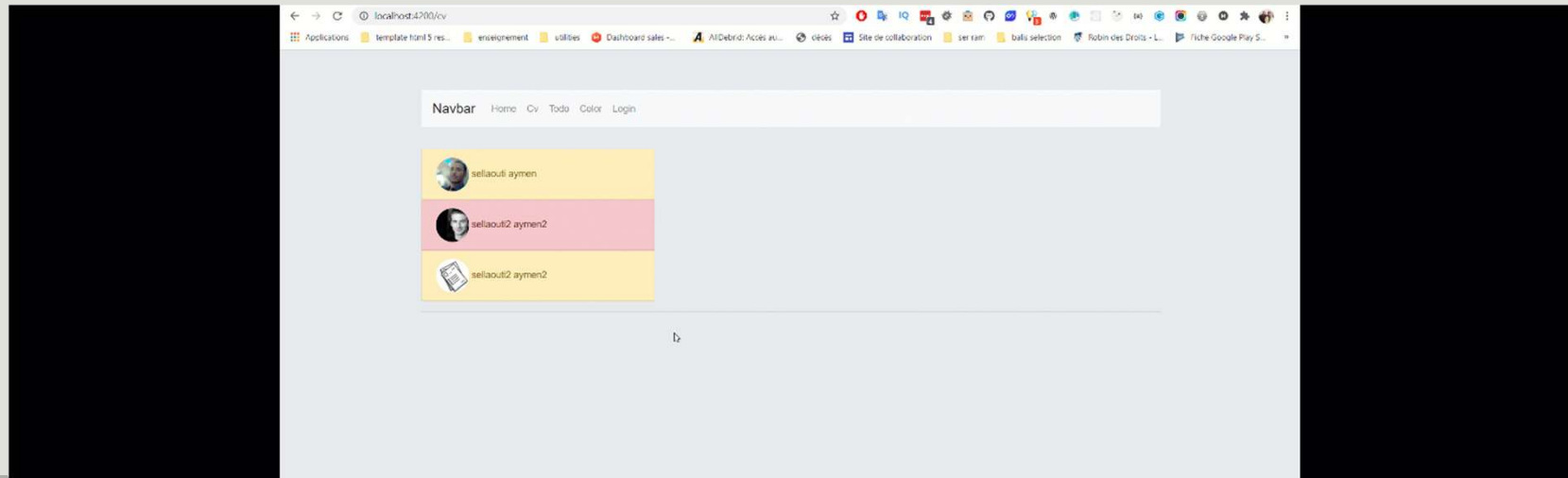
---

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

# Exercice



- Ajouter les fonctionnalités suivantes à votre cvTech:
  - Une page détail qui va afficher les détails d'un cv.
  - Un bouton dans chaque cv qui au click vous 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

# 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.

# Approche basée Template

## NgForm

---

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

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

# Approche basée Template

## Les classes CSS auto-générée

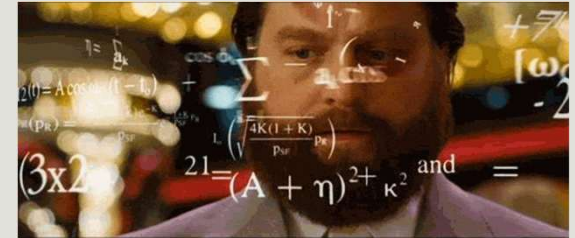
---

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

[illegible]

- 134

# Exercice



localhost:4200/login

Applications template.html 5 res... enseignement utilities Dashboard sales ~... A AllDebrid: Accès au... vidéos Site de collaboration ser ram balls selection Robin des Droits - L... Fiche Google Play S...

Navbar Home Cv Todo Color Login

Email :

password :

Login

# Approche basée Template

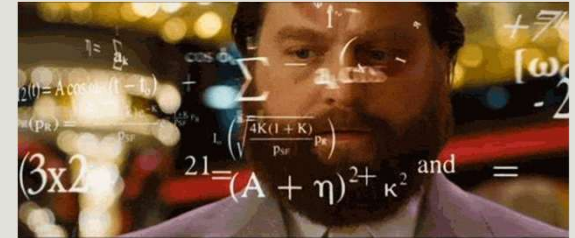
## Accéder aux propriétés d'un champ (contrôle) du formulaire

---

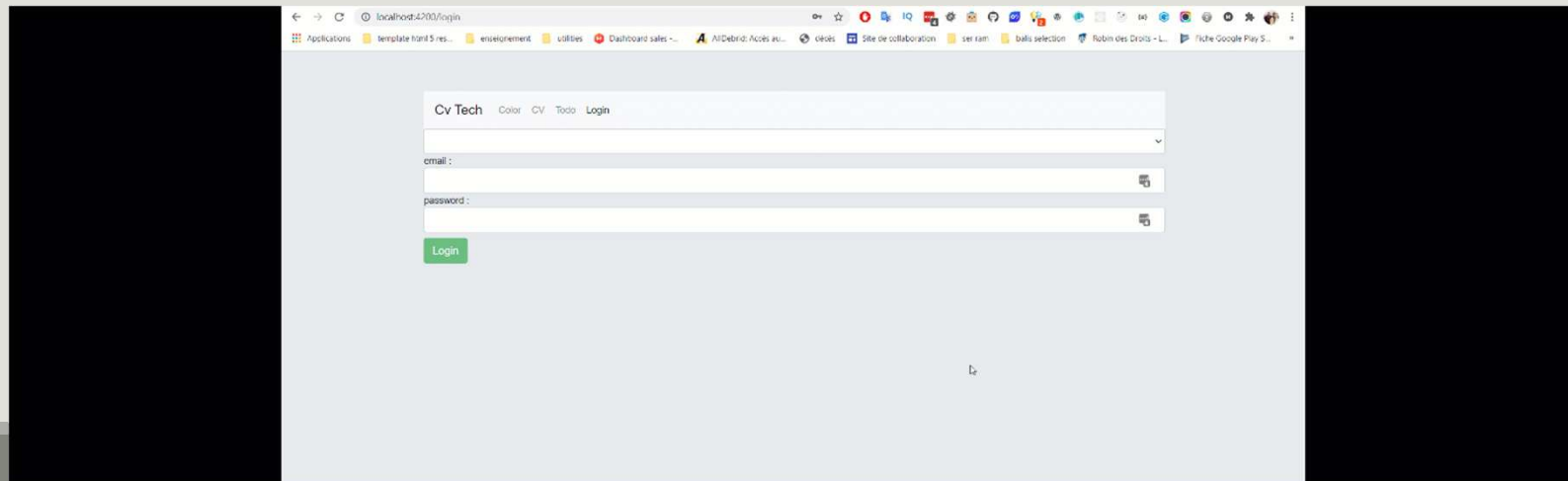
- Pour accéder à l'objet form et ces propriétés nous avons utilisé `#notreForm=«ngForm »`
- Pour les champs du formulaire c'est la même chose mais au lieu du ngForm c'est un `ngModel`  
`#notreChamp=« ngModel »`



# Exercice



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



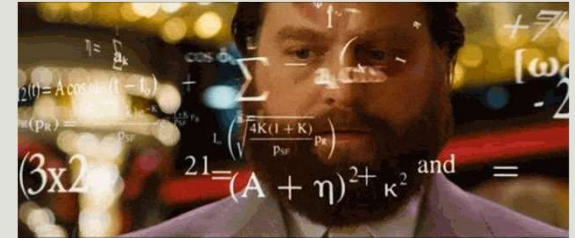
# Approche basée Template

## Associer des valeurs par défaut aux champs

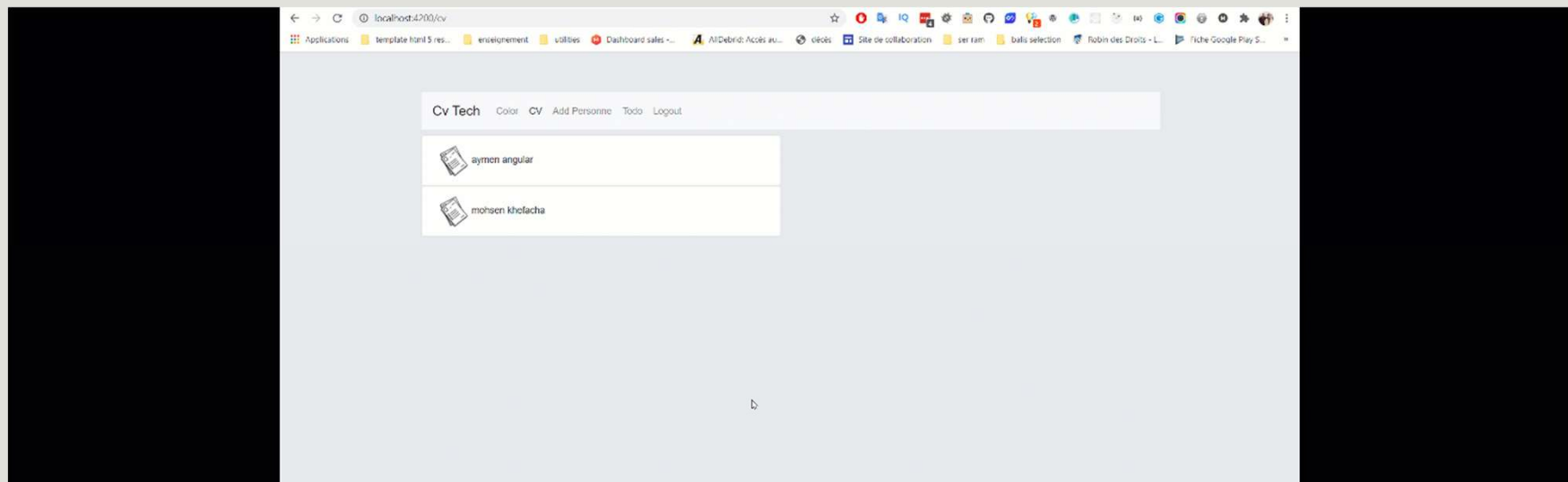
---

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

# Exercice



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



# Angular HTTP et Déploiement

---

AYMEN SELLAOUTI

# Objectifs

---

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

# HTTP

---

- Angular est un Framework FrontEnd
- Pas d'accès à la BD
- Pas de possibilité de persistance des données
- Pas de puissance permettant des traitements lourds.

Le Module HTTPClient

# Programmation Asynchrone

---

Programmation non bloquante.

# Les promesses

---

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

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

- Le fonctionnement des promesses est le suivant :
  - On crée une promesse.
  - La promesse va toujours retourner deux résultats :
    - resolve en cas de succès
    - reject en cas d'erreur
  - Vous devrez donc gérer les deux cas afin de créer votre traitement



# Promesse

---

```
const promise = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve(3);  
  }, 5000);  
});  
promise.then(function (x) {  
  console.log('resolved with value :', x);  
});
```

# Qu'est ce que la programmation réactive

---

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

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

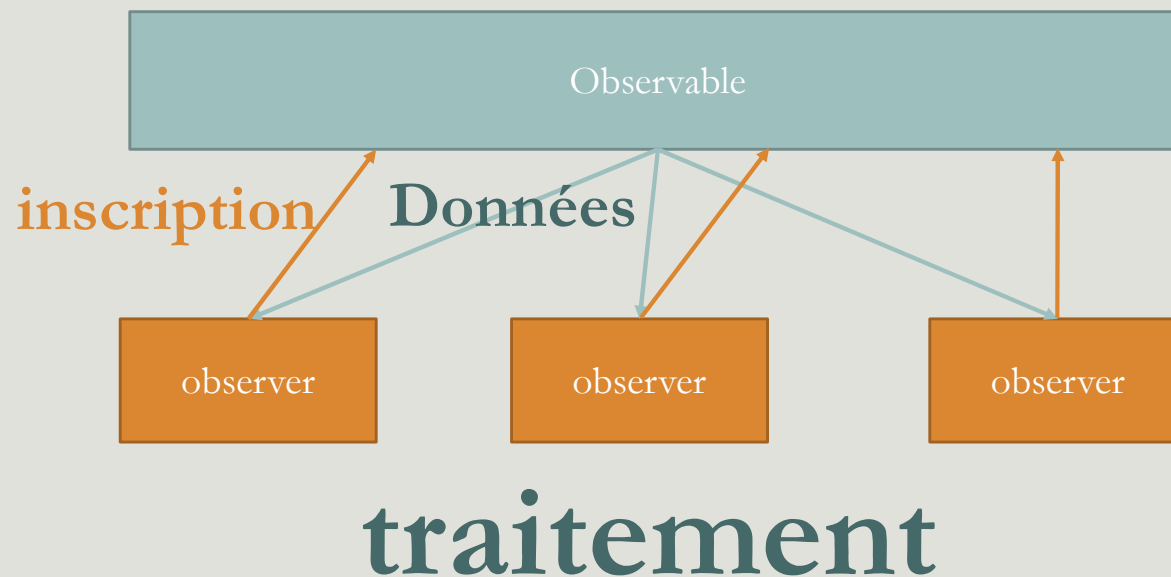
# Le pattern « Observer »

---

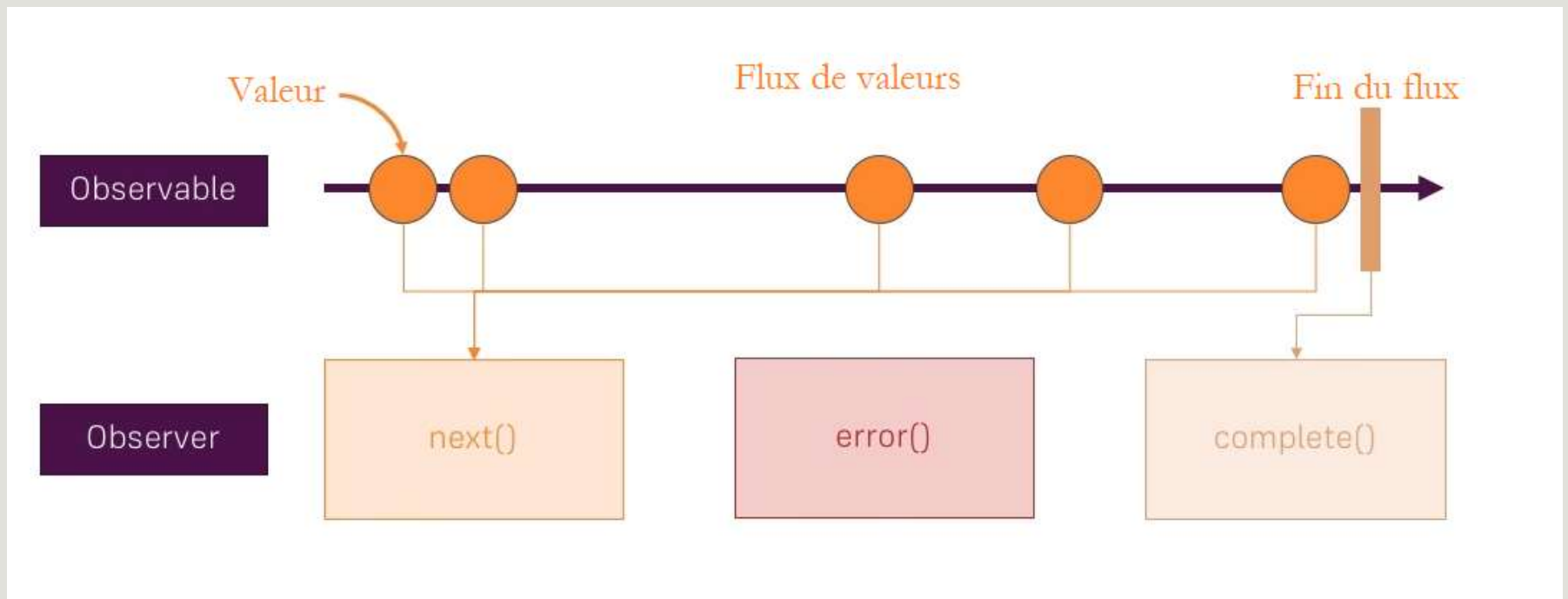
- Le patron de conception **Observateur** permet à un objet de garder la trace d'autres objets, intéressés par l'état de ce dernier.
- Il définit une relation entre objets de type un-à-plusieurs.
- Lorsque l'état de cet objet **change**, il **notifie** ces **observateurs**.

# 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   |

# S'inscrire à un observable

---

- Afin de **montrer votre intérêt à un flux Observable**, vous pouvez utiliser la méthode **subscribe** qu'il vous offre.
- En vous inscrivant à cet observable, vous pouvez **passer à cette méthode** un **objet qui a 3 fonctions** :
  - **next**, qui sera **appelé à chaque fois qu'une nouvelle valeur arrive dans le flux**. Elle prend en **paramètre cette valeur** et vous permet d'implémenter ce que vous voulez faire avec.
  - **Error, déclenché une fois en cas d'erreur** qui vous permet d'implémenter le comportement en réaction à cette erreur
  - **Complete, déclenché dès la fin du flux** qui vous permet d'implémenter le comportement en réaction à la fin du flux.

# S'inscrire à un observable

---

```
myObservable$.subscribe({  
  next: (data) => {  
    //TODO: implementez le comportement quand vous recevez les données  
  },  
  error: (error) => {  
    //TODO: implementez le comportement quand vous avez une erreur  
  },  
  complete: () => {  
    //TODO: implementez le comportement à la fin du flux  
  },  
});
```



# Exemple d'un Observable

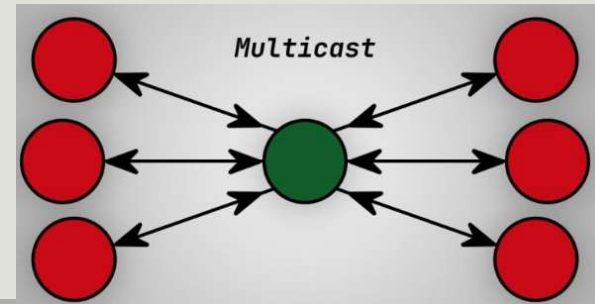
## Création et inscription

```
export class TestObservableComponent {  
  myObservable$: Observable<number>;  
  constructor() {  
    this.myObservable$ = new Observable((observer) => {  
      let i = 5;  
      const intervalIndex = setInterval(() => {  
        if (!i) {  
          observer.complete();  
          clearInterval(intervalIndex);  
        }  
        observer.next(i--);  
      }, 1000);  
    });  
    this.myObservable$.subscribe({  
      next: (val) => {  
        console.log(val);  
      },  
    });  
  }  
}
```



# Hot Vs Cold Observable

- Les **Cold Observables** commencent à émettre des valeurs uniquement quand on s'y inscrit. Les **Hot observables**, par contre émettent toujours.
- Les **Cold Observables** diffusent un flux par inscrit, ils sont **unicast**. Chaque nouvelle inscription crée un **nouveau contexte d'exécution**.
- Les **Hot observables**, sont **multicast**, le **même flux est partagé par tous les inscrits**.
- Dans les **Cold Observables**, la **source de données est à l'intérieur** de l'observable.
- Dans les **HotObservables**, la **source de données est à l'extérieur** de l'observable.



# asyncPipe

---

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

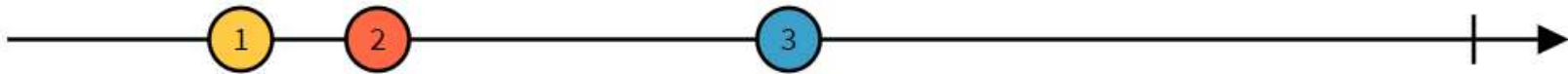
# Les operateurs de l'observable

---

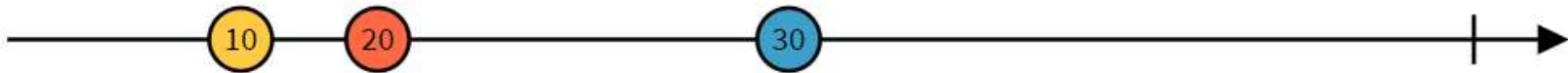
- Les opérateurs sont des fonctions. Il y a deux types d'opérateurs :
- **Un opérateur pipeable** est une fonction qui prend un observable comme entrée et renvoie un autre observable. C'est une opération pure : le précédent Observable reste inchangé.
  - Syntaxe : `monObservable.pipe(opertaeur1(), operateur2(),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

# Quelques opérateurs utiles de l'Observable

## map

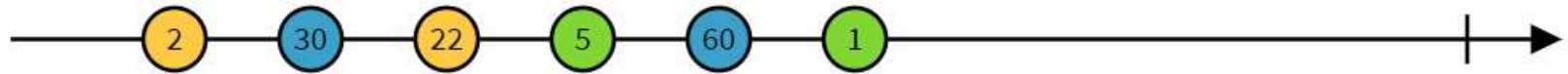


`map(x => 10 * x)`



# Quelques opérateurs utiles de l'Observable

## filter



```
filter(x => x > 10)
```



# Les opérateurs de création

## from

---

- **from** est utilisé pour **convertir des types d'objets JavaScript** comme un **tableau** ou **un objet itérable** en une séquence **observable** de valeurs.
- L'opérateur émet également une **chaîne** sous forme de **séquence de caractères**.

```
from([1, 2, 3]).subscribe({  
  next: (data) => console.log('[from]', data),  
  complete: () => console.log('[from] complete'),  
});
```

|        |          |
|--------|----------|
| [from] | 1        |
| [from] | 2        |
| [from] | 3        |
| [from] | complete |

# Les operateurs de création of

---

- l'opérateur **of** est utilisé afin de **convertir un argument en un observable**
- il ne fait **aucun aplatissement ou conversion** et **émet** chaque argument sous le **même type qu'il reçoit**
- of est couramment utilisé lorsque vous avez simplement **besoin de renvoyer une valeur là où un observable est attendu** ou de **démarrer une chaîne observable**.

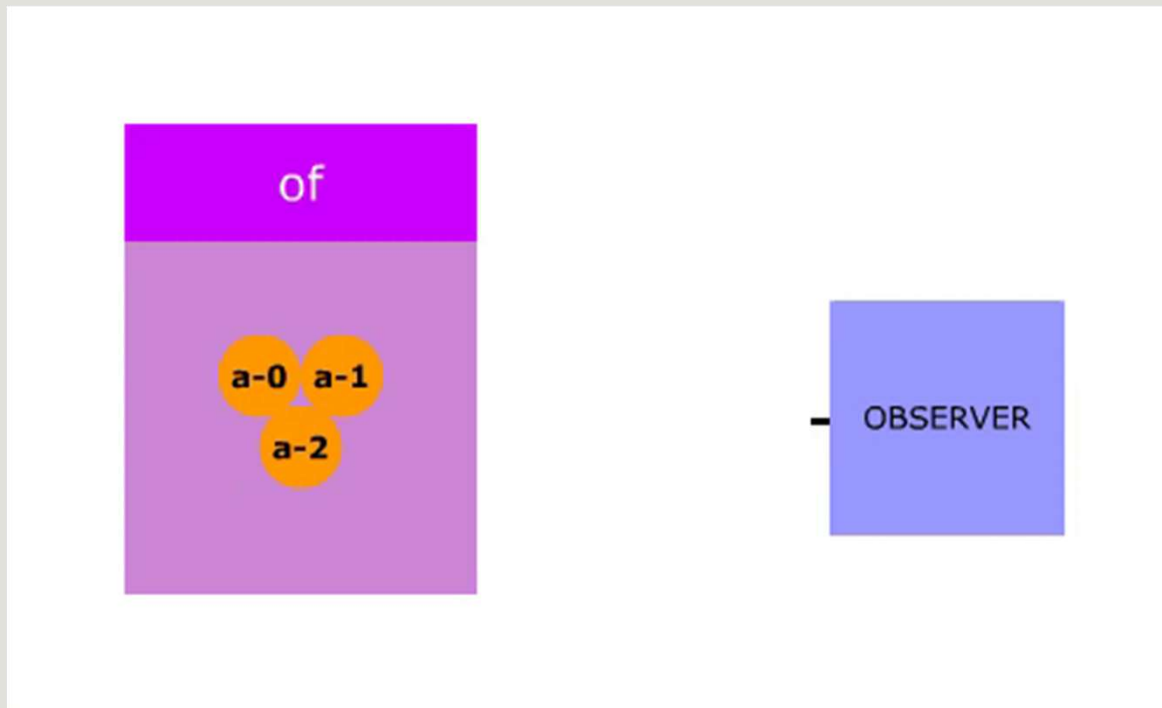
```
of([1, 2, 3], new Date(), {  
  name: 'sellaouti',  
  firstname: 'aymen',  
}).subscribe({  
  next: (data) => console.log('[of]', data),  
})
```

```
[of] ▶ (3) [1, 2, 3]  
[of] Tue Jan 03 2023 13:46:43 GMT+0100 (West Africa Standard Time)  
[of] ▶ {name: 'sellaouti', firstname: 'aymen'}  
[of] complete
```



# Les opérateurs de création of

---



# Les operateurs de création

## tap

---

- L'opérateur **tap** est utilisé pour gérer les effets de bord.
- En effet, il n'a aucun effet sur le flux de données et il vous permet de faire des opérations mais sans toucher au flux.

```
from([1, {name: 'aymen'}, 3])  
  .pipe(  
    tap((data) => console.log(data))  
  )  
)
```

# Les operateurs de création timer

- L'opérateur **timer** est un opérateur de création utilisé pour créer un observable qui commence à émettre les valeurs après un délai d'attente, et la valeur continuera d'augmenter après chaque appel.
- Il prend en **entrée** en **premier paramètre quand déclencher** l'événement.
- En **second paramètre** en cas de volonté de répétition chaque **combien de millisecondes reprendre l'émission**.

```
timer(1000).subscribe((val) => console.log('[timer] : '+ val));
```

```
[timer] : 0
```

```
timer(1000, 1000).subscribe((val) => console.log('[timer] : '+ val));
```

```
[timer] : 0
```

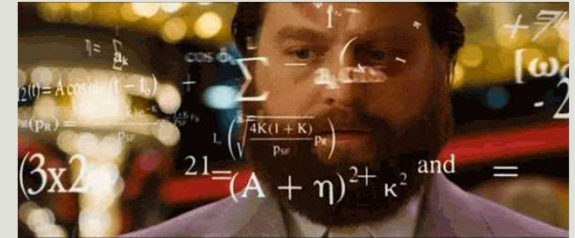
```
[timer] : 1
```

```
[timer] : 2
```

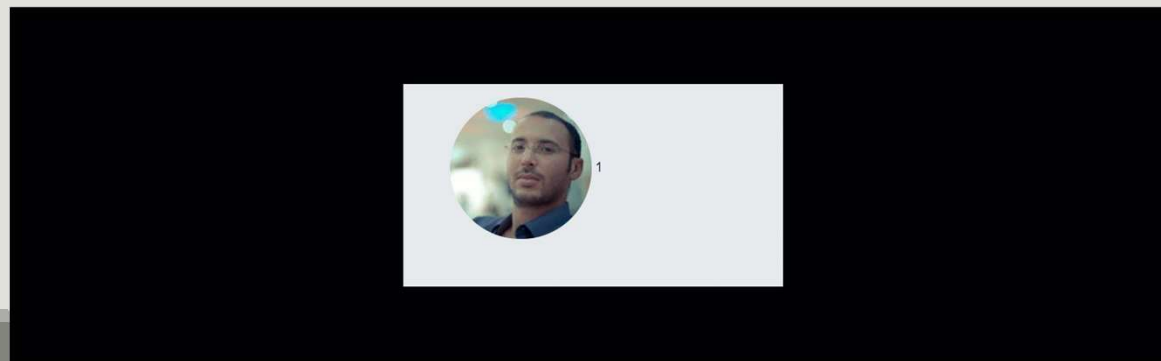
```
[timer] : 3
```

```
[timer] : 4
```

# Exercice



- Ecrire un composant qui affiche une suite d'images non stop.
- Utiliser un observable comme source d'images.
- Vous devez uniquement utiliser des opérateurs pour faire le travail.
- La taille de l'image, la liste des images et le temps entre chaque image doit être paramétrable.



# Quelques opérateurs utiles de l'Observable

---

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

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

<http://rxmarbles.com/>

# Les subjects

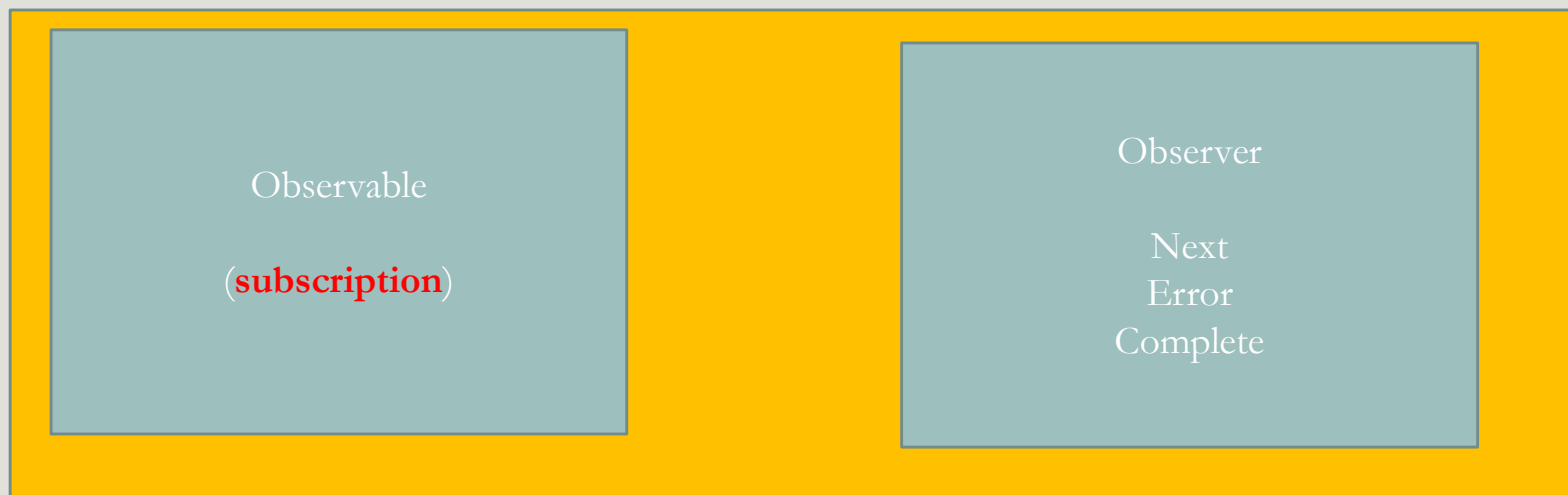
---

- Afin de créer des **flux chauds**, RxJs nous fournit une structure de données appelées **Subject**.
- Pensez au Subject comme à **un flux que vous créer au fur et à mesure** que vous recevez les données.
- C'est une **création en temps réel**, vous **n'avez aucune information au préalable** sur le contenu du flux.
- C'est comme la **transmission d'un match à la télé en direct**. Votre chaine ne sait pas quelles images elle va recevoir, elle ne les a pas en mémoire sous forme d'une vidéo, à chaque fois qu'elle recoit une image de la source, elle vous la transmet.

# Les subjects

---

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



# Les subjects

- Afin de créer un flux chaud commencer par **instancier un Subject**. Ceci vous permet d'avoir un flux vide auquel on peut s'inscrire.

```
nbUser$ = new Subject<Number>();
```

- Pour ajouter une valeur dans le flux, il suffit d'appeler la méthode next.

```
this.nbUser$.next(1);
```

1

```
this.nbUser$.next(2);
```

1

2



# Les subjects

---

- Si vous êtes intéressé par ce flux, c'est un flux comme un autre, il vous suffit donc de vous inscrire.

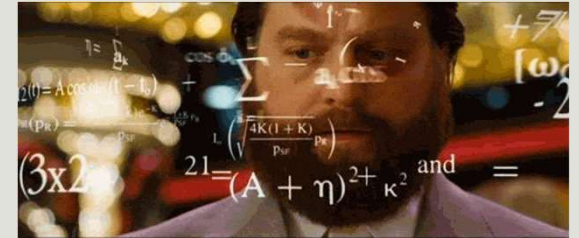
```
nbUser$.subscribe({  
  next:(nb) => {},  
  error:(e) => {},  
  complete:() => {},  
})
```

# Les subjects

---



# Exercice



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

# Installation de HTTP

---

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

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

# Installation de HTTP

---

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

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

# Interagir avec une API Get Request

- Afin d'exécuter une requête **get** le module http nous offre une méthode **get**.
- Cette méthode retourne un **Observable**.
- Inscrivez vous à cet observable et définissez le comportement en cas de succès, erreur ou complétion

```
this.http.get(API_URL).subscribe({  
  next: (response:Response)=>{  
    //ToDo with DATA  
  },  
  error: (err:Error)=>{  
    //ToDo with error  
  },  
  complete: () => {  
    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

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

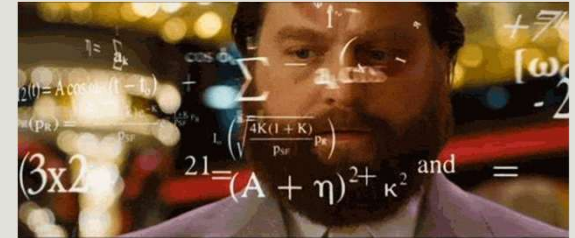
# Documentation

---

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



# Exercice



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

# Les headers

---

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

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

- Elle propose une panoplie de méthode helpers permettant de la manipuler.
- `set(clé,valeur)` permet d'ajouter des headers. Elle écrase les anciennes valeurs.
- `append(clé,valeur)` concatène de nouveaux headers.

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

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

# Les paramètres

---

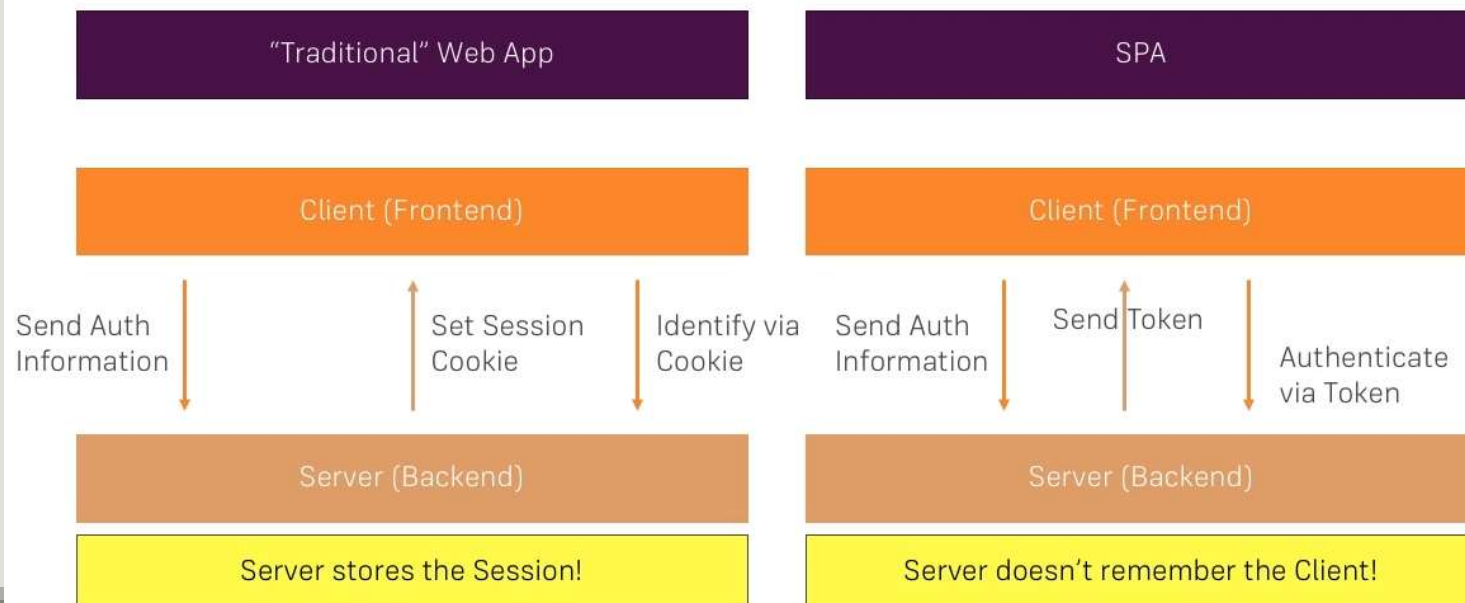
- Afin d'ajouter des paramètres à vos requêtes, le HttpClient vous offre la classe HttpParams.
- Cette classe est une classe immuable (read Only).
- Elle propose une panoplie de méthode helpers permettant de la manipuler.
- `set(clé,valeur)` permet d'ajouter des headers. Elle écrase les anciennes valeurs.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

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

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

# Authentication

## How does Authentication work?



# Ajouter le token dans la requête

---

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

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

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

# Ajouter le token dans la requête

---

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

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

# Sécuriser vos routes

---

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

# Guard

---

- Ce sont des classes qui permettent de gérer l'accès à vos routes.
- Un guard informe sur la validité ou non de la continuation du process de navigation en retournant un booléen, une promesse d'un booléen ou un observable d'un booléen.
- Le routeur supporte plusieurs types de guards, par exemple :
  - `CanActivate` permettre ou non l'accès à une route.
  - `CanActivateChild` permettre ou non l'accès aux routes filles.
  - `CanDeactivate` permettre ou non la sortie de la route.



# Guard / canActivate

---

- Afin d'utiliser le guard `canActivate` (de même pour les autres), vous devez créer une classe qui implémente l'interface `CanActivate` et donc qui doit implémenter la méthode `canActivate` de sorte qu'elle retourne un booléen permettant ainsi l'accès ou non à la route cible. 1
- Vous devez ensuite ajouter cette classe dans le provider. 2
- Finalement pour l'appliquer à une route, ajouter la dans la propriété `canActivate`. Cette propriété prend un tableau de guard. Elle ne laissera l'accès à la route que si la totalité des guard retourne true. 3
- Vous pouvez utiliser la méthode : `ng g g nomGuard`

# Guard / canActivate

1

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

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

# Guard

---

- A partir d'Angular 14 et la possibilité d'utiliser la fonction inject dans tous les contextes d'injection, Angular préconise les fonctionnels Guards.

```
export const myGuard: CanActivateFn = (route, state) => {  
  return true;  
};
```

# Guard / canActivate

---

2

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

App.module.ts

# Guard / canActivate

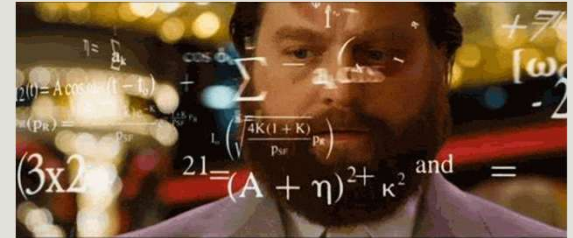
---

3

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

# Exercice

---

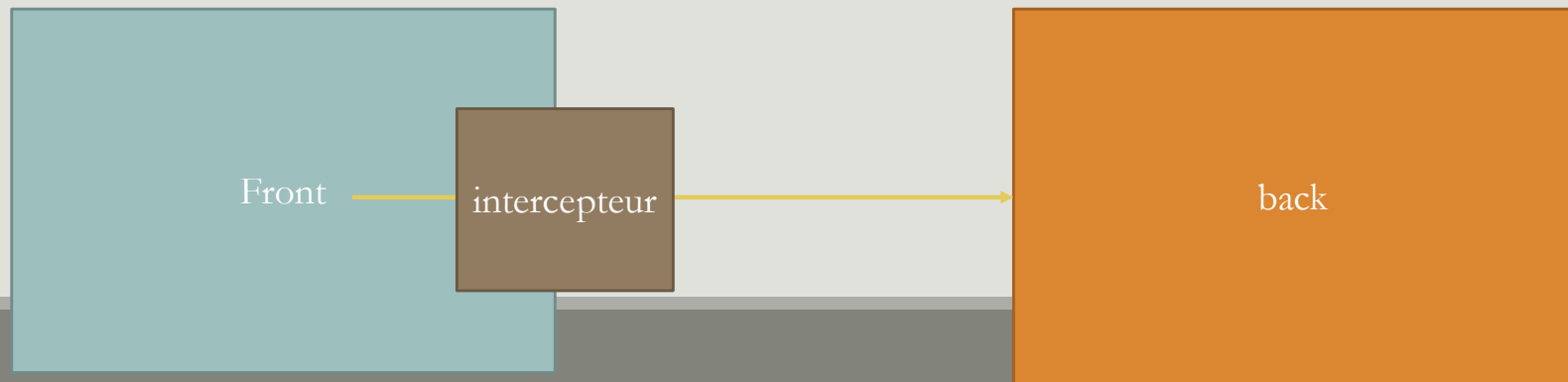


- Ajouter les guards nécessaires afin de sécuriser vos routes.
- Une personne déjà connectée ne peut pas accéder au composant de login.

# Les intercepteurs

---

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



# Les intercepteurs

---

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor, } from '@angular/common/http';
import { Observable } from 'rxjs';
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept( request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    return next.handle(request);
  }
}
```



# Les intercepteurs

---

- Un intercepteur est injecté au niveau du provider en ajoutant un **objet décrivant votre intercepteur**.
- Dans cet objet renseigner le Token **HTTP\_INTERCEPTOR** via la clé **provide**
- **Associer le à votre intercepteur en utilisant la clé useClass**
- Ajouter à cet objet une **clé multi** qui aura comme **valeur true**, elle permettra de **provider plusieurs intercepteurs**.

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

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

# Les intercepteurs : changer la requête

---

- Par défaut la **requête est immutable**, on **ne peut pas la changer**.
- Solution : la **cloner**, changer les headers du clone et le renvoyer.

```
const cloneReq = request.clone({  
  setHeaders: {  
    'Authorization': token  
  } });  
// Chainer la nouvelle requete avec next.handle  
return next.handle(newReq);
```

# Déploiement

---

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

`ng build`

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`

# Angular

# Les modules

---

AYMEN SELLAOUTI

# 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**.
- Une application Angular contient **au moins un Module** : AppModule.
- Un Module Angular peut contenir des composants, des providers de service...
- Une application simple est généralement composée d'un seul module. Par contre dès que votre application grandit penser à la séparer en Modules.
- Chaque module **vie séparée** des autres modules. Par défaut **il n'expose rien**, tout ce qui est à l'intérieur du module **reste uniquement dans le module tant qu'on ne l'exporte pas**.
- Lorsqu'on importe un module, on **importe réellement tout ce qu'il exporte**.

# 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 { }
```

# 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 accessible que pour les composants, directives et les pipes déclarés dans le même module.
- Pour utiliser un de ces éléments à l'extérieur du module, 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`.

# Exemple pour le TodoModule

```
import { CommonModule } from "@angular/common";
import { NgModule } from "@angular/core";
import { FormsModule } from "@angular/forms";

import { TodoComponent } from "../todo.component";
import { TodoRouting } from "../todo.routing";

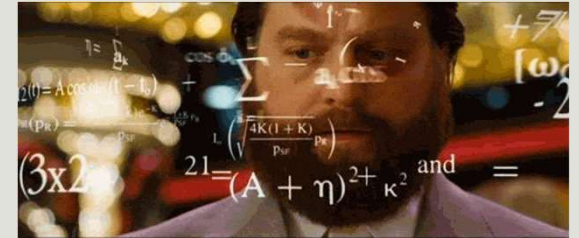
@NgModule({
  declarations: [TodoComponent],
  imports: [CommonModule, FormsModule, TodoRouting],
  // Si vous n'avez pas besoin de TodoComponent à l'extérieur,
  // ne l'exporter pas
  exports: [TodoComponent],
})
export class TodoModule {}
```

```
import { NgModule } from "@angular/core";
import { Route, RouterModule } from "@angular/router";
import { NF404Component } from "../nf404/nf404.component";
import { TodoComponent } from "../todo.component";

const routes: Route[] = [
  { path: "todo", component: TodoComponent },
  { path: "**", component: NF404Component },
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
export class TodoRouting {}
```

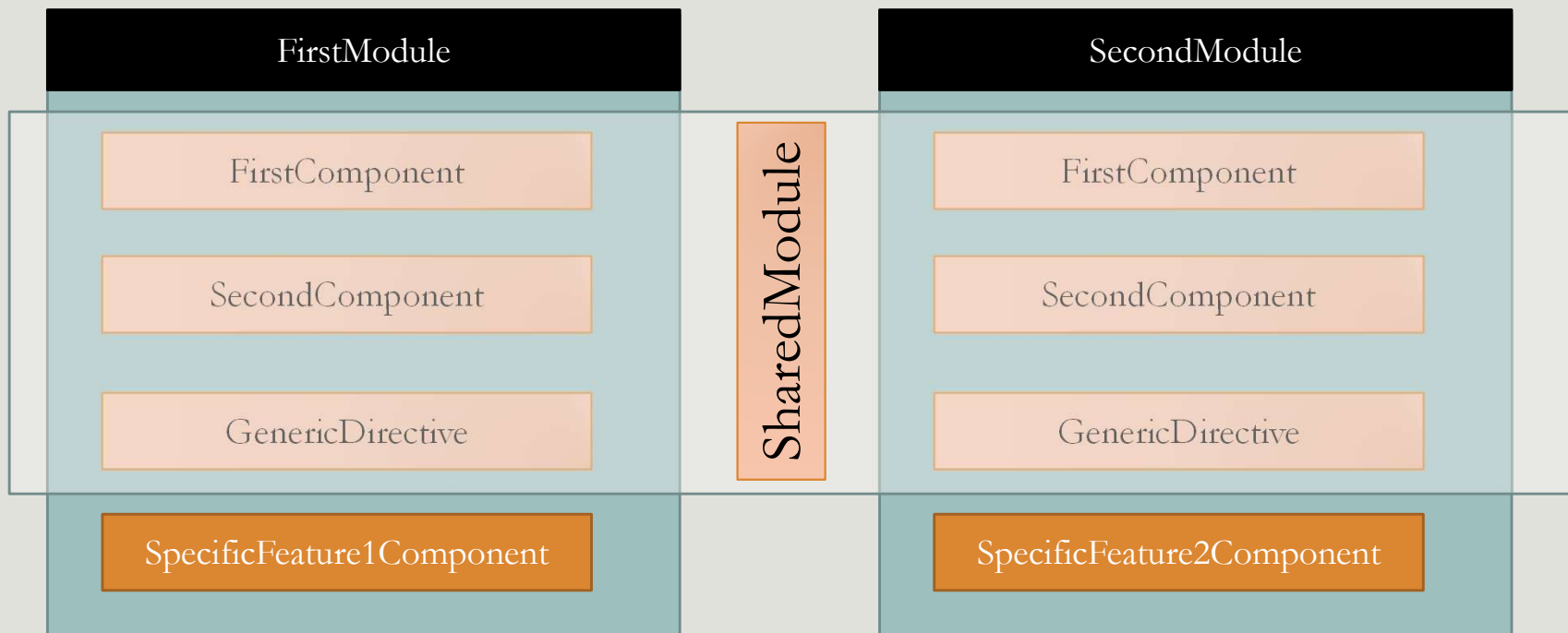
# Exercice



- Implémentez le CvModule.

# Shared Module

---



# Shared Module

---

```
@NgModule({
  declarations: [
    FirstComponent,
    SecondComponent,
    GenericDirective,
  ],
  imports: [
    CommonModule
  ]
  exports: [
    FirstComponent,
    SecondComponent,
    GenericDirective,
    CommonModule
  ]
})
export class SharedModule {
}
```

# Lazy Loading

- Par défaut, tous les modules que vous déclarez au niveau du AppModule sont chargés au lancement de l'application.
- Ceci pose un problème au niveau du Bundle généré de votre application.
- Une grande application aura une taille assez conséquente ce qui peut provoquer un problème au chargement de l'application et donc un problème d'expérience utilisateur.
- L'idée du lazyLoading et de **charge au départ le module principale** et puis de **ne charger un module que si on appelle l'une de ses routes**.
- Ceci va nous faire gagner en performance.

The screenshot shows a web browser at localhost:4200. The page has a header 'Cv Tech' and a file upload section with buttons 'Choisir un fichier', 'Aucun fichier choisi', and 'Upload'. Below this, the text 'front works!' is displayed. The Chrome DevTools Network tab is open, showing a list of requests. The 'personnes' request is highlighted in red, indicating a failure.

| Name         | Status   | Type    | Initiator | Size  | Time    | Waterfall |
|--------------|----------|---------|-----------|-------|---------|-----------|
| localhost    | 304      | docu... | Other     | 210 B | 298 ... |           |
| runtime.js   | 200      | script  | (index)   | 211 B | 13 ms   |           |
| polyfills.js | 200      | script  | (index)   | 212 B | 13 ms   |           |
| styles.js    | 200      | script  | (index)   | 212 B | 180 ... |           |
| vendor.js    | 200      | script  | (index)   | 213 B | 302 ... |           |
| main.js      | 200      | script  | (index)   | 212 B | 26 ms   |           |
| personnes    | (failed) | xhr     | VM3716:1  | 0 B   | 2.02 s  |           |

# Lazy Loading

---

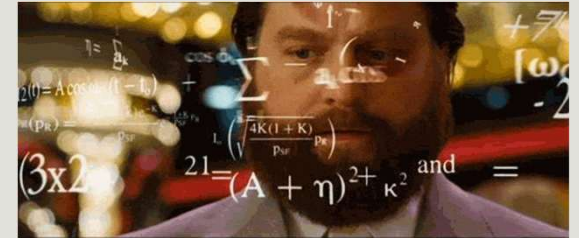
➤ Afin d'implémenter le *Lazy Loading*, on doit suivre les étapes suivantes :

1. Le **module** à charger doit **lui-même gérer sa partie routing**
2. Au niveau du routing principal (AppRoutingModule), créer une **route**, ajouter lui un path 'ca sera le **préfixe** de toutes les routes du module', et ajouter une nouvelle clé qui est **loadChildren**. Cette clé va informer angular et lui demander de ne charger le module associé que lorsque on appelle le path défini.
3. Ce **paramètre** prend ou une **chaîne de caractère** qui spécifie le module à charger ou une callback function.
4. Finalement **enlever les imports des modules lazy loaded** au niveau du AppModule

```
{  
  path: "cv",  
  loadChildren: "./cv/cv.module#CvModule",  
},
```

```
{  
  path: "cv",  
  loadChildren: () => import('./cv/cv.module').then(  
    m => m.CvModule  
  ),  
},
```

# Exercice



- ## ➤ Testez le lazy loading avec le CvModule



# Preloading Lazy Loading

---

- Le **problème** qu'on peut identifier avec le **lazy loading** est le fait qu'en passant d'un module à un autre on aura **toujours un chargement des nouveaux modules**.
- Si vos **modules** sont **très volumineux** ou que la connexion du client est mauvaise, il y aura **plusieurs latences**. Ceci va provoquer un problème avec l'utilisateur.
- La question qui se pose est : **Y a-t-il un moyen de personnaliser les stratégies de chargement** ???

# Preloading Lazy Loading

- La méthode **forRoot** de votre RouterModule prend en **second paramètre un objet** vous permettant de **configurer** la **stratégie de chargement** avec la propriété **preloadingStrategy**.
- Cette propriété prend en paramètre, par défaut **NoPreloading**.
- La deuxième valeur qu'elle peut prendre est **PreloadAllModules**. Elle demande à Angular de **précharger** tous les **lazyLoaded Modules** une fois le **Module principal chargé**.

```
import { PreloadAllModules } from "@angular/router";
@NgModule({
  imports: [RouterModule.forRoot(routes, {
    preloadingStrategy: PreloadAllModules
  })],
  exports: [RouterModule],
})
```

|                     |          |        |               |       |        |  |
|---------------------|----------|--------|---------------|-------|--------|--|
| vendor.js           | 200      | script | (index)       | 213 B | 270 ms |  |
| main.js             | 200      | script | (index)       | 212 B | 25 ms  |  |
| personnes           | (failed) | xhr    | VM14822:1     | 0 B   | 2.01 s |  |
| personnes           | (failed) |        | Other         | 0 B   | 2.00 s |  |
| common.js           | 200      | script | bootstrap:149 | 210 B | 9 ms   |  |
| cv-cv-module.js     | 200      | script | bootstrap:149 | 211 B | 9 ms   |  |
| todo-todo-module.js | 200      | script | bootstrap:149 | 211 B | 8 ms   |  |

# Preloading Lazy Loading

## Créer votre propre stratégie

---

- Avec **PreloadAllModules**, tous les modules sont **préchargés**, ce qui peut en fait créer un **goulot d'étranglement** si l'application a un **grand nombre de modules à charger**.
- Une meilleure stratégie serait de **charger sélectivement les modules requis au démarrage**. Par exemple, module d'authentification, module principal, module partagé, etc.
- Pour **précharger sélectivement** un module, nous devons utiliser une **stratégie de préchargement personnalisée**.
- Créez d'abord une **classe** qui implémente l'interface **PreloadingStrategy**. La classe doit implémenter la méthode **preload()**.
- C'est cette méthode qui détermine s'il **faut précharger le module ou non**.

# Preloading Lazy Loading

## Créer votre propre stratégie

- La signature de la fonction **preload** prend en paramètre un **objet Route** représentant la route ciblée et en deuxième paramètre une fonction **load** qui retourne un Observable.
- Si vous retournez la méthode load, le module sera **préchargé**.
- Si vous **ne voulez pas le précharger** retourner un **Observable de null**.

```
@Injectable({providedIn: 'root'})
export class CustomPreloadingStrategy implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data["preload"]) {
      return load();
    }
    else {
      return of(null);
    }
  }
}
```

---

aymen.sellaouti@gmail.com