

# Formation : Développement d'applications avec Angular

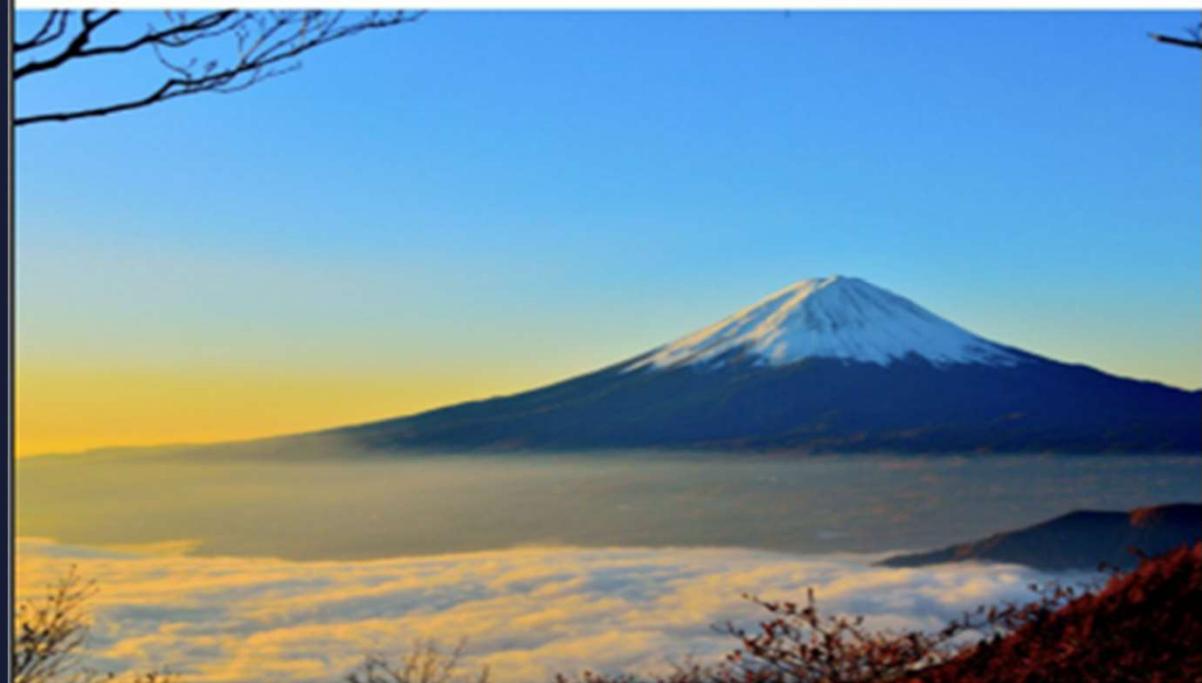
Aymen sellaouti

2024



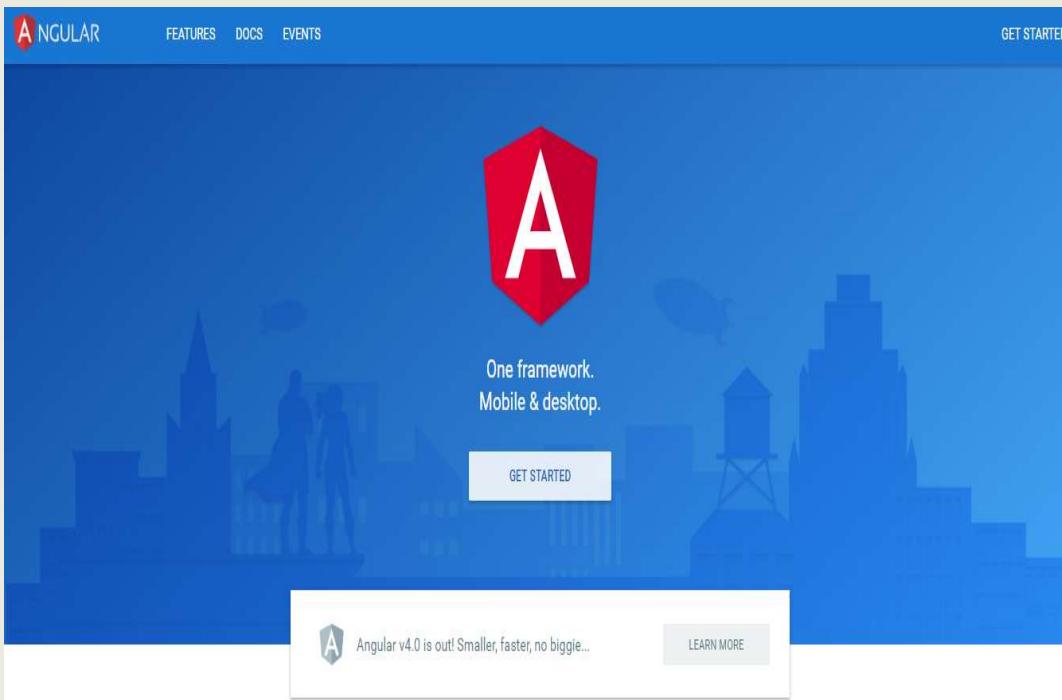
## Learn Angular

Gravissez tous les concepts jusqu'au sommet



Maîtriser les concepts du Framework Angular  
pour développer des applications robustes

# Références



# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis.** Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

# Pré-requis

- ▶ HTML 5 / CSS 3 / Bootstrap



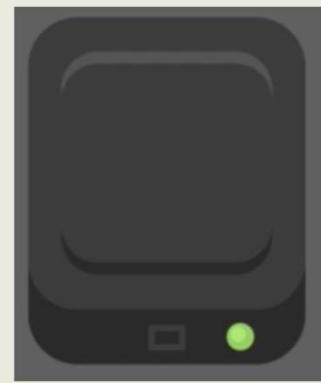
- ▶ JavaScript



- ▶ Programmation Orientée Objet

**POO**

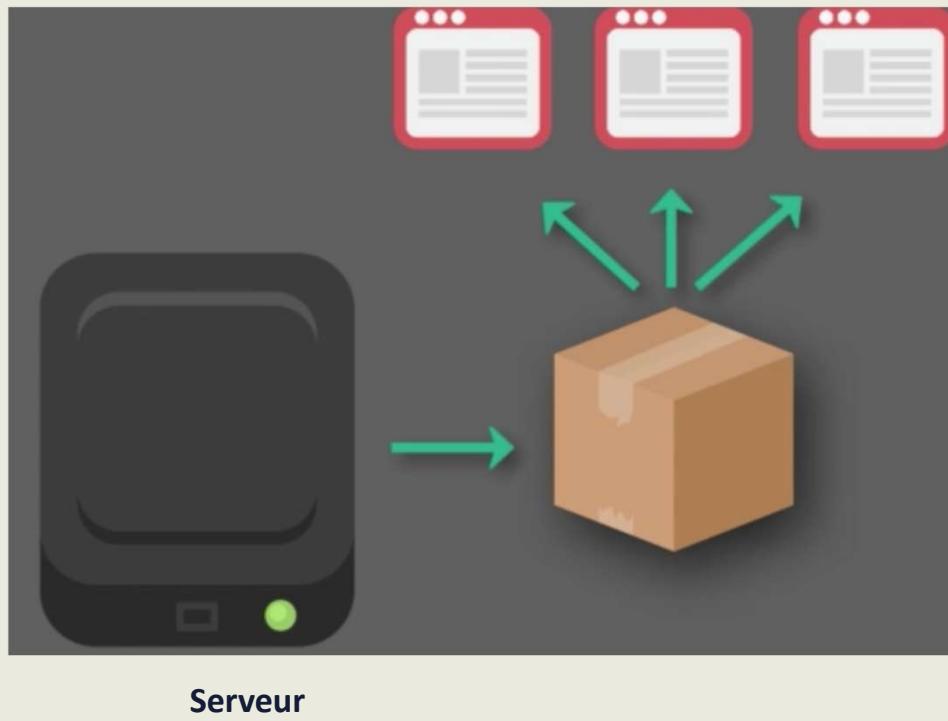
# Site web



Serveur



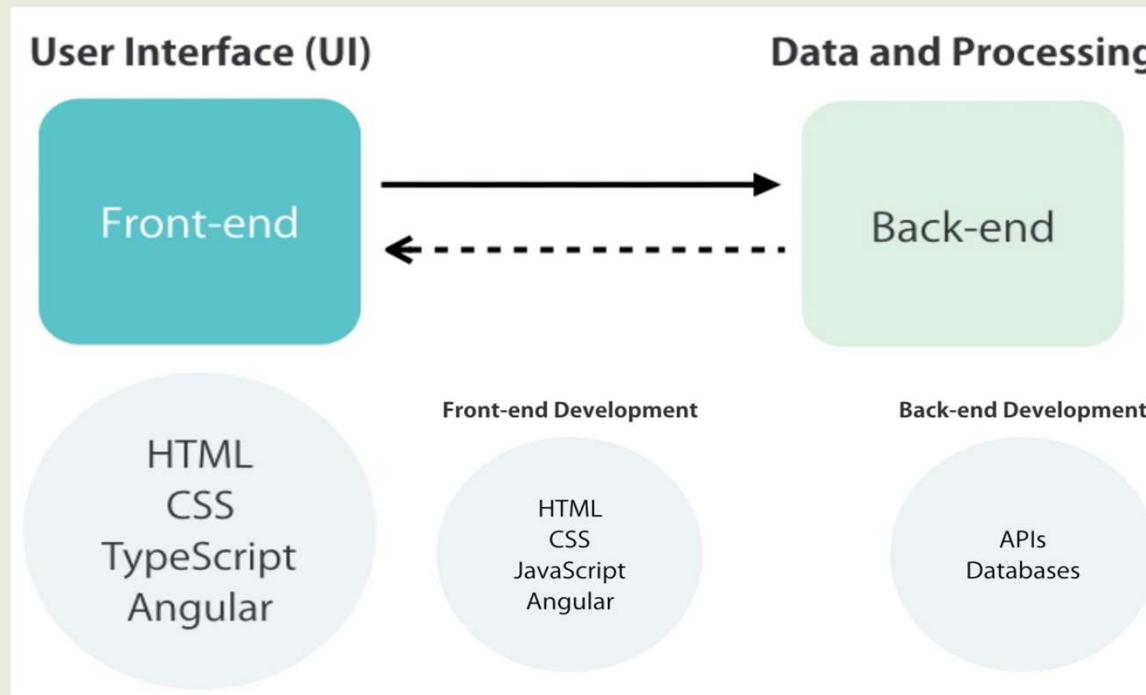
# Application Web



Intervention du JS

Application web en SPA  
*(Single Page Application)*

# Front vs Back



# C'est quoi NodeJs?



# C'est quoi NodeJs?



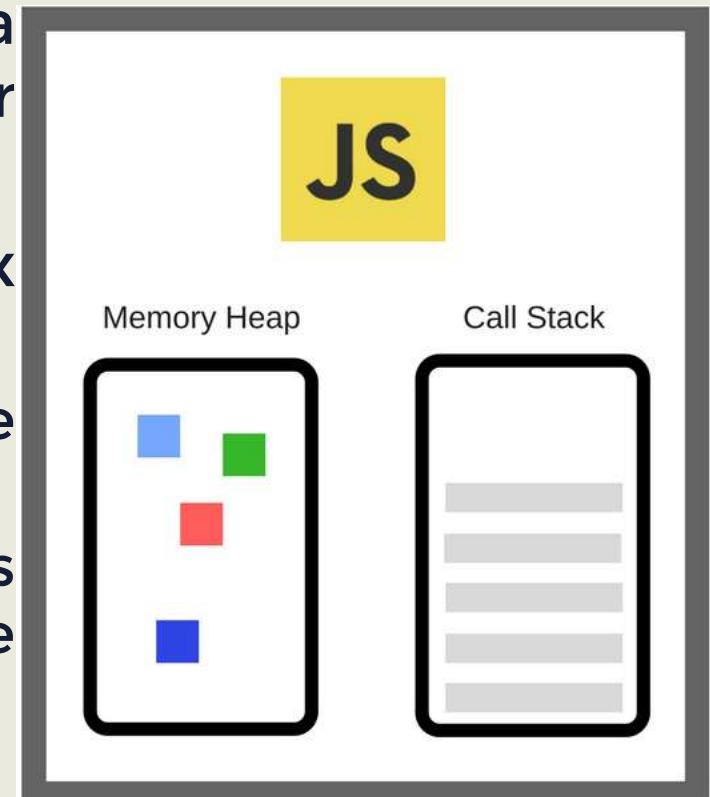
Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

- Environnement d'exécution JS
- Utilise V8 le moteur JS de Google.

# Moteur Javascript : Javascript Engine



- Tout code JavaScript que vous écrivez a besoin d'un moteur Javascript pour l'exécuter.
- Le moteur JavaScript possède deux composants principaux:
  - **Memory Heap** : sert à allouer la mémoire utilisée dans le programme.
  - **Call Stack** (pile d'exécution) : contient les données des fonctions exécutées par le programme.



# Moteur Javascript : Javascript Engine



# V8 Javascript Engine



- V8 est un moteur JavaScript développé par Google et utilisé par chrome pour exécuter du code JavaScript.
- Il est *Open Source*
- Ecrit en C++



# Qu'est-ce qu'un environnement d'exécution JS



- Un **environnement d'exécution JS** est l'endroit où votre code JS va être exécuté. C'est **là où vivra votre JavaScript Engine**.
- Il va, entre autre,, **déterminer quelles variables globales** sont accessibles pour vous (**window** pour le runtime envirement de votre browser)
- **Ajax, DOM et d'autres API's ne font pas parti de JavaScript.** C'est l'environnement d'exécution Js (Ici fourni par votre Browser) qui les rend disponible.

# Est-ce que JavaScript est Synchrone ou Asynchrone ?



- Js est **SYNCHRONE**.
- Les fonctions asynchrones telles que **setTimout** ne font pas partie de JS.
- Elle font partie des API **de votre runtime envirement**.
- Donc JS peut **agir d'une façon ASYNCHRONE mais ceci n'est pas innée**, ce n'est pas dans son Core mais plutôt du au API offert par le runtime.
- Dans NodeJs c'est **libuv qui permet cet aspet Asynchrone**.

# Asynchrone

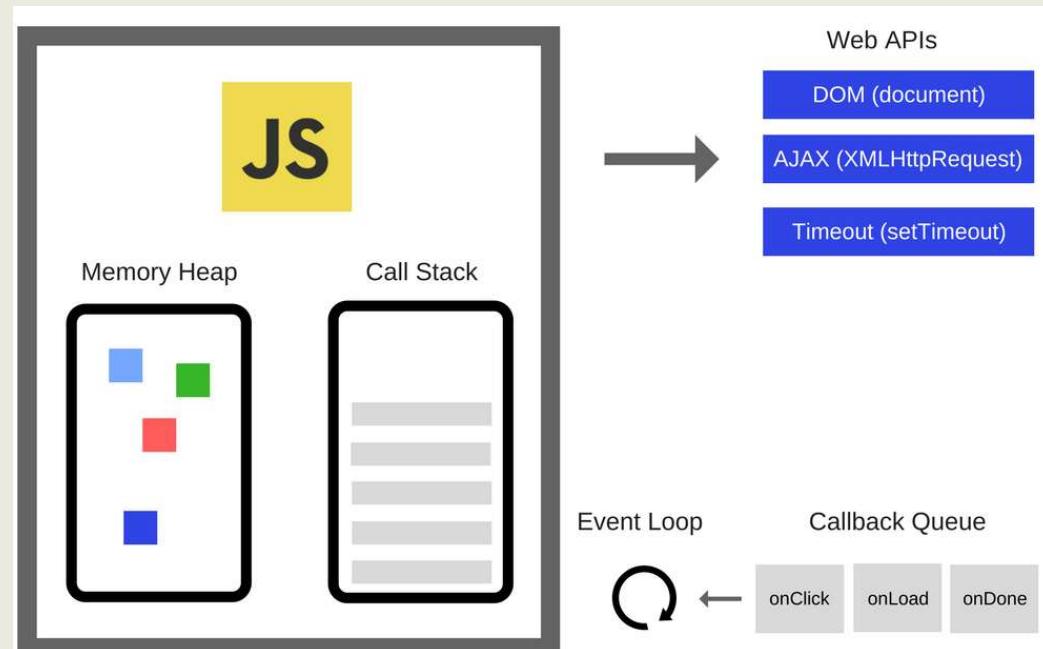


- Un code asynchrone est un code non bloquant
- Une fonction non bloquante est une fonction qui s'exécute en parallèle avec le reste du code.
  
- Exemple : setTimeout, makeRequest, readFile.

# Qu'est-ce qu'un environnement d'exécution JS



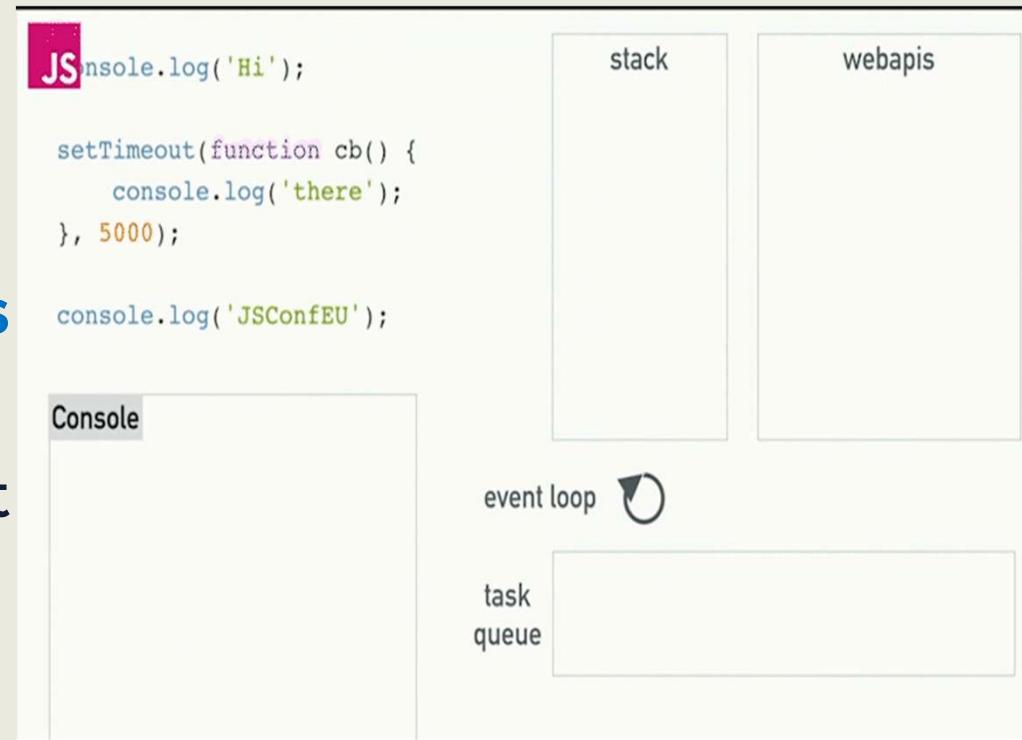
- Si on gardait uniquement l'aspect Synchrone de JS, ça sera un problème énorme avec le browser.
- La solution proposée par les API est l'ajout de la composante synchrone via l'Event loop et le callback queue.
- Ceci va permettre d'effectuer des traitements sans bloquer le stack du moteur JS.



# Comment ça marche dans le browser ???



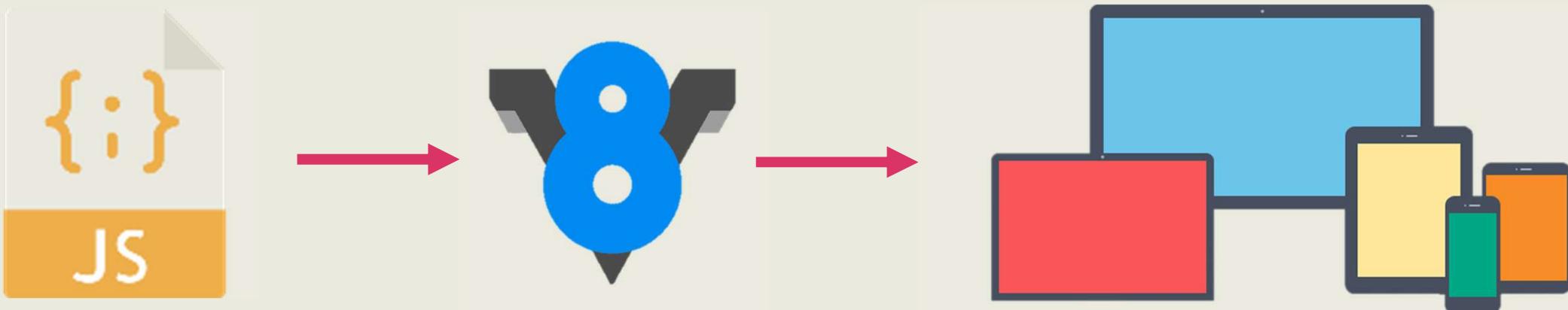
- A part le moteur JS, votre environnement possède des API's qui permettent à votre Moteur JS d'exécuter certaines fonctionnalités comme le timeout, ou fetch.
- A ces API's est associé un t loop et des callback queue.
- Pour le browser vous avez l'API DOM.



# Comment est né Node ?



- Au départ on ne pouvait exécuter Js que dans un Browser. C'est lui qui fournissait son environnement d'exécution.



- Une question a été donc posée. Pouvons nous exécuter du Js en dehors du browser ? Pouvons nous avoir du Js côté Serveur ?

# Comment est né Node ?



2009



Ryan Dahl  
Node.js Creator  
Software Engineer @ Joyent



# Comment est né Node ?



*window  
document  
history  
location*



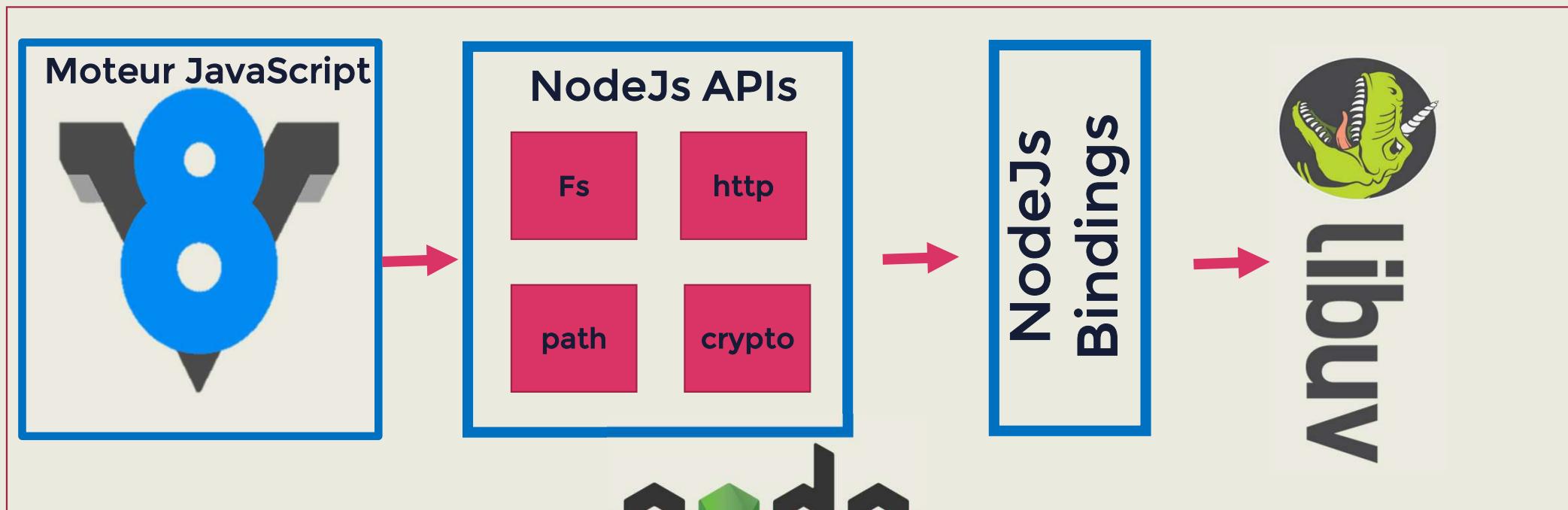
*global  
process  
module  
\_\_filename*

# **NodeJs un environnement d'exécution**



- Afin de s'exécuter, Javascript a besoin d'un environnement d'exécution.
- C'est comme un conteneur qui contient tout ce qu'il vous faut pour exécuter du code Js.
- Le noyau de l'environnement d'exécution est le moteur Js.
- Il y a aussi les APIs Node
- Libuv la bibliothèque qui permet de gérer les I/O Asynchrone
- Les nodes Js Binding

# NodeJs un environnement d'exécution



# Node Package Manager (NPM)

- Lorsque vous installez NodeJs, vous récupérer son Gestionnaire de Package npm
- NPM est un gestionnaire de module pour NodeJs.
- Il permet de gérer les dépendances des modules automatiquement.
- Il fonctionne en ligne de commande.
- Il permet de rechercher, installer et gérer vos dépendances.

# Node Package Manager (NPM)

- Afin d'installer une dépendance, vous devez lancer la commande **npm install nomPackage**

**npm install uuid**

- npm permet d'installer des dépendances dans un contexte :
  - locale (au niveau de votre projet uniquement), c'est le comportement par défaut, ces modules sont installés dans un dossier `node_modules`
  - global, on ajoutant l'option `-g` :

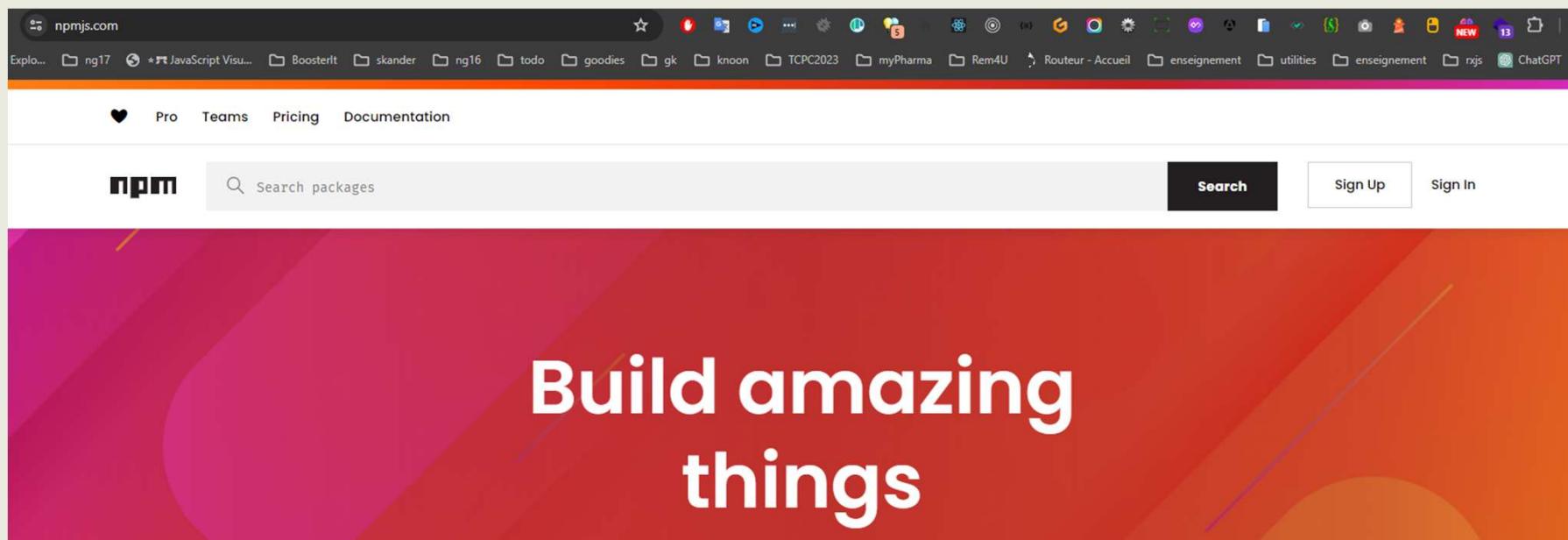
**npm install uuid -g,**

- ces modules sont installés dans **un seul dossier**. Afin de récupérer le **path de ce dossier** tapez :

**npm root -g**

# Node Package Manager (NPM)

➤ Le site <https://www.npmjs.com/> vous permet de visualiser les packages partagés par la communauté :



# Node Package Manager (NPM)

- Lister les modules
  - npm ls
  - npm ls -g
- Rechercher un module
  - npm search <Module name>
- Désinstaller
  - npm uninstall <Module name>
  - npm uninstall -g <Module name>

# **Node Package Manager (NPM)**

- Pour lister les paquets qui ne sont pas à jour
  - `npm outdated`
- Pour mettre à jour les modules
  - `npm update`

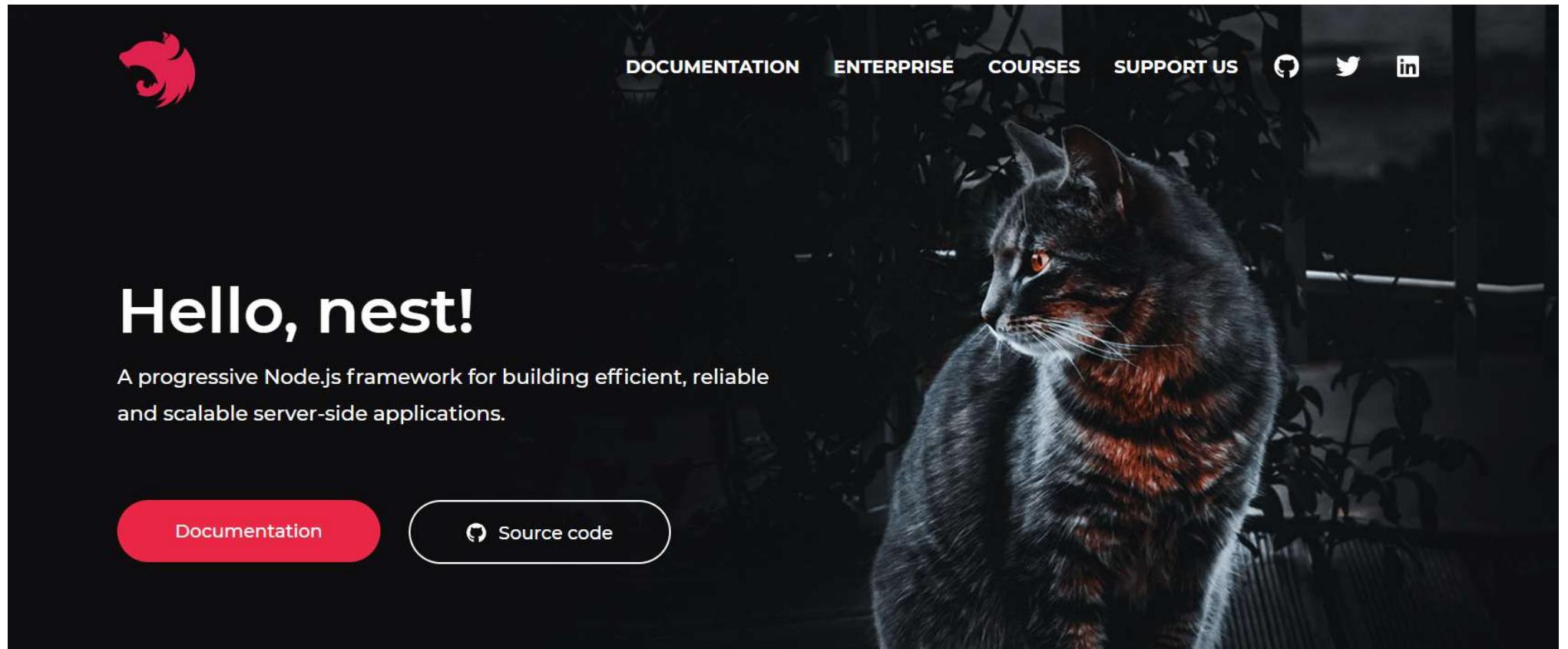
# EXPRESS NestJs

Aymen sellaouti



**<https://github.com/aymensellaouti/nestGk131025>**

# Références



The screenshot shows the official Nest.js website. At the top left is a red stylized logo resembling a flame or a bird. To its right are navigation links: DOCUMENTATION, ENTERPRISE, COURSES, and SUPPORT US. Below these are social media icons for GitHub, Twitter, and LinkedIn. The main visual is a close-up photograph of a dark, fluffy cat with bright orange eyes, looking slightly to the left. On the left side of the page, the text "Hello, nest!" is displayed in large white letters, followed by a subtitle: "A progressive Node.js framework for building efficient, reliable and scalable server-side applications." Below this are two buttons: a red one labeled "Documentation" and a white one labeled "Source code".

# Plan du Cours

- 1. 1- Installation du framework.**
- 2. 2- Les modules, les contrôleurs et les services**

# Introduction



- Nest (NestJs) est un **framework NodeJs**.
- Utilise **TypeScript**
- NestJs est basé sur **ExpressJs (par défaut)** ou si vous le souhaitez sur Fastify.
- Nest ajoute une **couche d'abstraction** sur ces deux Framework et il permet aussi d'utiliser leurs modules.

# Introduction Pourquoi NestJs



- C'est un FRAMEWORK
- Une communauté en croissance continue et très active
- La documentation
- Possibilités de construire des Rest API's, des applications

MVC, des microservices, d'exposer du GraphQL, des Web Sockets ou des CLI's et des CRON jobs.

- Designs patterns implémentés et prêts à l'emploi (IOC)
- Un cli puissant qui facilite le développement
- Plusieurs modules prêts à l'emploi
- Intégration facile de plusieurs technologies

Install

```
> npm i @nestjs/core
```

Repository

nestjs.com/nest

Homepage

nestjs.com

Fund this package

Weekly Downloads

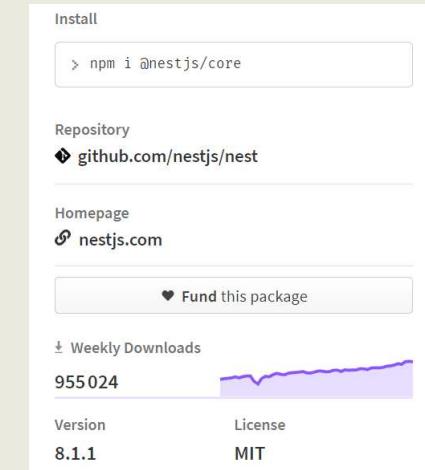
955 024

Version

8.1.1

License

MIT



nest Public

A progressive Node.js framework for building efficient, scalable, and enterprise-grade server-side applications on top of TypeScript & JavaScript (ES6, ES7, ES8) 

nodejs javascript node microservices framework typescript javascript-framework

TypeScript MIT 4,277 41,398 45 (3 issues need help) 32 Updated 28 minutes ago



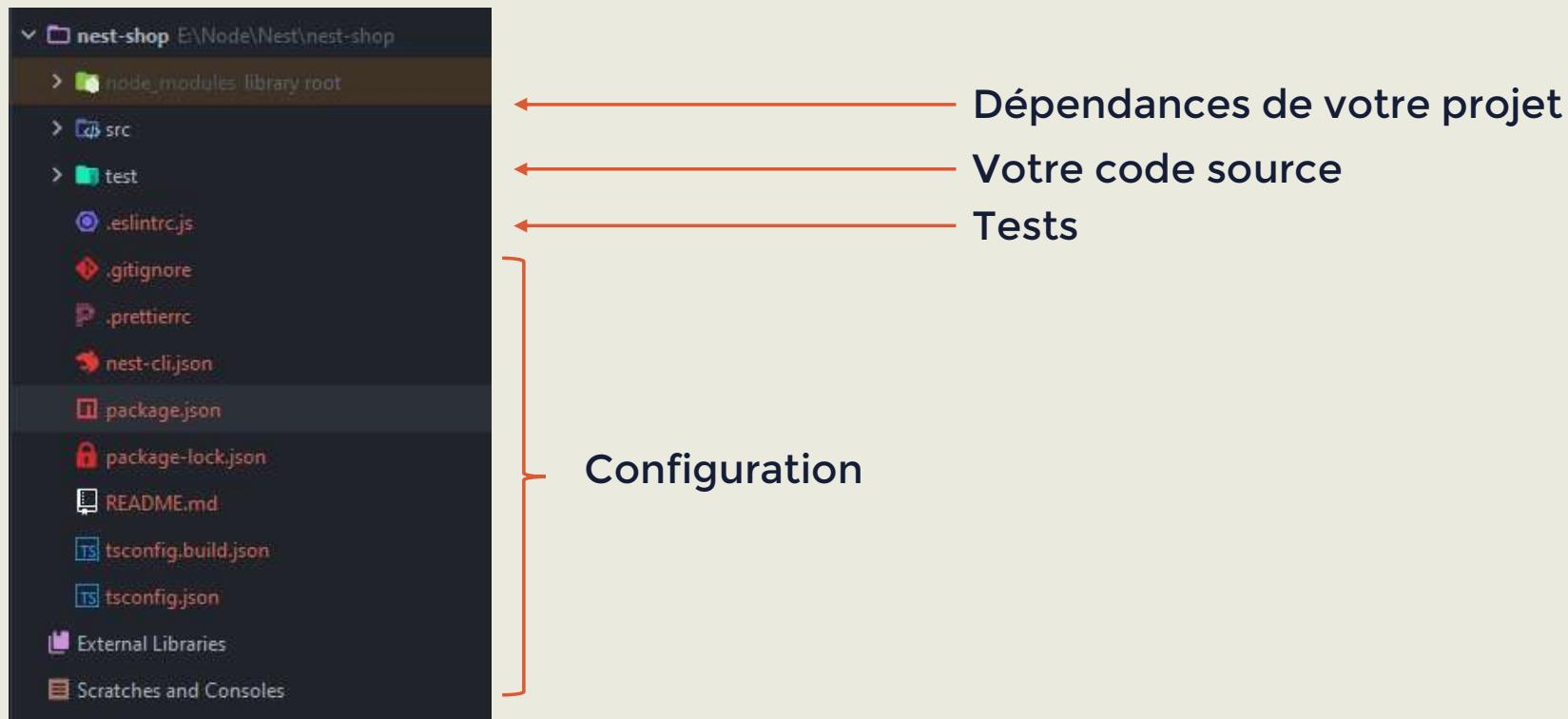
# Installation

- Afin d'installer Nest vous devez avoir NodeJs ( $\geq 10.13.0$ ).
- Vous pouvez ensuite cloner un projet prêt à l'emploi ou utiliser le Nest Cli
- Pour installer le Nest Cli vous lancez la commande :  
`npm i -g @nestjs/cli`
- Une fois le nest CLI installé, vous avez accès à la commande **nest** qui vous permettra de créer votre projet et de scafoldier du code.

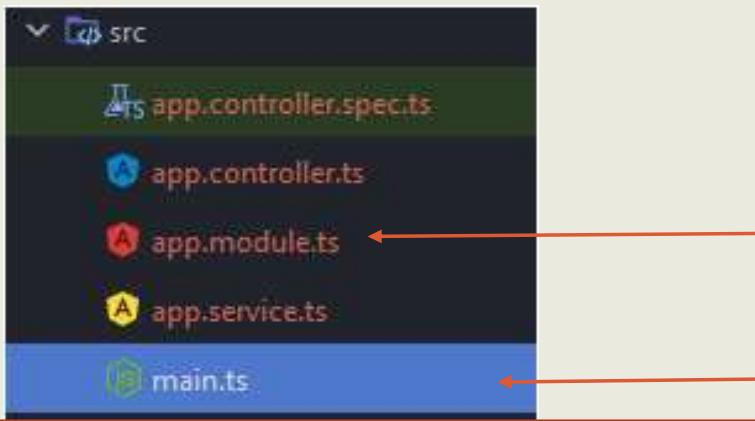
# Installation

- Pour créer un nouveau projet, lancer la commande **nest new NomProjet**.
- Une fois fini, pour lancer votre projet taper la commande
  - **npm run start:dev**
  - Ou
  - **nest start -watch**
- Votre application est maintenant accessible sur le port 3000 via l'url **localhost:3000**

# Structure d'un projet Nest



# Structure d'un projet Nest

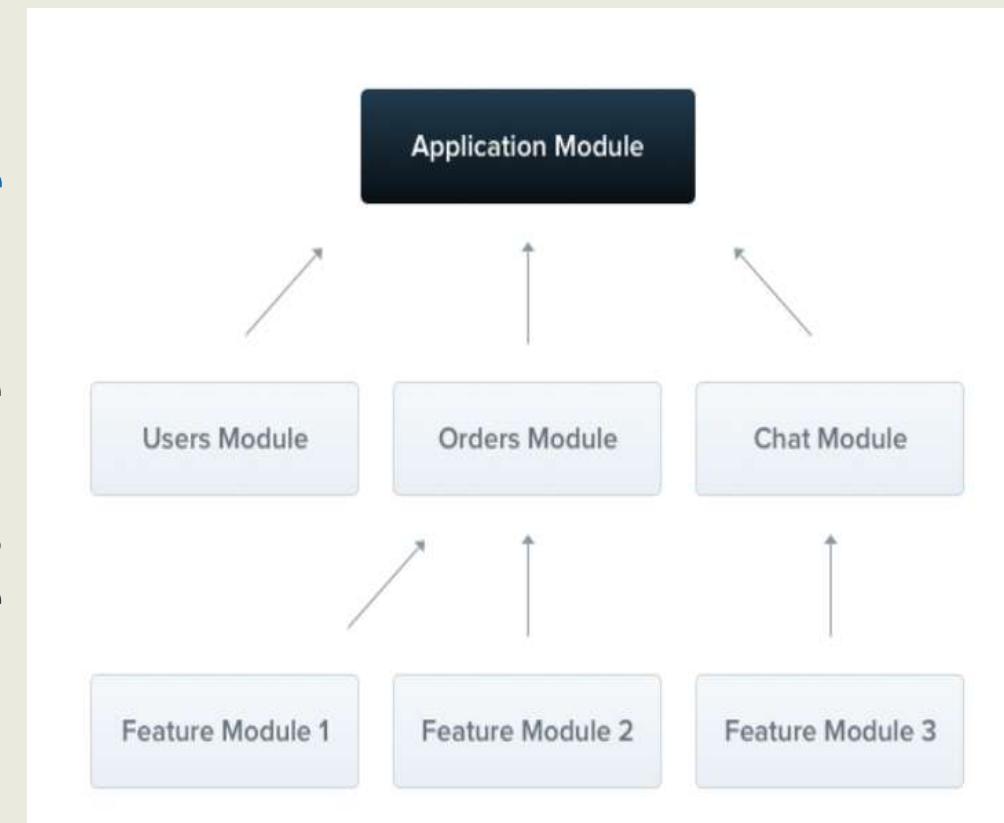


Le module principal de votre application

```
1 import { NestFactory } from '@nestjs/core';
2 import { AppModule } from './app.module';
3
4 async function bootstrap() {
5   const app = await NestFactory.create(AppModule);
6   await app.listen(3000);
7 }
8 bootstrap();
```

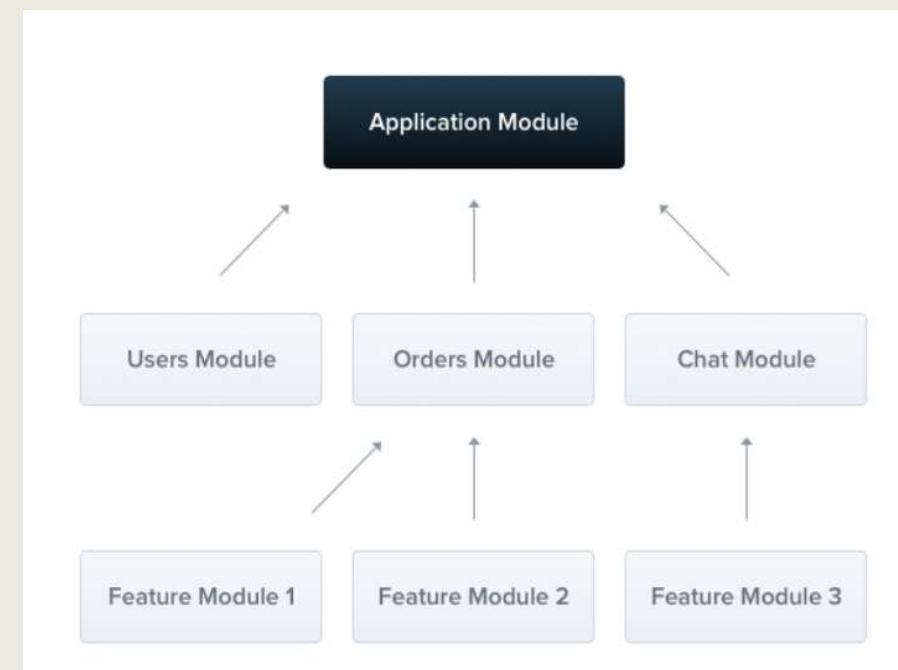
# Les modules

- NestJs a choisi de décomposer les projets en **Modules**.
- Un module est une partie isolée **de votre application**.
- Elle **encapsule** plusieurs **fonctionnalités liées**. Par exemple le module des utilisateurs qui se charge de gérer les users, les rôles etc..
- On peut dire qu'un module inclura les fonctionnalités nécessaires **pour un métier** de votre application.



# Les modules

- Un module est une **classe annotée** avec un décorateur `@Module()`. Le `@Module()` fournit des métadonnées que Nest utilise pour organiser la structure de l'application.
- Chaque application possède **au moins un module** c'est le module racine.
- Le module racine est le point de départ utilisé par Nest pour créer le graphe de l'application.
- Nest **recommande fortement la décomposition en Modules.**



# Les modules

➤ Les paramètres de l'annotation `@Module()` sont :

providers	les providers qui seront instanciés par l'injecteur Nest et qui peuvent être partagés au moins sur ce module.
controllers	l'ensemble des contrôleurs définis dans ce module
imports	la liste des modules importés qui exportent les providers requis dans ce module.
exports	Les providers fournis par ce module et qui peuvent être utilisés par d'autres modules.

# Les modules

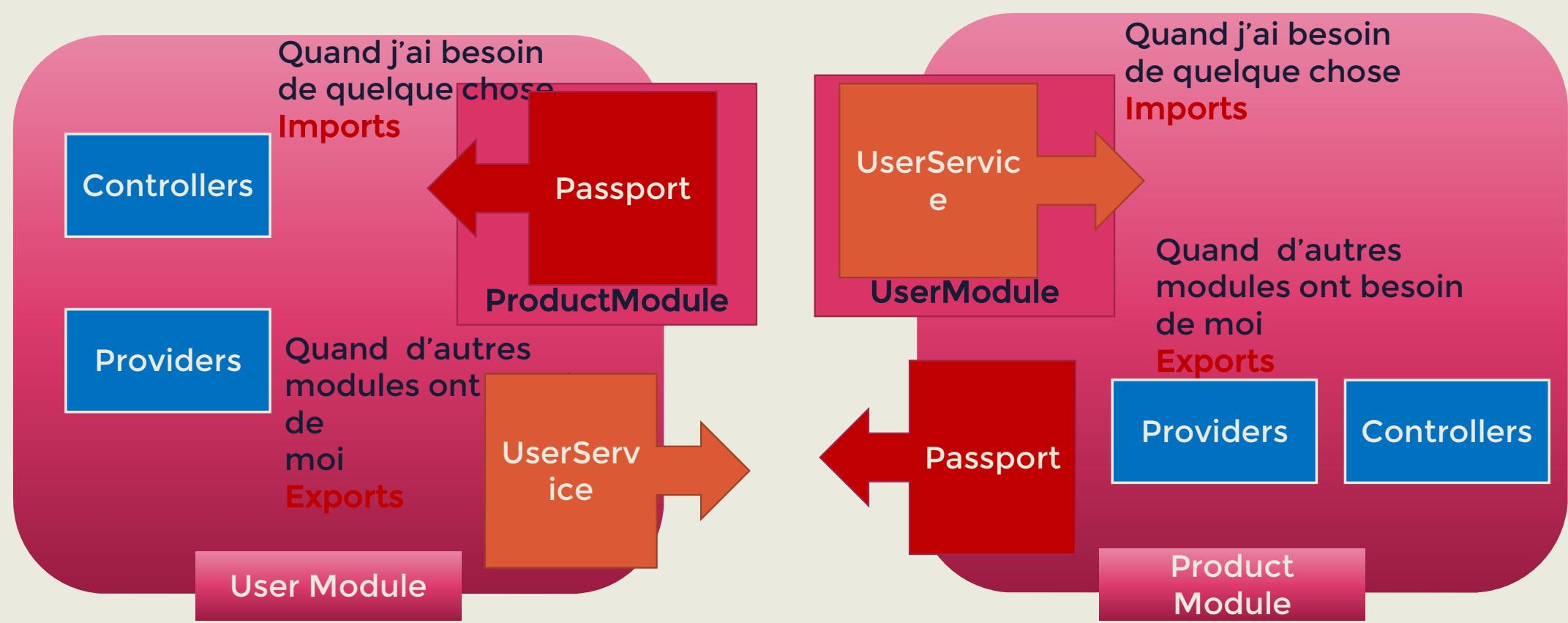
```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  exports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

# Les modules

- Par défaut, les modules encapsulent leurs providers
- Ceci implique que les providers sont uniquement accessibles à l'intérieur de ce module par défaut.
- Nous pouvons conclure donc que dans un module, nous ne pouvons utiliser (réellement injecter) que ses providers ou ceux exportés par les modules qu'il a importé.
- NestJs décrit les providers exportés par un module comme son Interface ou son API Publique.

# Les modules



# Les modules

➤ Vous pouvez créer un module Nest via le Cli en utilisant la commande

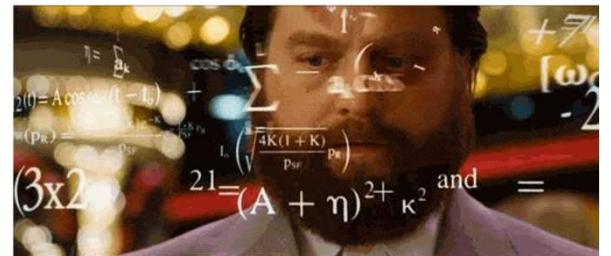
`nest generate module nomDuModule`

Ou via le raccourci :

`nest g mo nomDuModule`

# Exercice

- Créer un module Premier.
- Intégrer le avec votre App.module.ts



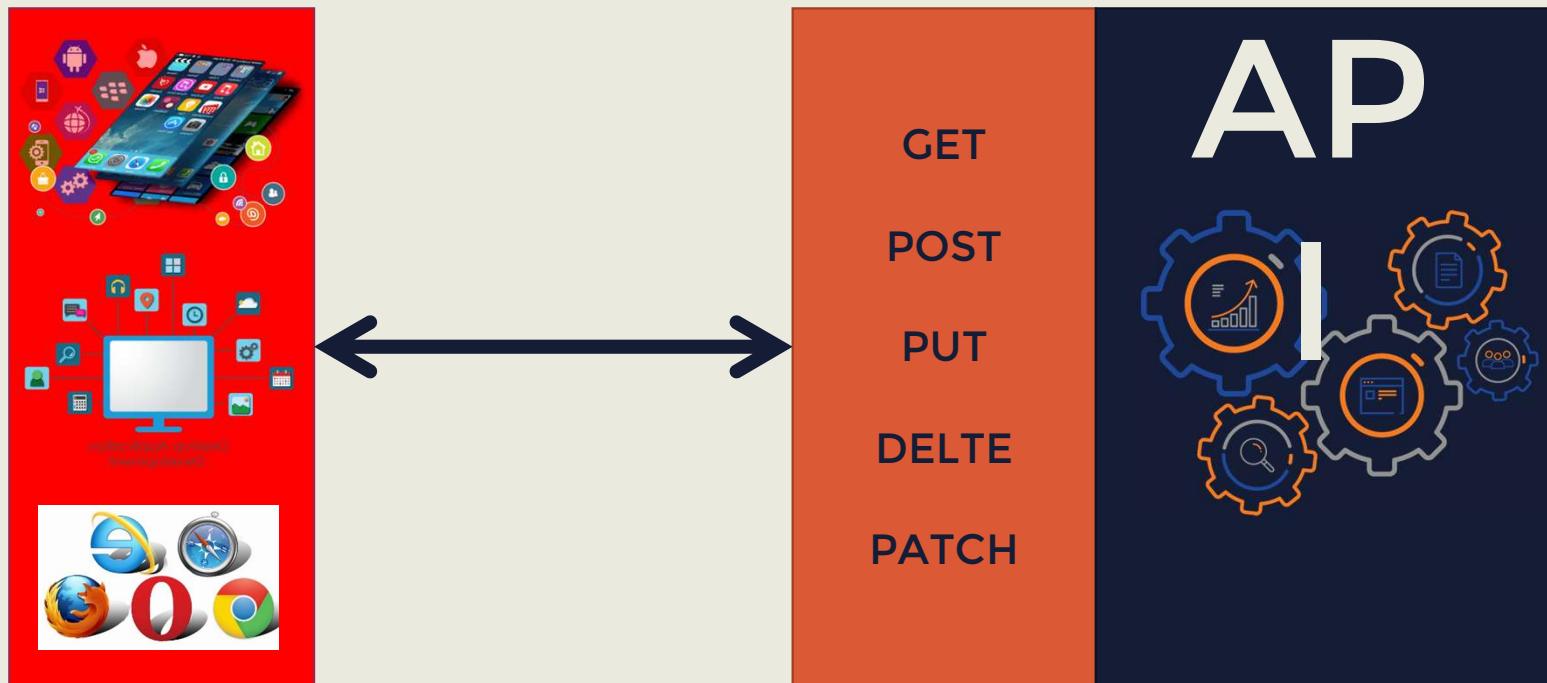
# Les contrôleurs

Rest

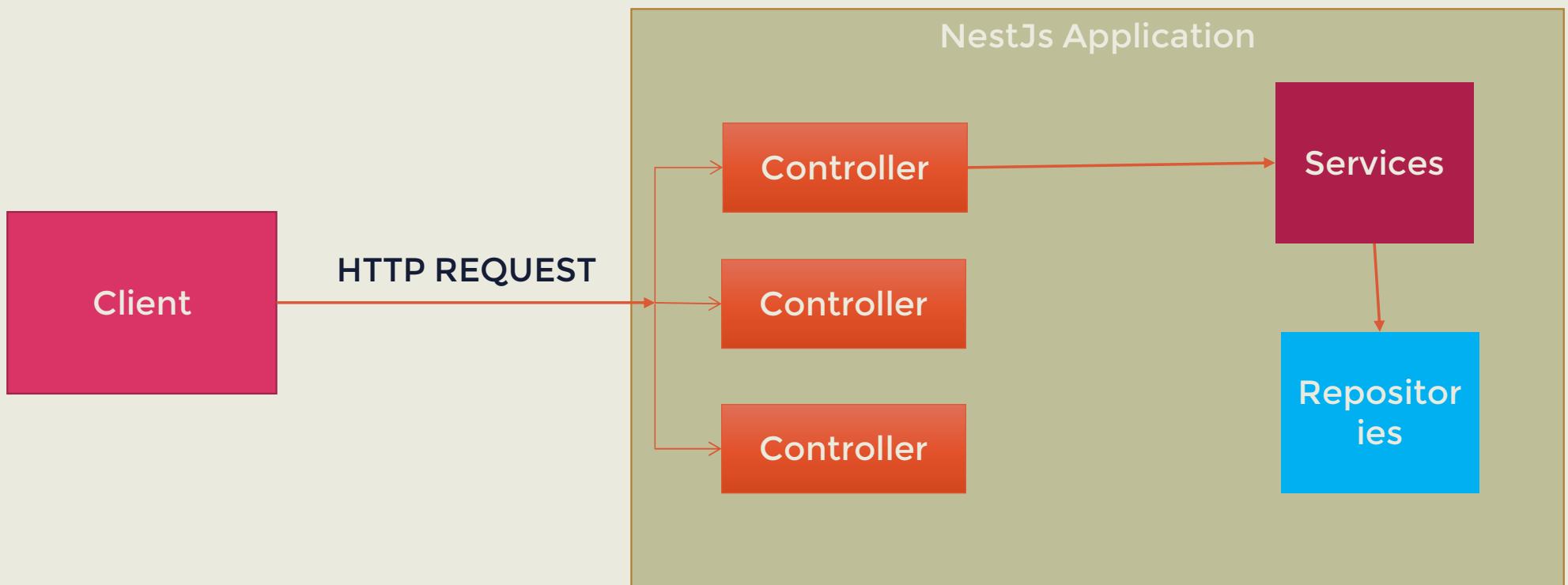
- REST (REpresentational State Transfer) est un style architectural, un design Pattern pour les API.
- Une API RestFull est une ([API](#)) qui utilise le protocole HTTP pour GET, PUT, POST et DELETE les données.



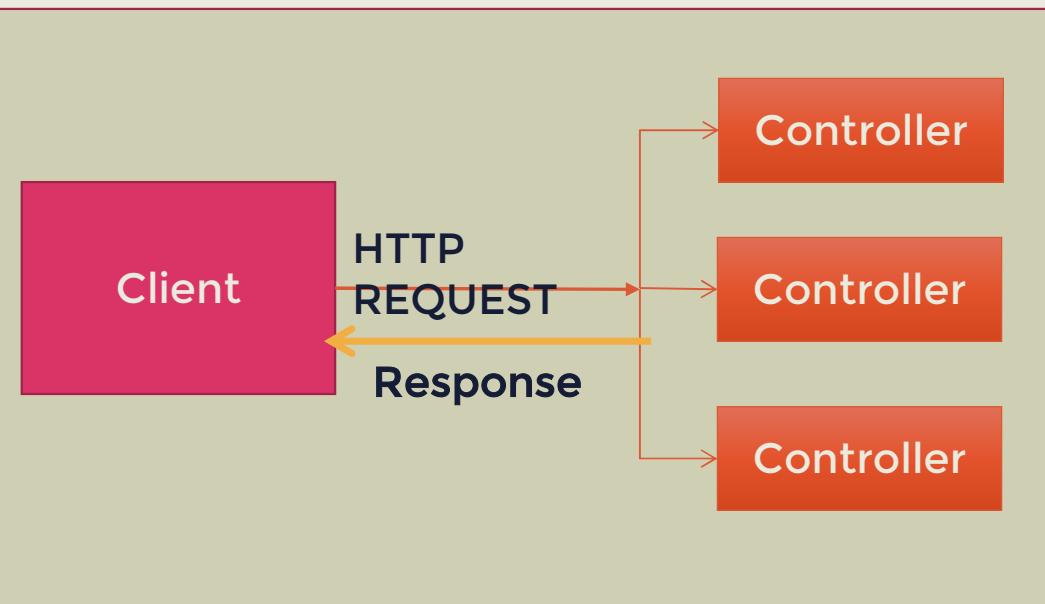
# REST API



# Architecture



# Les contrôleurs



- Le rôle d'un Controller est de réceptionner les **requêtes** HTTP entrantes, de préparer une **réponse** et de la retourner.
- Le mécanisme de **routage** contrôle quel contrôleur reçoit quelle **requête**.
- Chaque contrôleur peut gérer **plusieurs routes**.
- Chaque route est responsable d'une action.
- Afin de créer un contrôleur, nous utilisons des classes et des **décorateurs**.
- Les décorateurs associent les classes aux métadonnées requises et permettent à Nest de **créer une carte de routage** permettant de lier les demandes aux contrôleurs correspondants.

# Les contrôleurs

- Pour identifier une classe comme étant un contrôleur, vous devez l'annoter (la décorer) avec **@Controller** et l'ajouter dans son module sous la clé **controllers**

```
import { Controller, Get } from '@nestjs/common';

@Controller()
export class AppController {

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

```
@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

# Les contrôleurs

- Pour résumer un contrôleur est une classe avec l'annotation `@Controller` contenant un groupement de méthodes permettant de gérer les requêtes http envoyées par vos clients
- Pour créer le contrôleur, vous pouvez le faire manuellement en créant la classe, en la décorant avec `@Controller` et en l'ajoutant dans le module associé, ou via la commande
  - `nest generate controller controllerName`
  - `nest g co controllerName`

# Routing

- Une route va identifier l'uri associé à une action (un handler).
- Nest propose des **décorateurs** (annotations) permettant de définir la route associée à une action de votre contrôleur.
- Pour chaque méthode HTTP vous avez un décorateur associé.
- Le décorateur prend en paramètre l'uri à gérer. Ceci nous permet d'avoir une combinaison uri + méthode identifiant exactement la requête HTTP à gérer par votre action.
- Les méthodes les plus utilisées sont :  
`@Get(), @POST(), @Delete(), @Put(), @Patch()`

# Routing

- Dans cet exemple si vous faites une requête Get sur la route localhost:3000/test, la méthode getHello sera exécutée.
- Elle vous renverra la réponse 'HELLO NEST'

```
@Get('test')
getHello(): string {
    return 'HELLO NEST';
}
```

# Routing

## Préfixer les routes d'un contrôleur

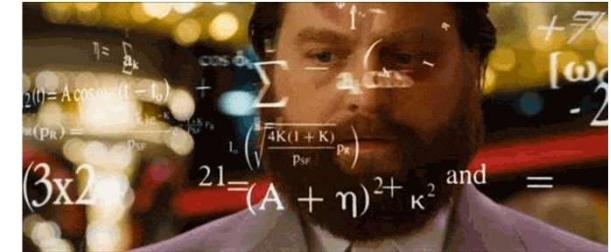
- Vous pouvez préfixer les routes d'un contrôleur en indiquant le préfixe comme premier paramètre de l'annotation `@Controller`.

```
// Ici toutes les routes de ce contrôleur commenceront par /tasks/
@Controller('tasks')
export class TasksController {
  @Get('all')
  getTasks(): string {
    // TODO
  }
}
```

- Dans cet exemple, toutes les routes gérées par ce contrôleur seront préfixées par le segment '`tasks`'. Donc pour accéder à la méthode `getTasks`, il faut l'uri `/tasks/all`

# Exercice

- Créer un module et un contrôleur todo et ajouter le au module todo.
- Créer une liste de todos. Chaque todo est caractérisé par son **id**, son **name**, sa **description**, sa **date de création** et son **statut**.
- Le statut doit être l'un de ces trois : En attente, En cours, Finalisé. Utiliser un **enum** pour définir ce type.
- Ajouter une méthode Get qui retourne la liste des todos.
- Tester la.



```
import { Controller, Delete, Get, Patch, Post, Put } from '@nestjs/common';
@Controller('todo')
export class TodoController {
  private todos = [];
  @Get()
  getTodos() {
    // Todo 1 : init the todos array
    // Todo 2 : Get the todo list
  }
}
```

```
export enum TodoStatusEnum {
  'actif' = "En cours",
  'waiting' = "En attente",
  'done' = "Finalisé"
}
```

# L'objet Request

➤ Si vous voulez récupérer l'objet Request (offerte par le framework que vous utilisez et qui est express par défaut), Nest vous offre une annotation vous permettant de le récupérer. C'est l'annotation `@Req.`

```
import { Controller, Get, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller()
export class AppController {

  @Get()
  getHello(@Req() req: Request): string {
    console.log(req);
    return 'HELLO NEST';
  }
}
```

# L'objet Request

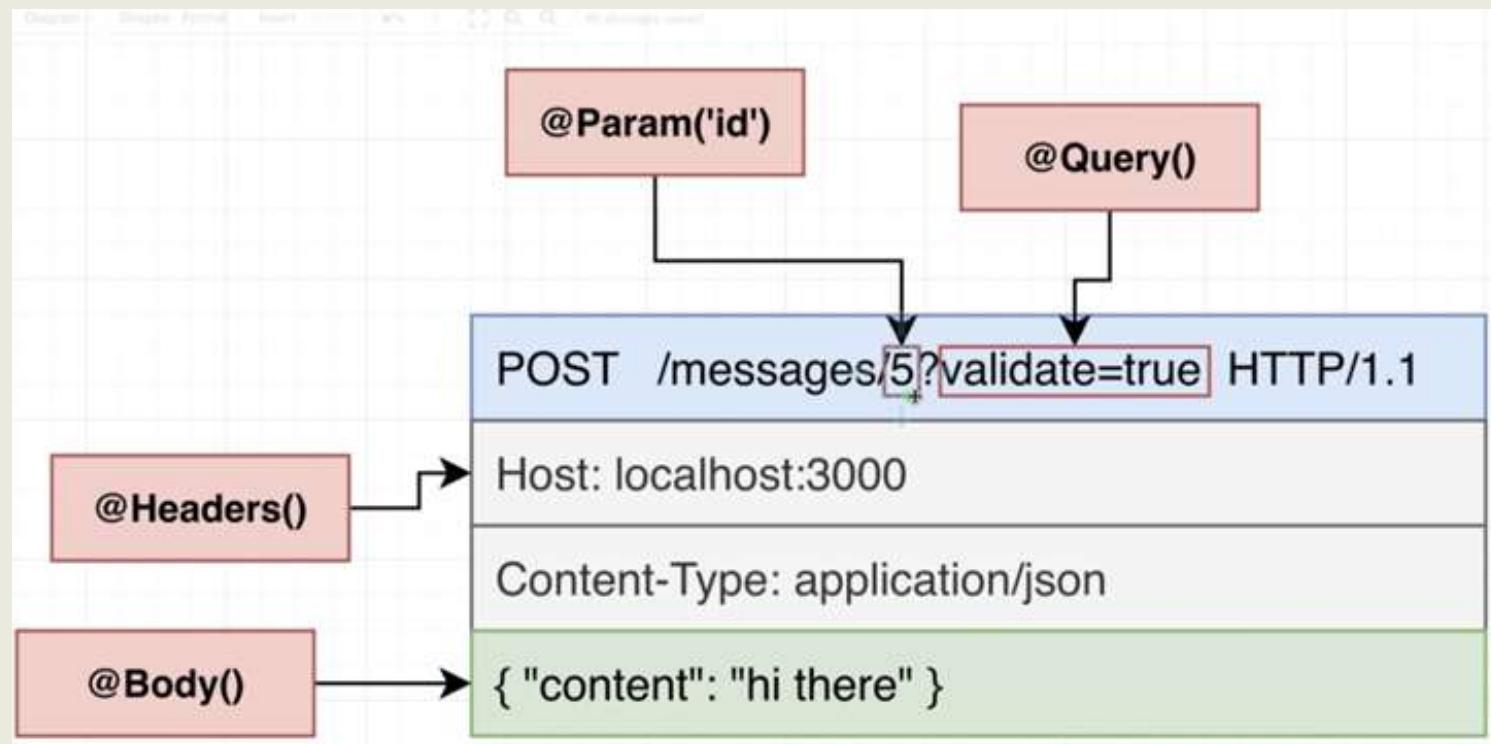
## Récupérer les différents éléments de la requête

➤ Au lieu de passer par l'objet **request** pour récupérer ce que vous voulez, il suffit d'utiliser les décorateurs offerts par Nest.

@Request()	req	Récupérer l'objet Request
@Param(key?: string)	req.params / req.params[key]	Récupérer les paramètres du Body de votre requête
@Body(key?: string)	req.body / req.body[key]	Récupérer le body de votre requête
@Query(key?: string)	req.query / req.query[key]	Récupérer les queryParams envoyé en GET
@Headers(name?: string)	req.headers / req.headers[name]	Récupérer les Headers
@Ip()	req.ip	Contient l'adresse IP de la requête

# Routing

## Récupérer les différents éléments de la requête



# Contrôleur

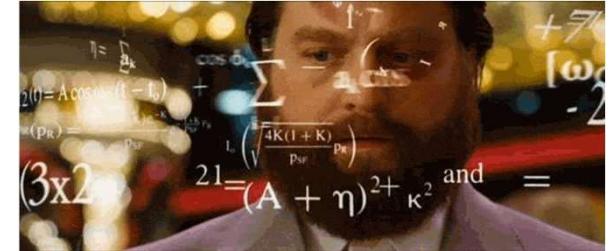
Request Body : Récupérer le body d'une requête POST

- Pour récupérer le body d'une requête POST vous devez utiliser le décorateur `@Body()` dans les paramètres de votre action.

```
@Post('/test')
testPost(
    @Body() body
) {
    console.log(body);
}
```

- Pour récupérer un champ particulier du body, ajouter comme paramètre de votre annotation la clé de champ : `@Body('name')` vous permettra de récupérer le champ **name** de votre body

# Exercice



- Dans votre contrôleur todo, créer une méthode qui permet d'ajouter un Todo.
- La génération de l'id doit être automatique. Penser à utiliser un générateur d'id unique comme le composant **uuid**.
- La date de création doit aussi être automatique.
- Le statut par défaut est « En attente ».

```
export enum  
TodoStatusEnum {  
    'actif' = "En cours",  
    'waiting' = "En attente",  
    'done' = "Finalisé"  
}
```

# Routing

## Définir des paramètres d'une route

- Lorsque vous créez une route et que vous voulez qu'un ou plusieurs de ces fragments soient dynamiques, préfixez les par

..

```
@Get('/post/:year/:id')
getPost(): string {
  return 'Post';
}
```

- En ajoutant un ? devant le nom du paramètre, vous informez le routeur que ce paramètre est optionnel.

```
@Get('/post/:year/:id?')
getPost(): string {
  return 'Post';
}
```

# Routing

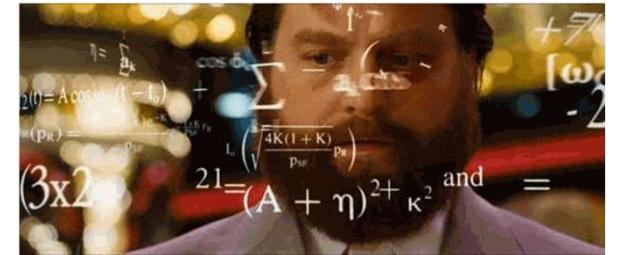
## Récupérer les paramètres d'une route

- Pour récupérer ces paramètres au niveau de votre action, utilisez le décorateur `@Param()` au niveau des paramètres de votre méthode.
- Si vous ne passez aucun paramètre au décorateur `@Param`, il vous retourne un tableau contenant tous les paramètres. Accédez ensuite via cet objet à la propriété que vous voulez avec le nom du paramètre.
- Si vous voulez accéder directement au paramètre, ajoutez son nom comme paramètre de l'`@annotation` `@Param`

```
@Get('/post/:year/:id')
getPost(@Param() mesRoutesParams): string {
  return 'Post créer en :' + mesRoutesParams.year;
}
```

```
@Get('/post/:year/:id')
getPost(
  @Param('id') id,
  @Param('year') year
): string {
  return `Post d'id ${id} créé en ${year}`;
}
```

# Exercice



Dans votre contrôleur todo créer les méthodes

- permettant de récupérer un todo via son id.
- permettant de supprimer un todo via son id.
- permettant de modifier un todo.

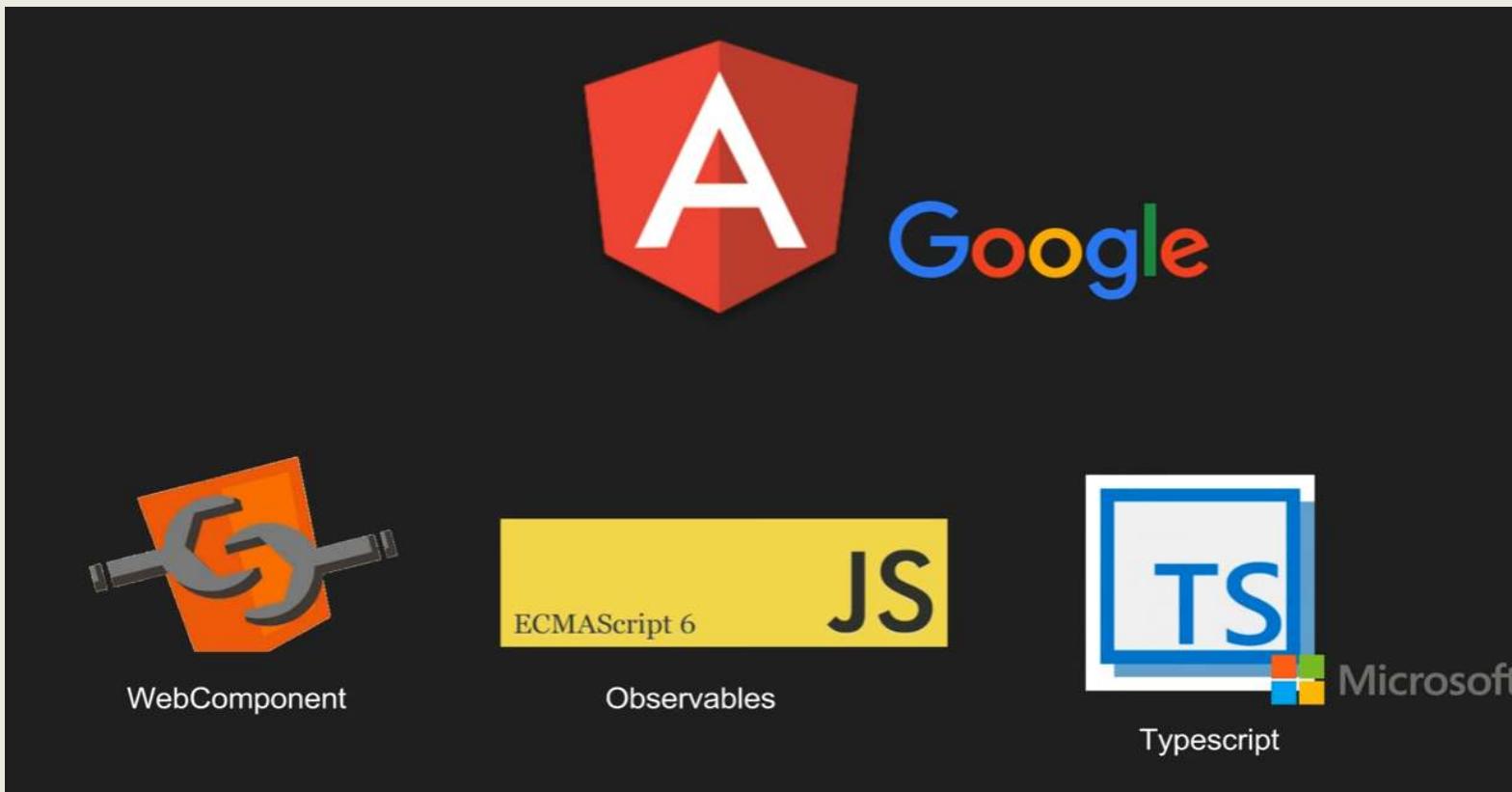
# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis.** Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

# **Introduction à Angular**

**<https://github.com/aymensellaouti/ngGk131025>**

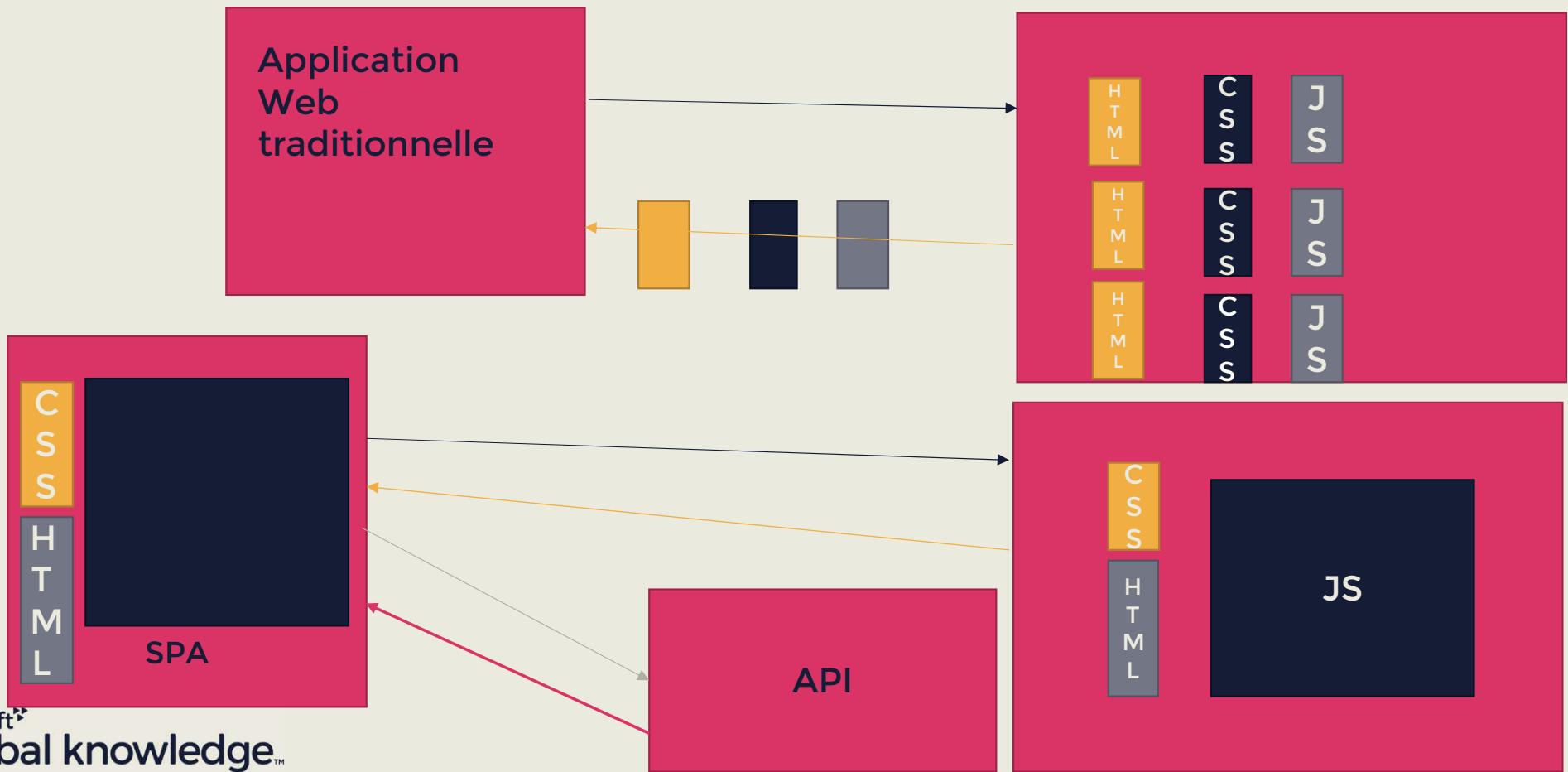
# C'est quoi Angular?



# C'est quoi Angular?

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

# SPA

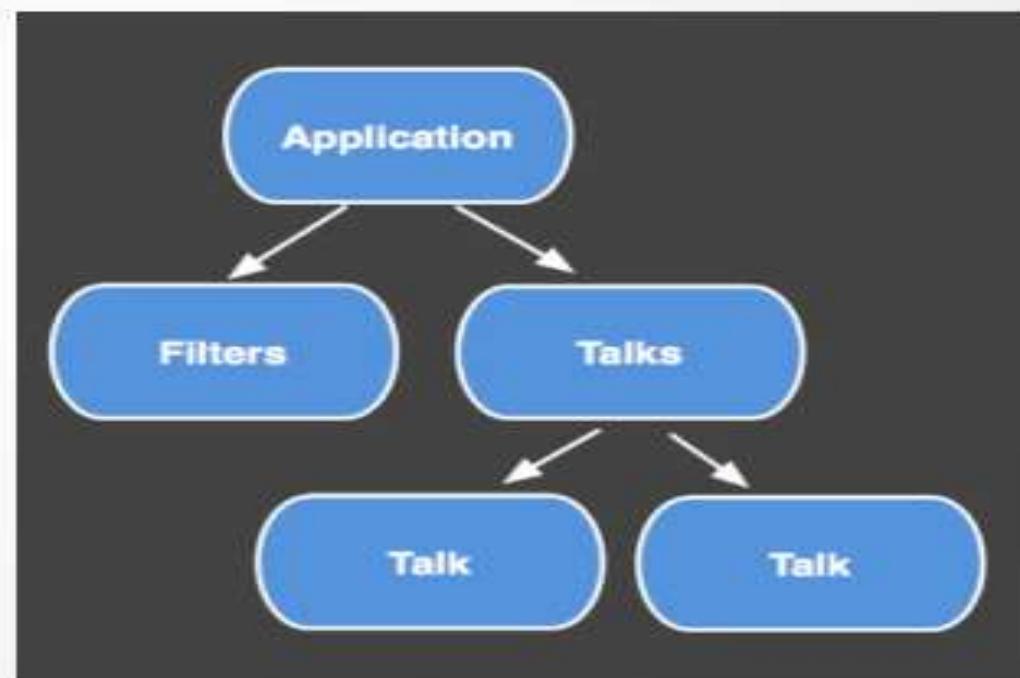


# Angular : Arbre de composants

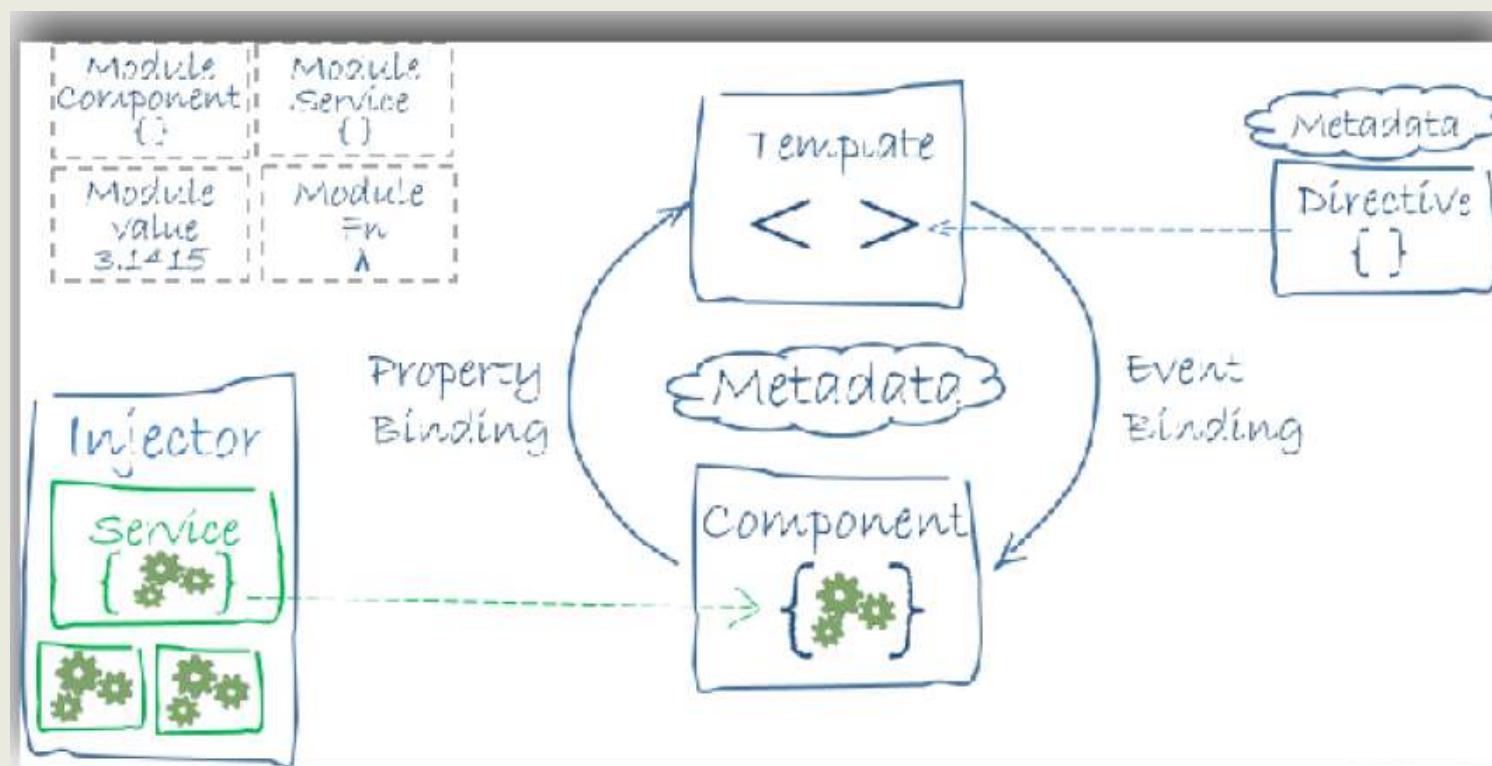
Speaker: Rich Hickey

FILTER

Rating	Title	Speaker	Action
9.1	Are We There Yet?	Rich Hickey	<a href="#">WATCH</a> <a href="#">RATE</a>
8.5	The Value of Values	Rich Hickey	<a href="#">WATCH</a> <a href="#">RATE</a>
8.2	Simple Made Easy	Rich Hickey	<a href="#">WATCH</a> <a href="#">RATE</a>



# Architecture Angular

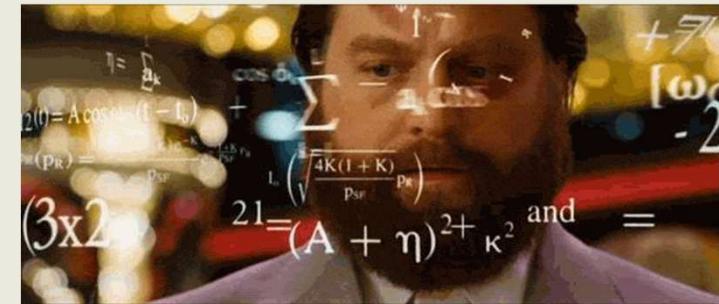


# Installation d'Angular

Deux méthodes pour installer un projet Angular.

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

# Installation d'Angular Angular Cli

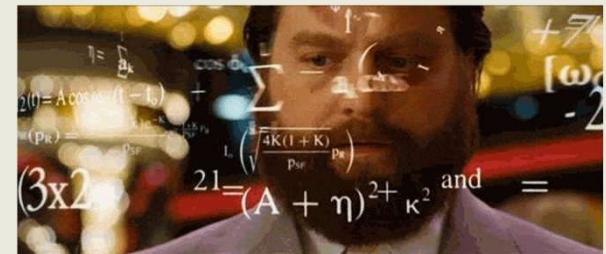


- Nous allons installer notre première application en utilisant **angular Cli**.
- Si vous avez Node c'est bon, sinon, installez **NodeJs** sur votre machine. Vous devez avoir une version de **node nécessaire pour la version Angular que vous installez**.
- Une fois installé vous disposez de npm qui est le **Node Package Manager**. Afin de vérifier si vous avez NodeJs installé, tapez **npm -v**.
- Installez 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@16.2.16 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

# Installation d'Angular Angular Cli



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

# Quelques commandes du Cli

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

# Ajouter Bootstrap

On peut ajouter Bootstrap de plusieurs façons :

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

# Ajouter Bootstrap

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

1- En l'ajoutant dans index.html

2- En ajoutant le **chemin** des dépendances dans les tableaux **styles** et **scripts** dans le fichier **angular.json**:

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

# Ajouter Bootstrap

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

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

Essayer la même chose avec font-awesome.

# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis.** Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

Aymen sellaouti

# Angular

## Les composants

# Objectifs

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

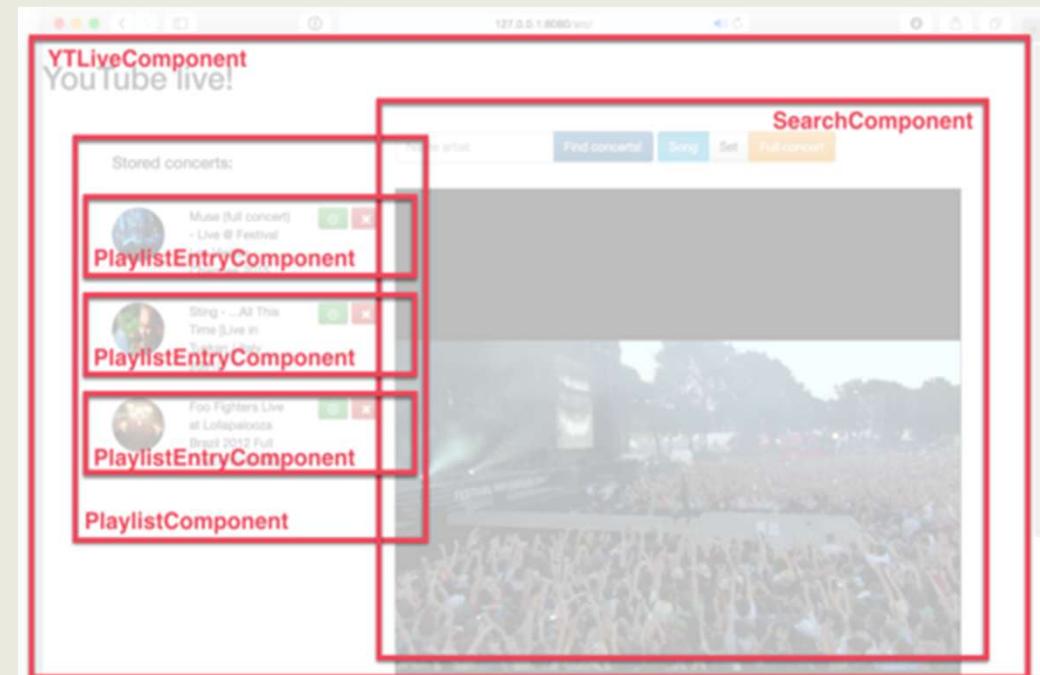
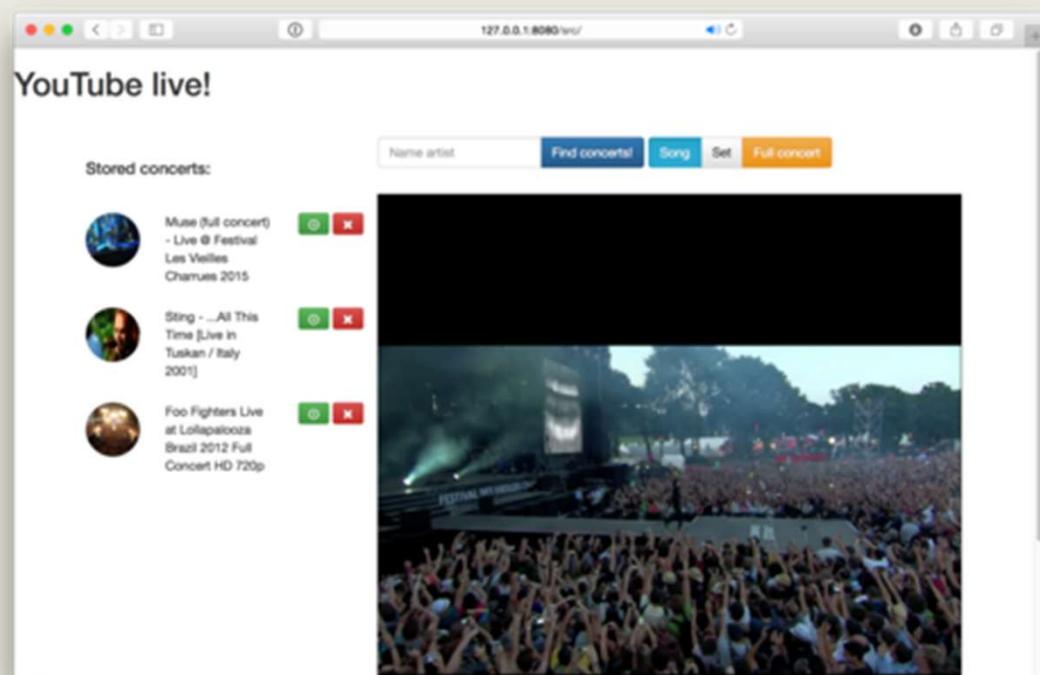
# Qu'est-ce qu'un composant (Component)

- 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 contrôler en angularJS)

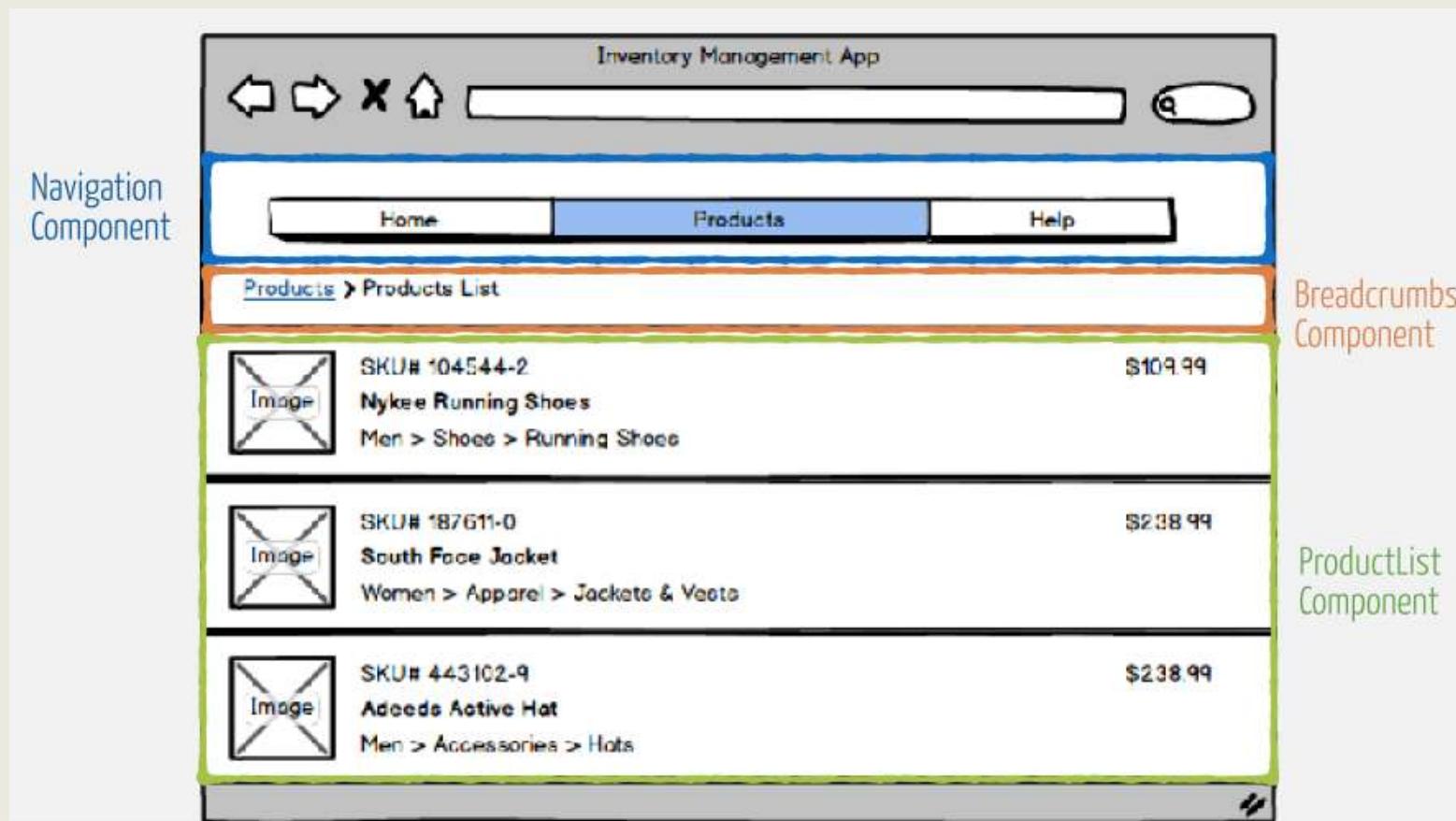
- Une application Angular est un **arbre de composants**
- La racine de cet arbre est l'application lancée par le navigateur au lancement.
- Un composant est :
  - **Composable** (normal c'est un composant)
  - **Réutilisable**
  - **Hiérarchique** (n'oublier pas c'est un arbre)

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

# Quelques exemples

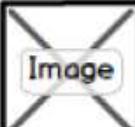


# Quelques exemples



# Quelques exemples

Product Row  
Component

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

# Quelques exemples



# Premier Composant

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

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

## Chargement de la classe Component

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

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

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

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

**Export de la classe** afin de pouvoir l'utiliser

# Création d'un composant

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

# Création d'un composant

- La commande `generate` possède plusieurs options

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

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

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

```
<!-- 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 :



# 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;  
  }  
}
```

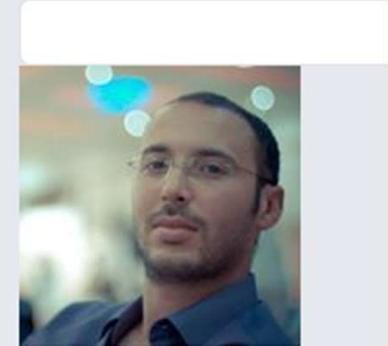
```
<!-- attributs: L'état de la balise -->  
<p hidden> First Component works!</p>  
<p hidden> {{name}} works!</p>  
<div class="alert alert-info">  
  Je test le binding  
</div>  
<p>Le contenu de l'input est : </p>  
<input  
  type="text"  
  class="form-control"  
>  

```

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

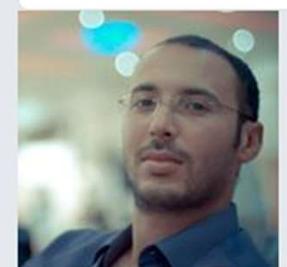
```
export class FirstComponent {  
  //Propriétés : état State  
  name = 'First Component';  
  isHidden = false;←  
  message = '';  
  imagePath = 'assets/images/as.jpg';  
  //Méthodes : comportement Behaviour  
  showHide() {  
    this.isHidden = !this.isHidden;  
  }  
  changeMessage(newMessage: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

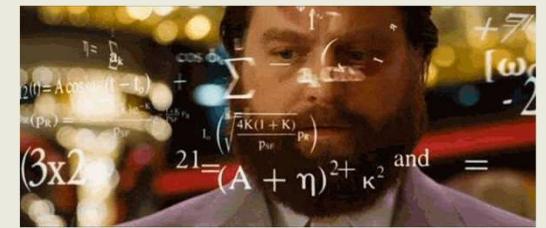
First Component works!

Je test le binding

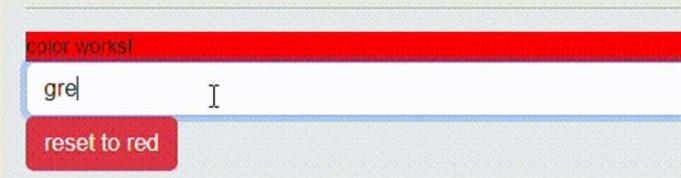
Le contenu de l'input est :



# 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]`

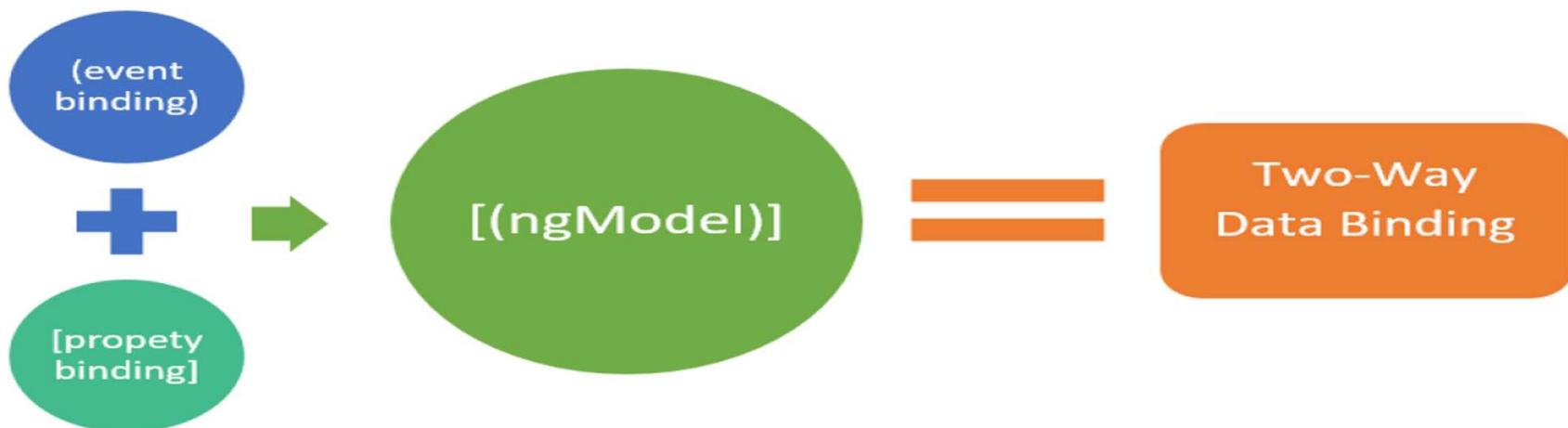


# Two way Binding

- 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 t Binding

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

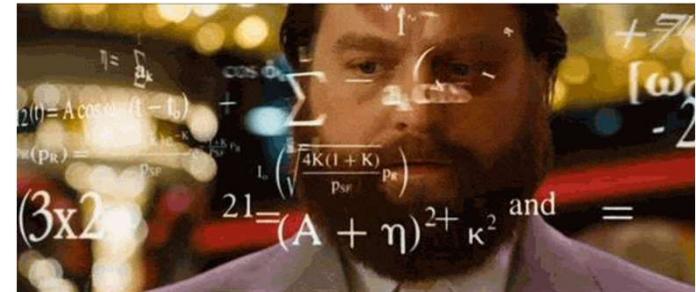
@Component({
  selector: 'app-two-way',
  templateUrl: './two-
way.component.html',
  styleUrls: ['./two-
way.component.css']
})
export class TwoWayComponent {
  two:any="myInitValue";
}
```

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 mises à jour.
- Préparer l'affichage de la carte visite. Vous pouvez utiliser ce thème gratuit :

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

# Exercice



Aymen Sellaouti  
trainer

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

Auto Rotation

name :  
sellaouti

firstname :  
aymen

job :  
trainer

path :  
rotating\_card\_profile3.png

# Exercice

## Two Way Binding



Sellaouti Aymen  
Enseignant  
tant qu'il y a de la vie il y a de l'espoir

Auto Rotation

Nom :

Prénom :

Job :

image :

Citation Favorite :

Décrivez nous votre travail :

Mots clé de votre travail :

## 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 Followers    114 Following    35 Projects

f g+ t

Nom :

Prénom :

Job :

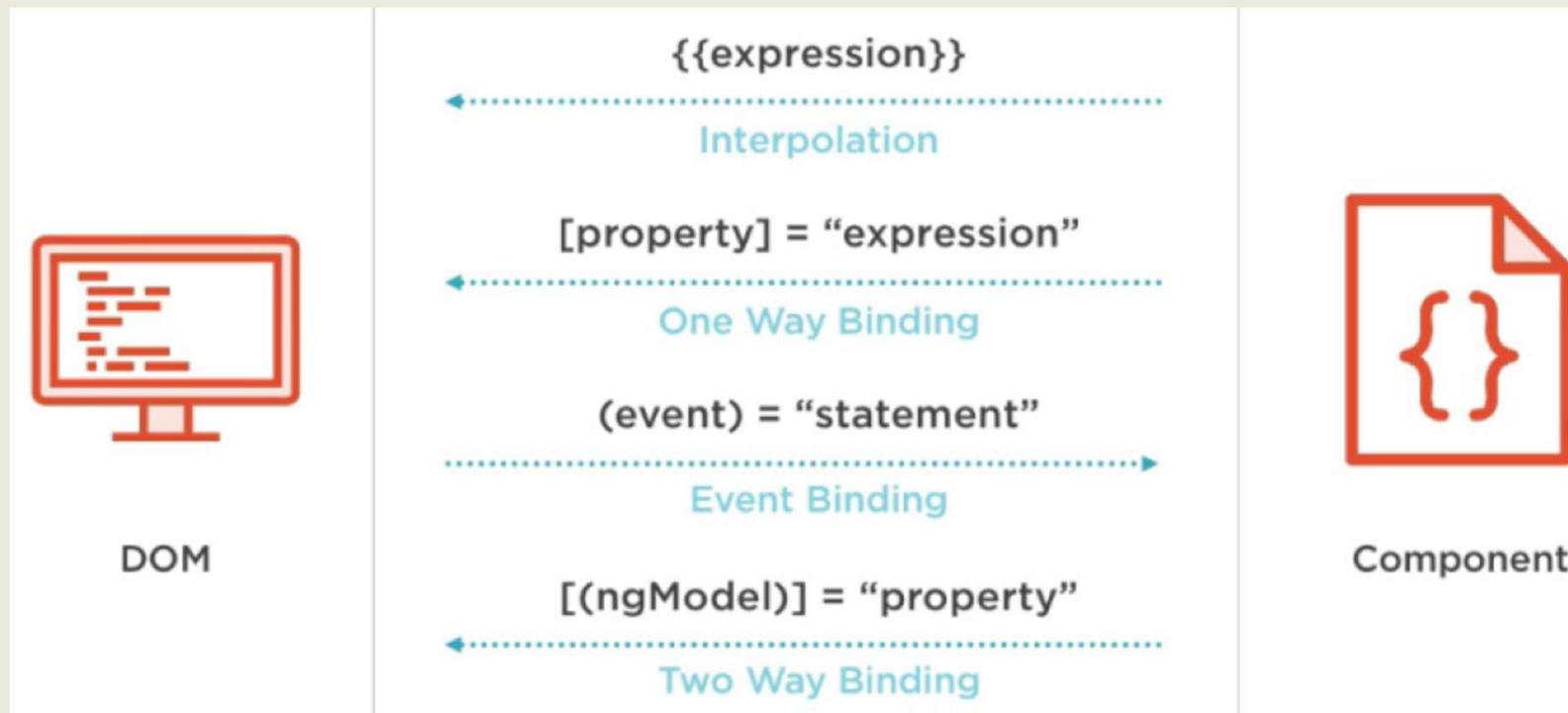
image :

Citation Favorite :

Décrivez nous votre travail :

Mots clé de votre travail :

# Résumé : Property Binding



# Récap Binding

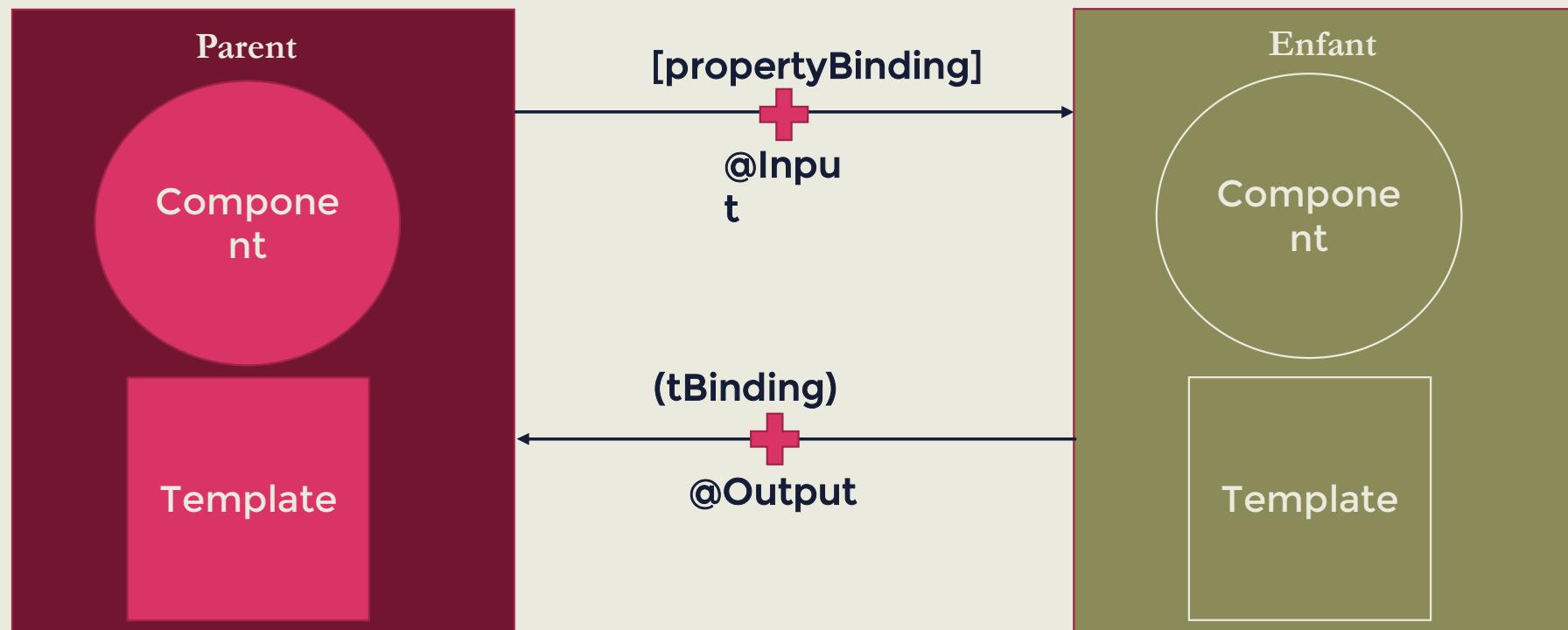
```
<div [style.backgroundColor]="color">  
  Color  
</div>  
  
<input [(ngModel)]="color"  
  type="text"  
  class="form-control"  
>  
  
le contenu de la propriété color est {{color}}  
<button (click)="loggerMesData()">log data</button>  
<br>  
<a (click)="goToCv()">Go to Cv</a>
```

HTML

```
@Component({  
  selector: 'app-color',  
  templateUrl: './color.component.html',  
  styleUrls: ['./color.component.css'],  
  providers: [PremierService]  
})  
export class ColorComponent implements OnInit {  
  color = 'red';  
  constructor() { }  
  
  ngOnInit() {}  
  processReq(message: any) {  
    alert(message);  
  }  
  loggerMesData() {  
    this.premierService.logger('test');  
  }  
  goToCv() {  
    const link = ['cv'];  
    this.router.navigate(link);  
  }  
}
```

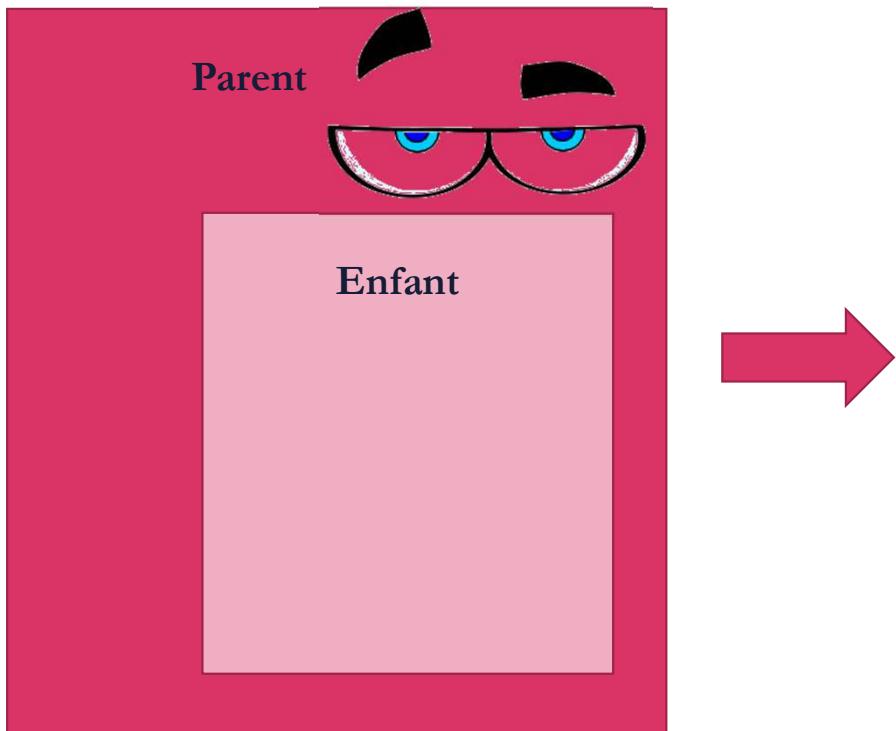
TS

# Interaction entre composants



# Pourquoi ?

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



```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-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'envoie d'une valeur est required par exemple).

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

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

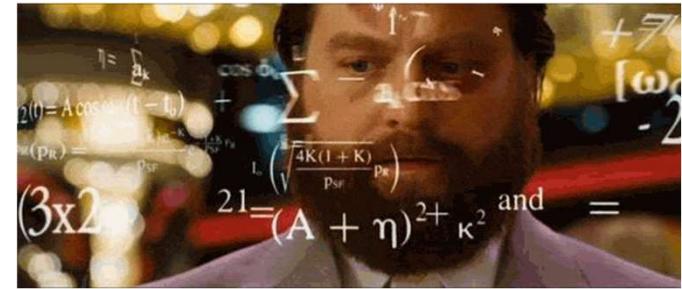
```
<app-fils [colour]="color"/>
```

Template du  
pere

Ts du fils

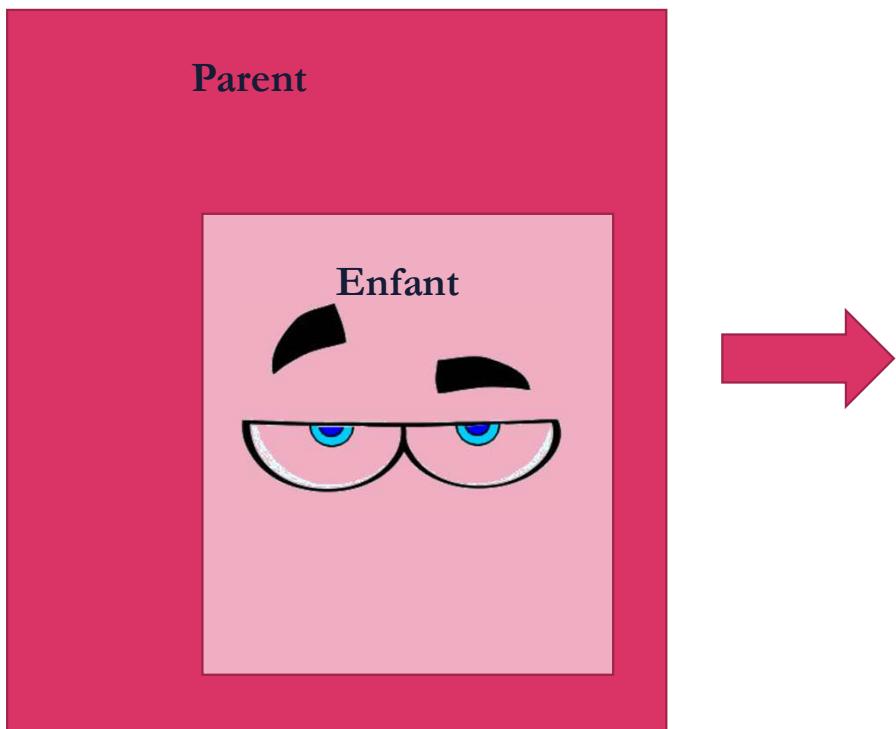
# Exercice

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



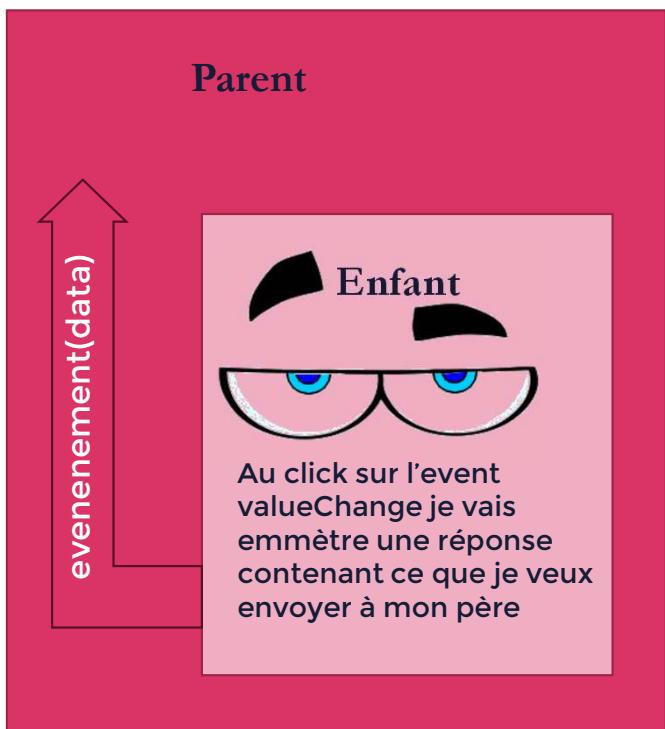
# Interaction du fils vers le père

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();
}
```

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

Template du  
père

Template du  
fils

# Interaction du fils vers le père

## 2. Configurer l'événement

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

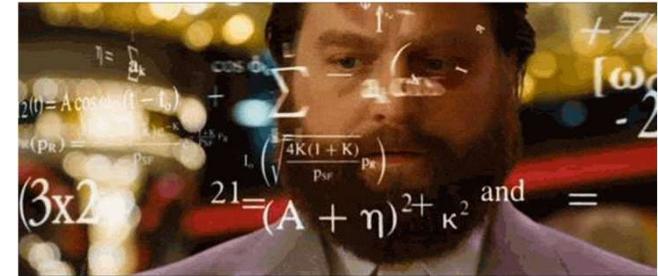
## 3. Récupérer l'événement dans le composant parent

```
<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);  
}
```

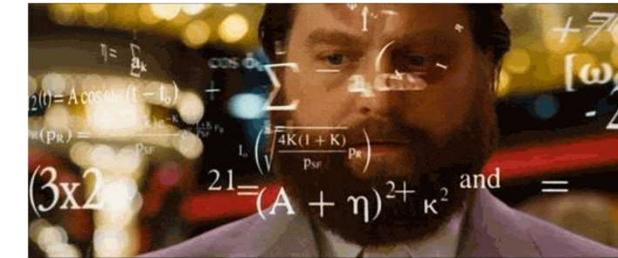
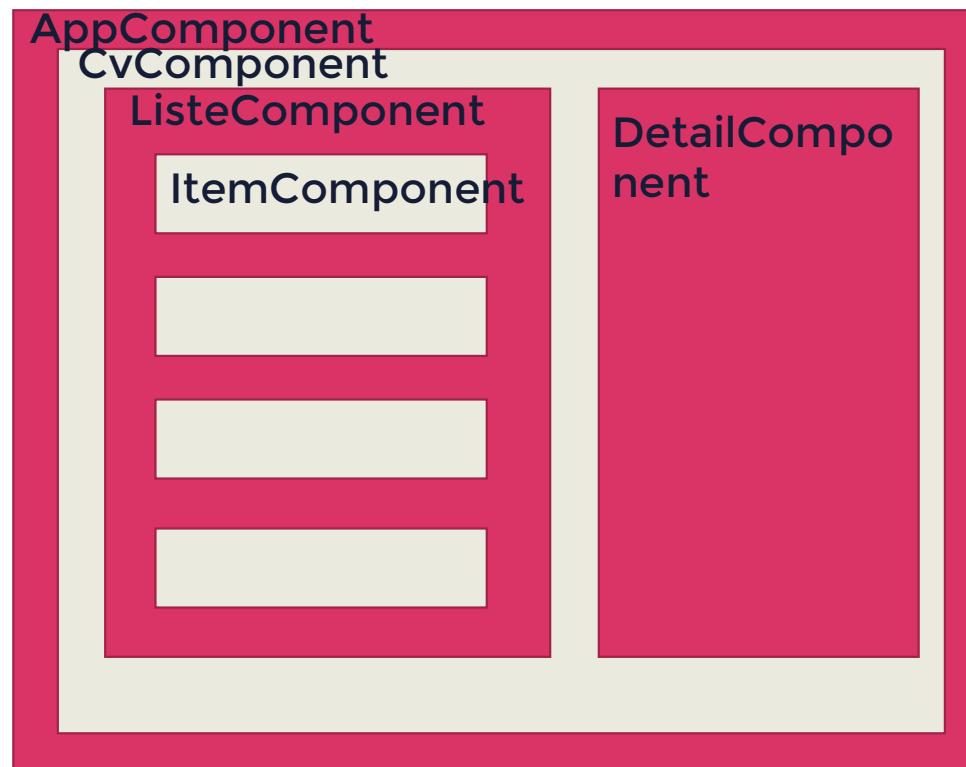
# Exercice



- 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.

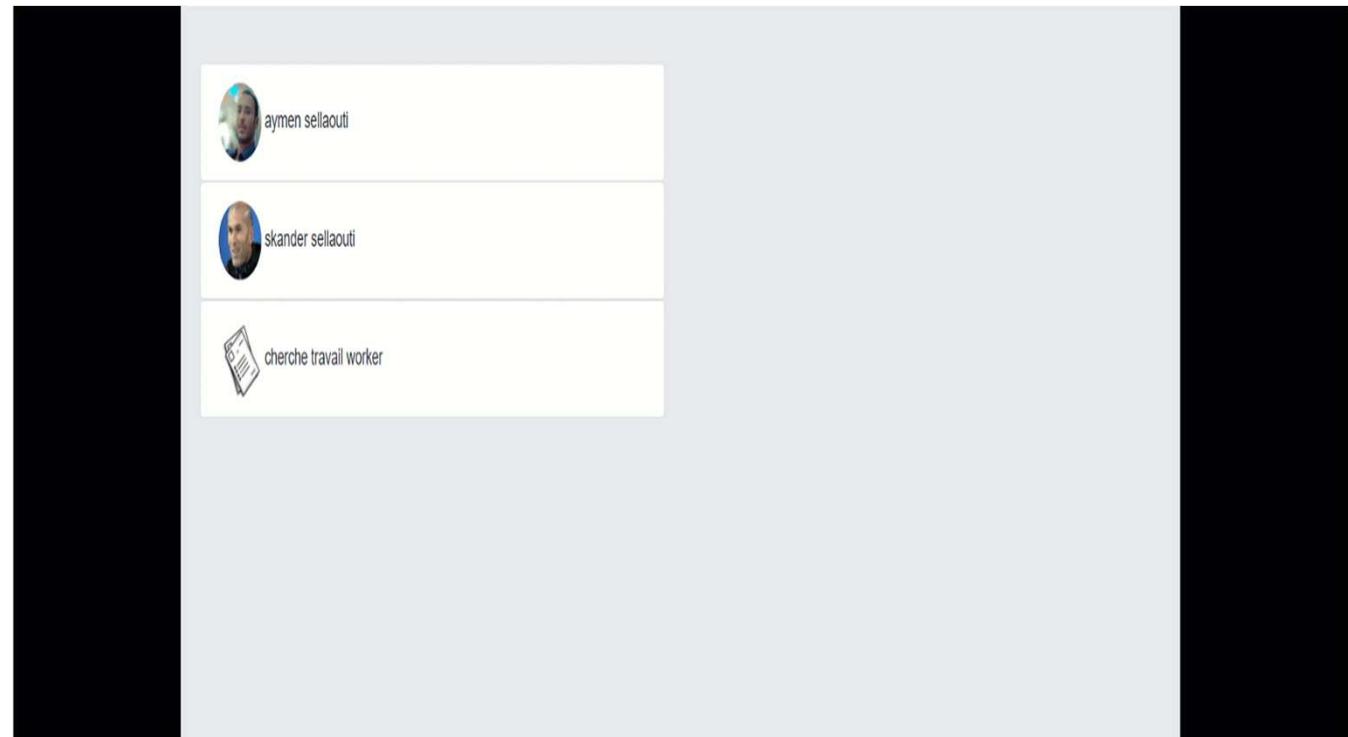
# Exercice

- 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. Il faudra suivre cette architecture.

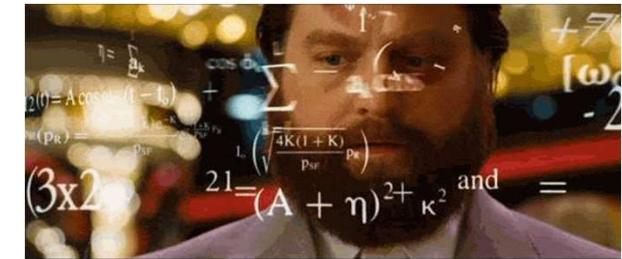


# Exercice

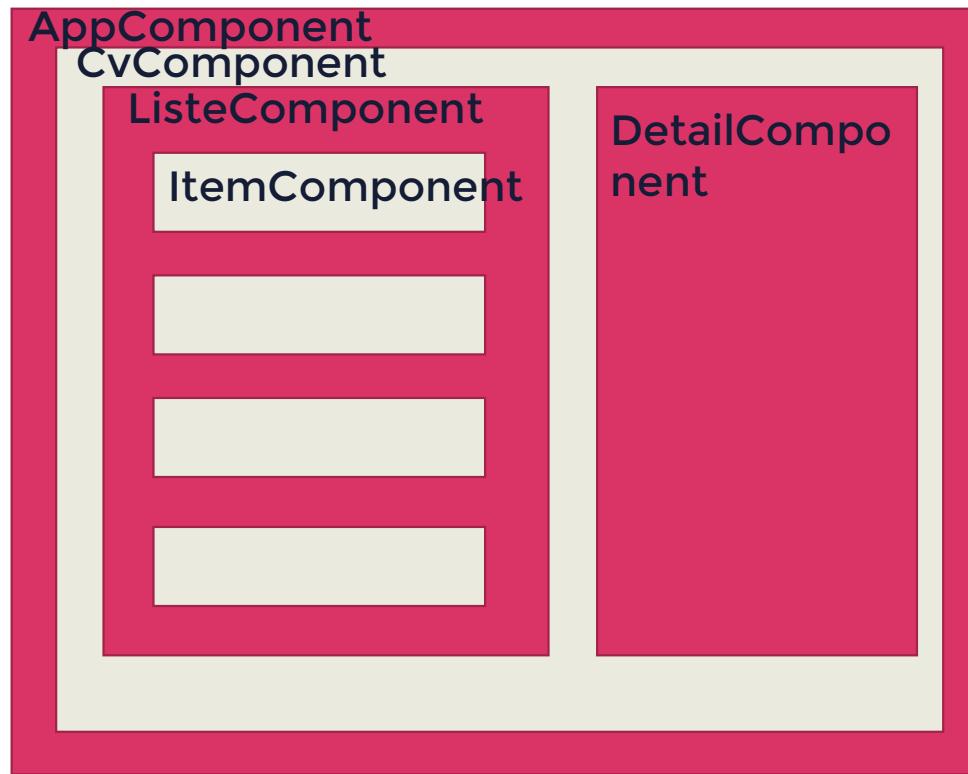
- Le but de cet exercice est de créer une mini plateforme de recrutement.
- La première étape est de créer la structure suivante avec une vue contenant deux parties :
  - Liste des Cvs inscrits
  - Détail du Cv qui apparaitra au click
- Il vous est demandé juste d'afficher un seul Cv et de lui afficher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



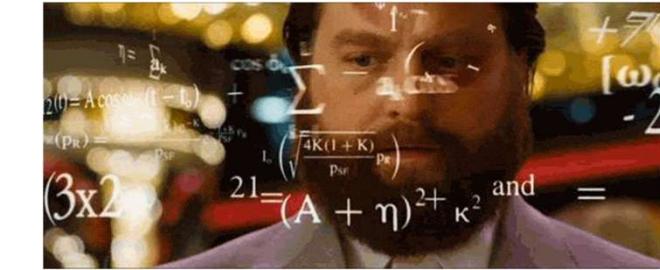
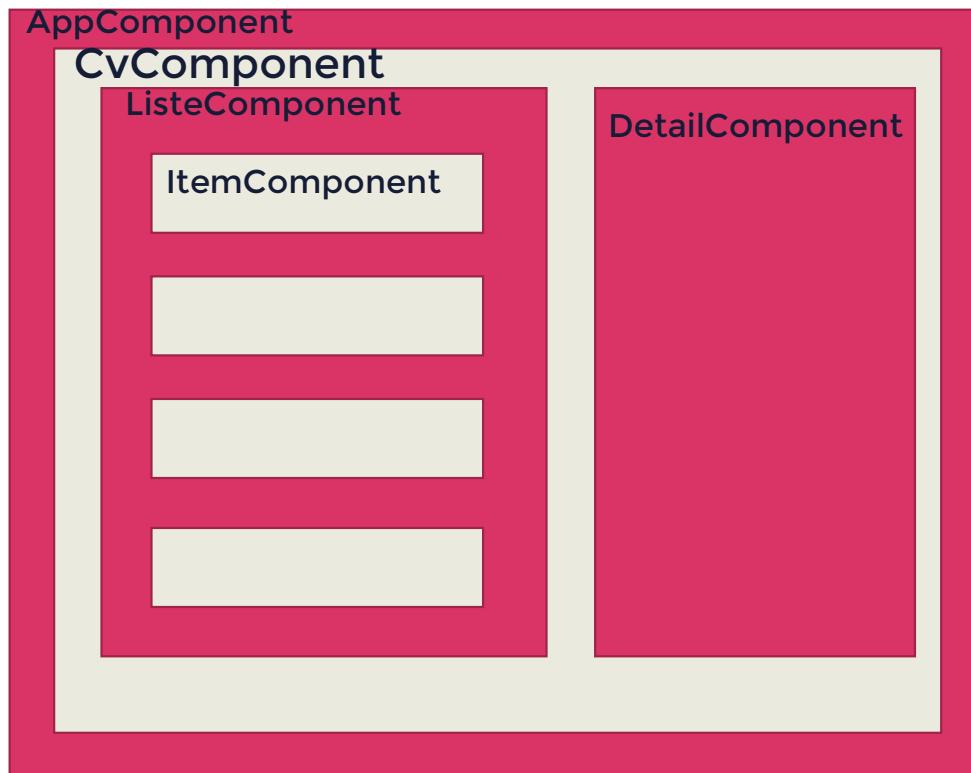
The image shows a mobile application interface. On the left, there is a large black area. In the center, there is a white rectangular list view containing three items. Each item has a small circular profile picture on the left and the user's name on the right. The first item is 'aymen sellaouti', the second is 'skander sellaouti', and the third is 'cherche travail worker'. The background of the entire screen is dark.



# Exercice



# Exercice



A profile view of a user's profile picture and details:

- Aymen Sellaouti
- Teacher
- 36

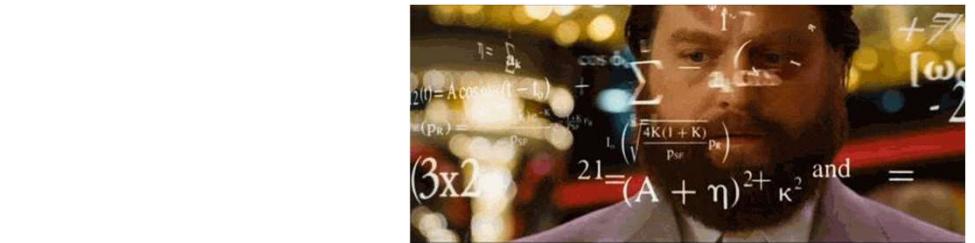
**Au click sur le Cv les détails sont affichés**

Auto Rotation

# Exercice

Un cv est caractérisé par :

- id
- name
- firstname
- Age
- Cin
- Job
- path



	sellaouti aymen
	sellaouti skander

Aymen Sellaouti  
Teacher

36

Auto Rotation

Au click sur le Cv les détails sont affichés

# Cycle de vie d'un composant

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

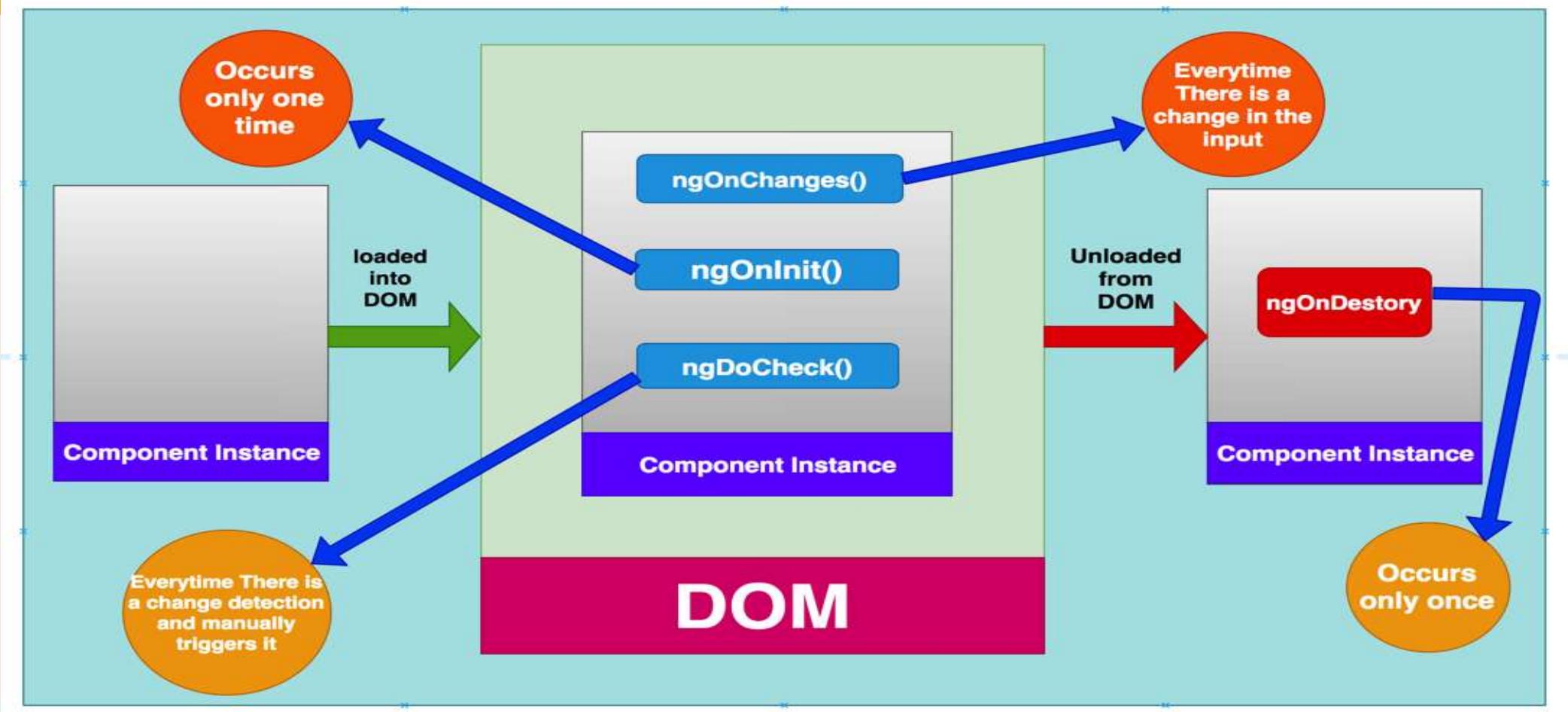
<https://medium.com/bhargav-bachina-angular-training/angular-understanding-angular-lifecycle-hooks-with-a-sample-project-375a61882478>

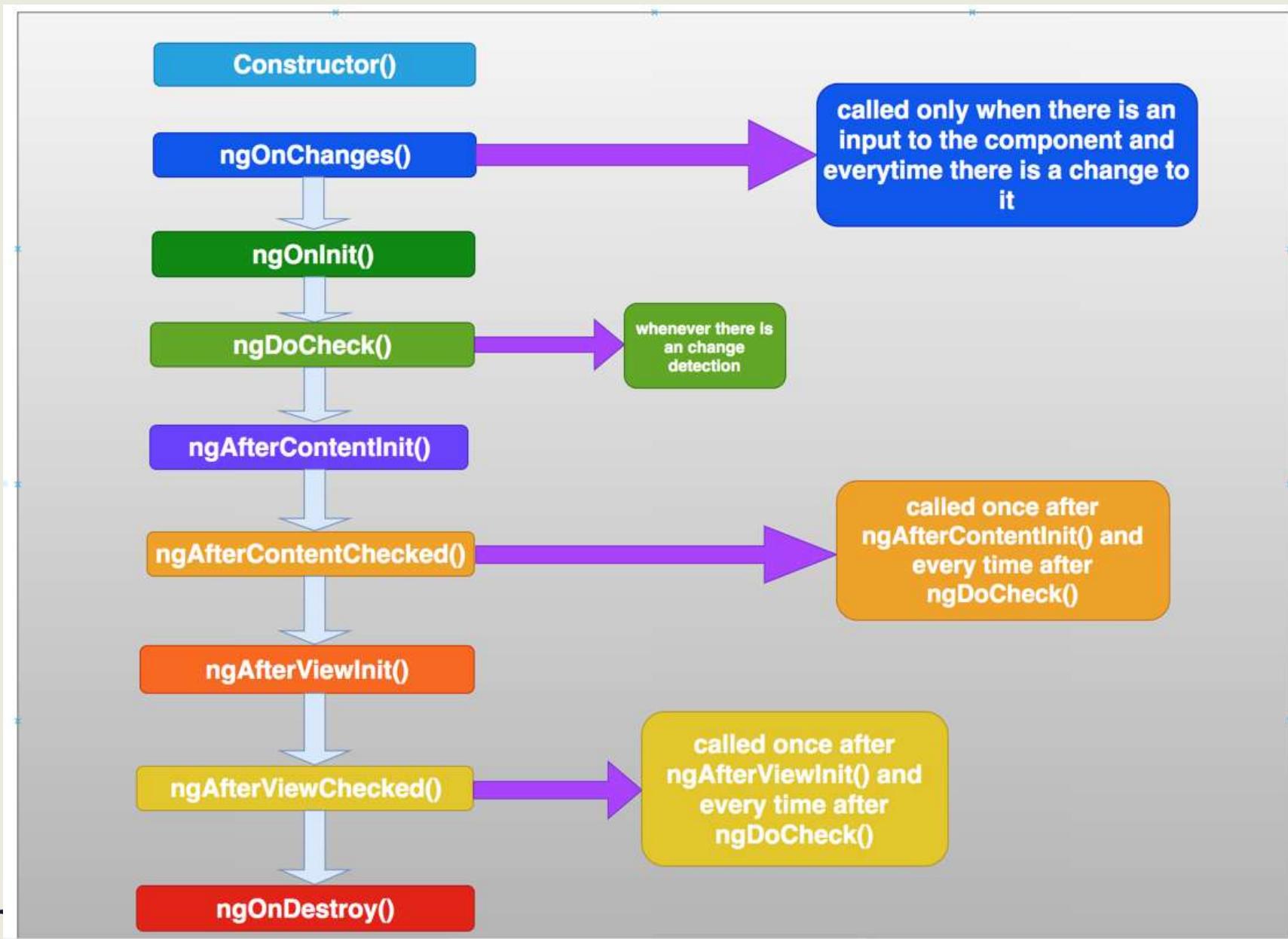
# Interfaces de gestion du cycle de vie d'un composant

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

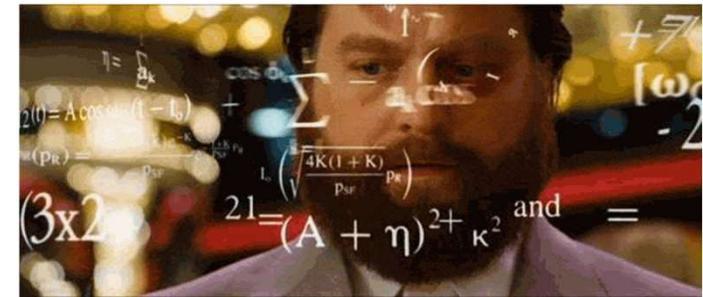
**ng + NomDeL'Interface**

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





# Exercice



- Créer un composant `ShowIsEven` qui contient un input de type number.
- Créer un composant `isEven` qui récupère un entier en input et qui affiche s'il est pair ou impair.
- Ce composant doit appeler le composant `isEven` et lui passer en paramètre la valeur de l'input.



# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

Aymen sellaouti

# Angular

## Les directives

# 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**

# 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 faché contre  
    moi et m'a caché :(  
</p>
```

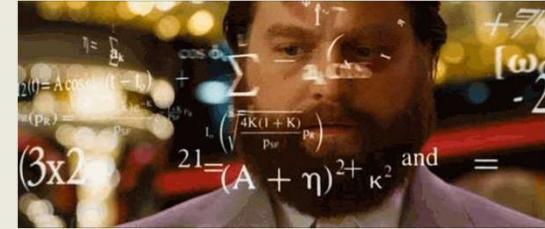
# Les directives structurelles \*ngFor

- 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.

CvTech Home Cv Color Task Manager Poc Add Students Login

Aymen Sellaouti

Zineddine Zidan

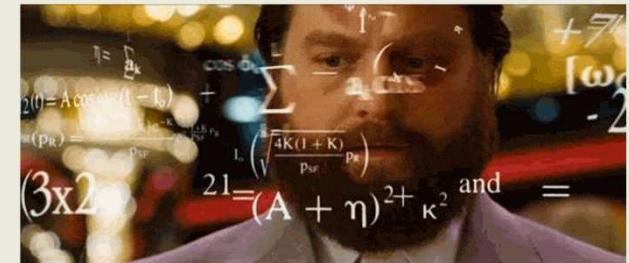
# Les directives d'attribut (ngStyle)

- 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'.

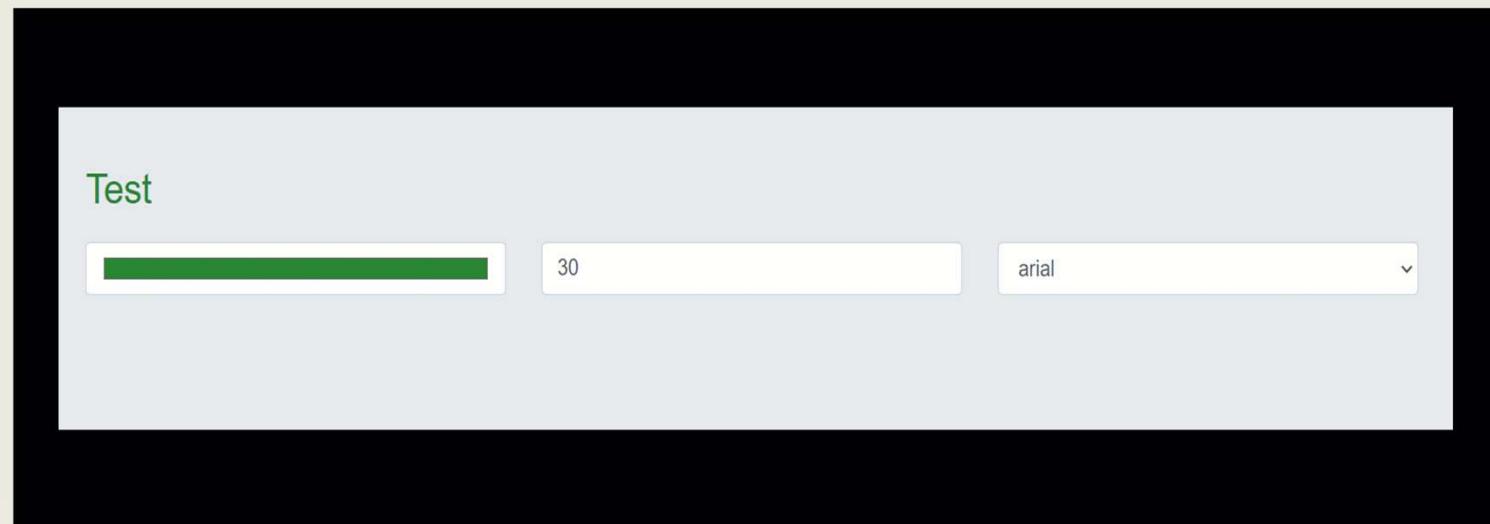
```
@Component({  
  selector: 'app-ngstyle-exemple',  
  templateUrl: './ngstyle-exemple.component.html',  
  styleUrls: ['./ngstyle-exemple.component.css']  
})  
export class NgstyleExempleComponent {  
  color = 'lightblue';  
}
```

```
<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.
  - Faites 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.



# 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 appliquée à la Div

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

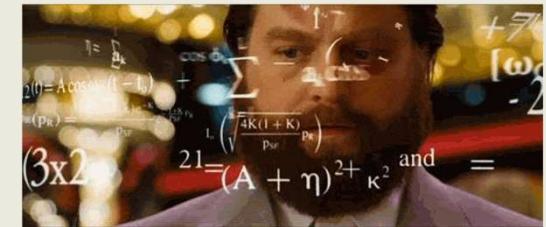
```
<div  
[ngClass]="{  
  on: isOn,  
  off: !isOn  
}"  
class="ampoule">lampe</div>
```

# 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



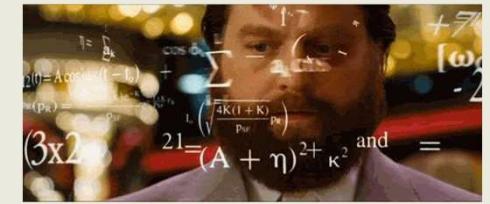
- Afin d'améliorer l'affichage de la liste des cvs dans votre CvTech, reprenez le composant cvList et faites-en sortes d'avoir les éléments successifs colorés d'une manière permutée entre deux couleurs.



# 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 `HostListener` il faut les importer du `core` d'angular

# Exercice



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

- Créez une directive
- Créez un hostbinding sur la couleur et la couleur de la bordure.
- Préparez une méthode qui permet de générer aléatoirement une couleur dans votre directive.
- Faites 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.

The screenshot shows a browser's developer tools with the DOM tree and styles panel open. The DOM tree highlights an  element with the class "apprainbow". The styles panel shows the following CSS:

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <div class="container">
      <app-root _ngcontent-pfp-c0 ng-version="8.2.14">
        <app-ng-style _ngcontent-pfp-c0 _ngcontent-pfp-c1>
          ...
            <input _ngcontent-pfp-c1 apprainbow class="form-control" style="border-color: #123456; color: #123456;"> == $0
        </app-ng-style>
      </app-root>
    </div>
  </body>
</html>
```

Styles Computed Event Listeners DOM Breakpoints >

element.style {  
 border-color: #123456;  
 color: #123456;  
}

# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes**
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

Aymen sellaouti

# Angular

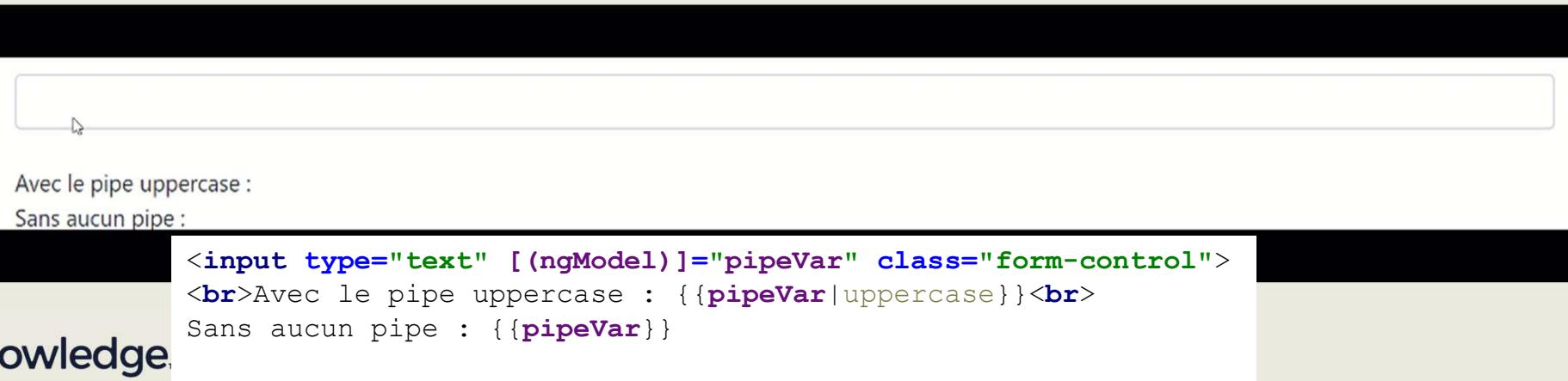
## Les pipes

# Objectifs

- 1. Définir les pipes et l'intérêt de les utiliser**
- 2. Vue globale des pipes prédéfinies**
- 3. Créer une pipe personnalisée**

# Qu'est-ce qu'une pipe

- Une 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 offertes par Angular et prêt à l'emploi.
- Vous pouvez créer vos propres pipes.



Avec le pipe uppercase :

Sans aucun pipe :

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

# Syntaxe

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

# Les pipes disponibles par défaut (Built-in pipes)

- La documentation d'angular vous offre la liste des pipes prêt à l'emploi.

<https://angular.io/api?type=pipe>

- uppercase
- lowercase
- titlecase
- currency
- date
- json
- percent
- ...

# Paramétriser une pipe

- Afin de paramétriser les pipes ajouter ':' après le pipe suivi de votre paramètre.
  - {{ maDate | date:"MM/dd/yy" }}
- Si vous avez plusieurs paramètres c'est une suite de ':'
  - {{ nom | slice:1:4 }}

# Pipe personnalisée

- Une pipe personnalisée est une classe décorée 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**
- La pipe doit être déclaré au niveau de votre module de la même manière qu'une directive ou un composant.
- Pour créer une 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 {

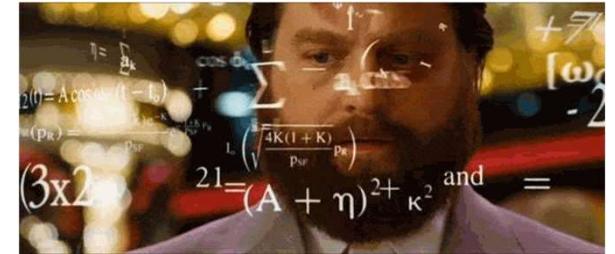
  transform(value: any, args?: any): any {
    switch (value) {
      case 'barca' : return ' blaugrana';
      case 'roma' : return ' giallorossa';
      case 'milan' : return ' rossoneri';
    }
  }
}
```

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

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

# Exercice

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



# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes**
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

Aymen sellaouti

# Angular Service et injection de dépendances



# 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(){};
```

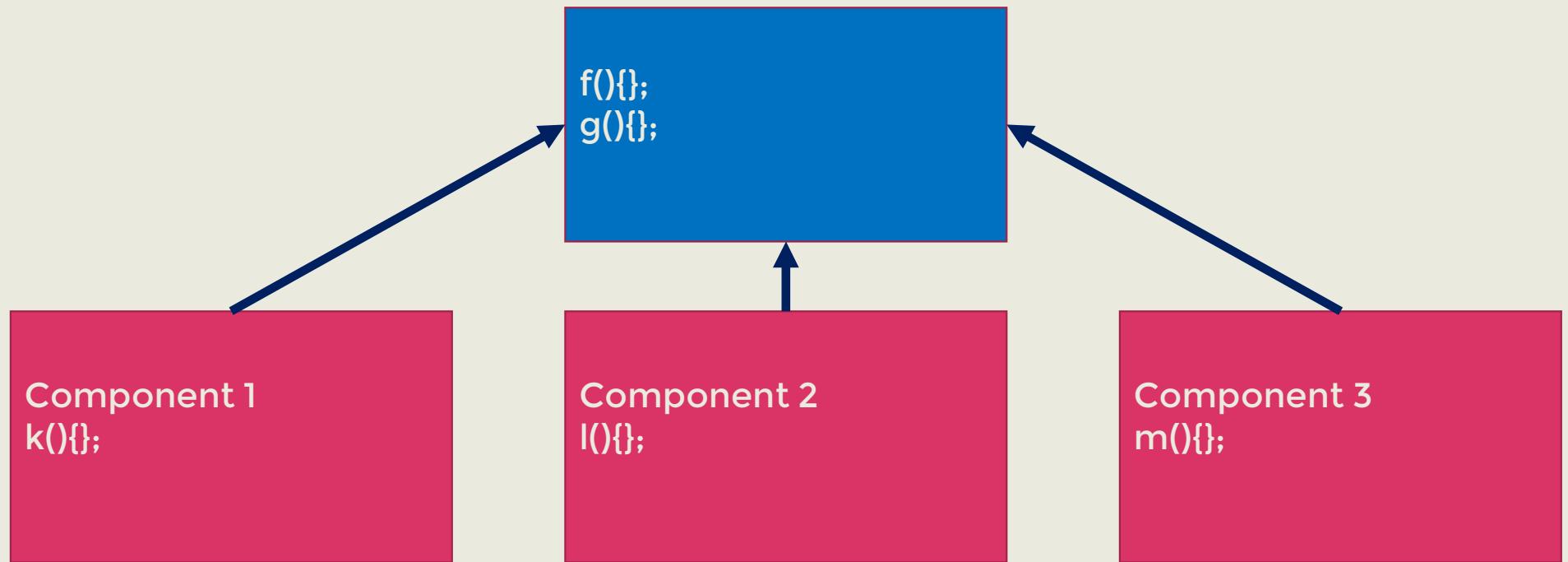
Component 3

```
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

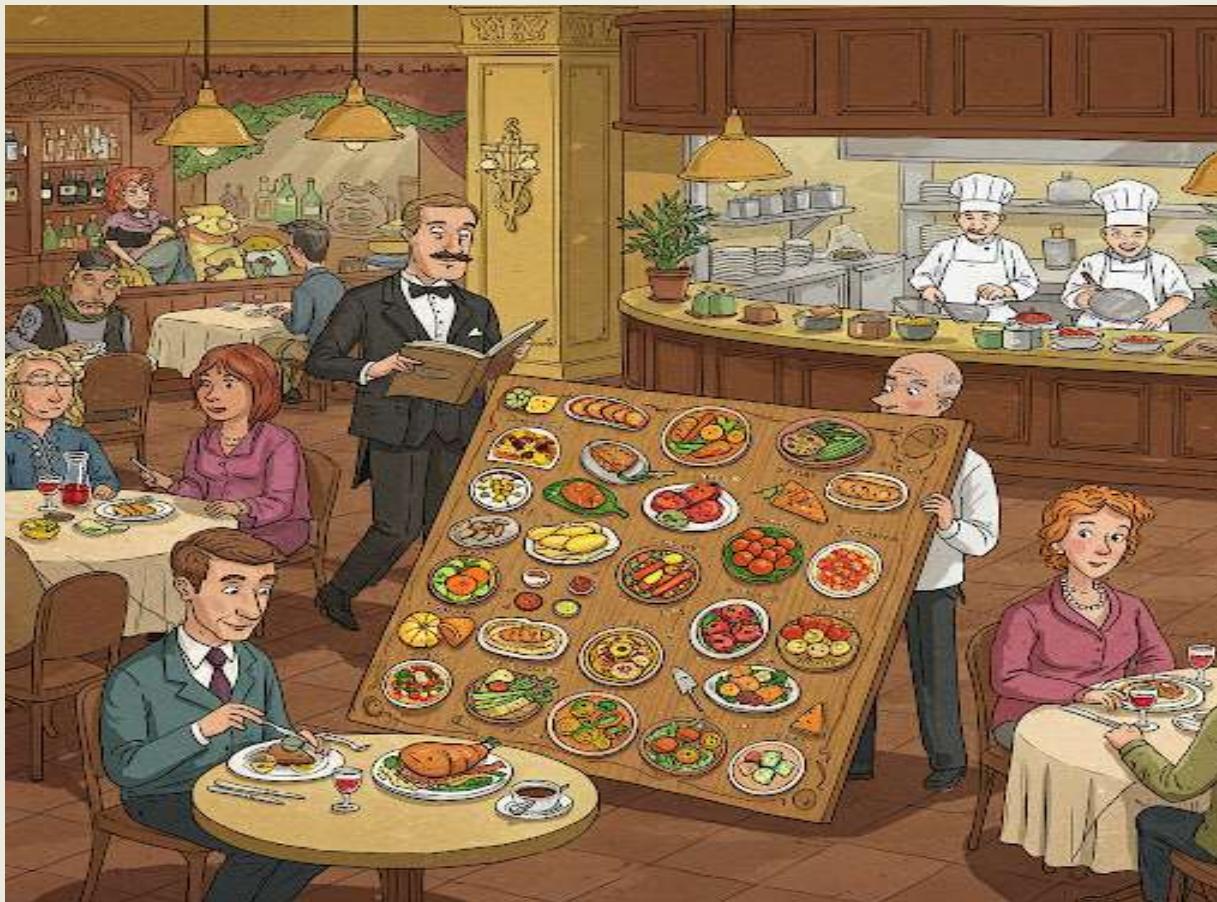
```
import { Injectable } from '@angular/core';

@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 linstanciation de ces classes dans les différentes classes qui en dépendent ?

# Injection de dépendance (DI)



- Déléguer cette tache à une entité tierce.

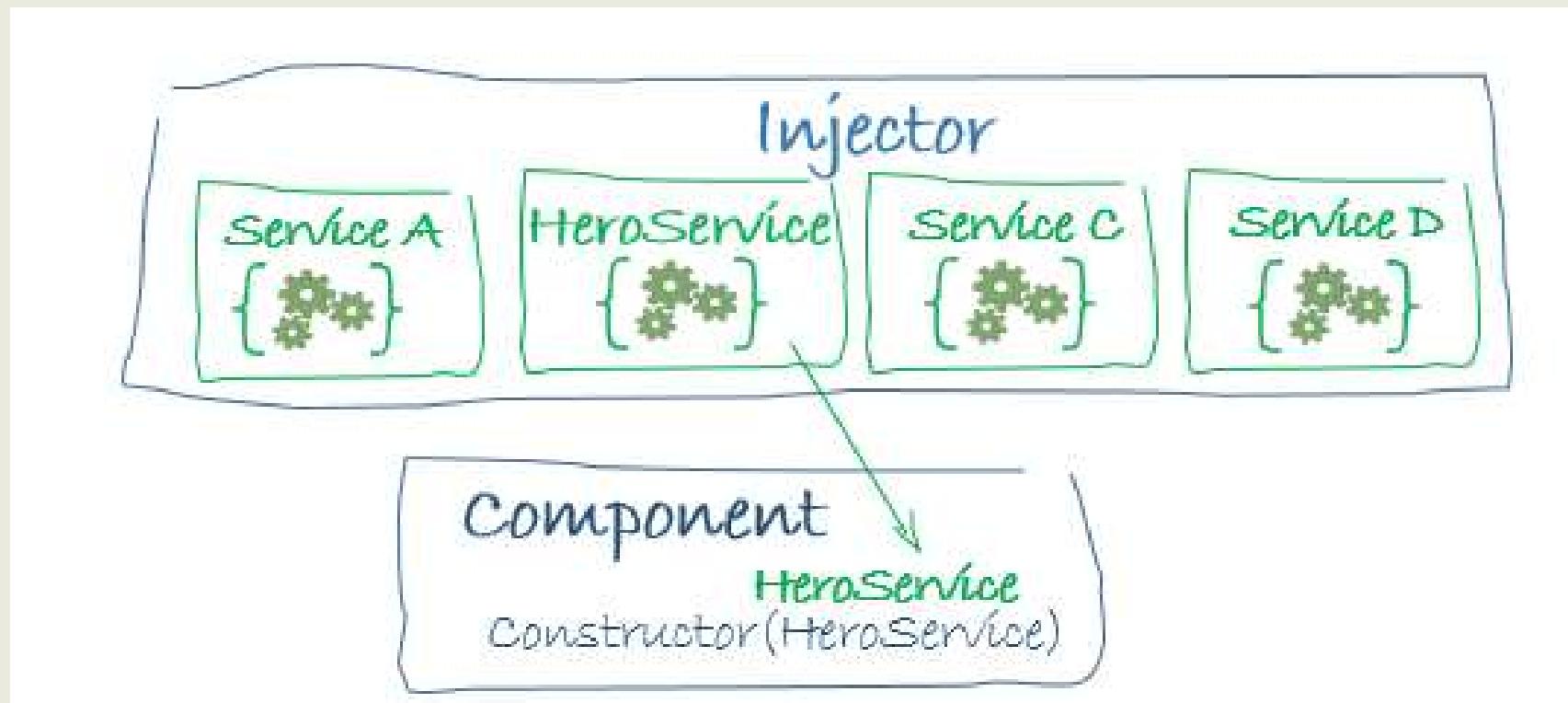
Classe A1{  
Constructor(B b,  
C c)  
...  
}

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

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

INJECTOR

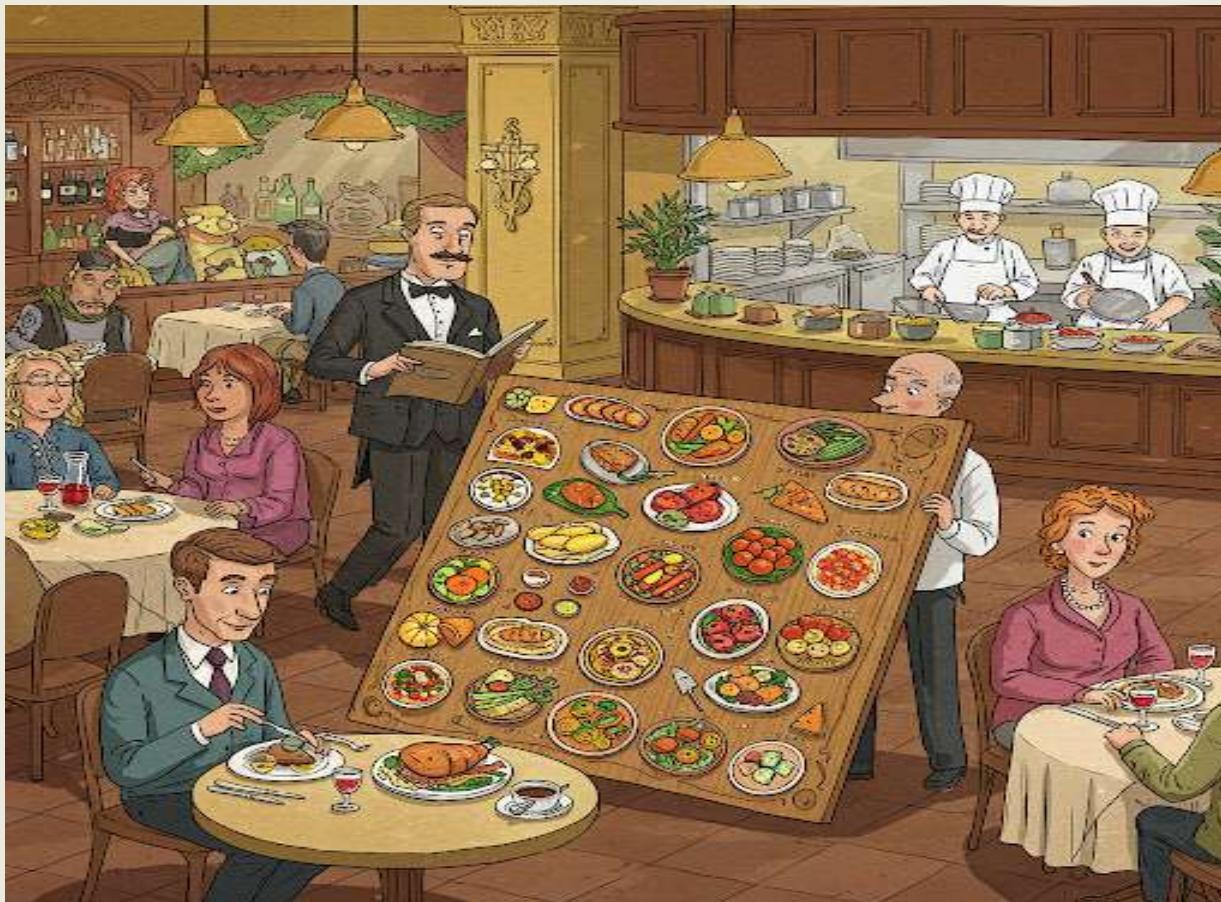
# Injection de dépendance (DI)



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

Provider  
s  
Injector

IOC  
Injection



# Injection de dépendance (DI)

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

# Injection de dépendance (DI)

- L'injection de dépendance utilise les étapes suivantes :
  - 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

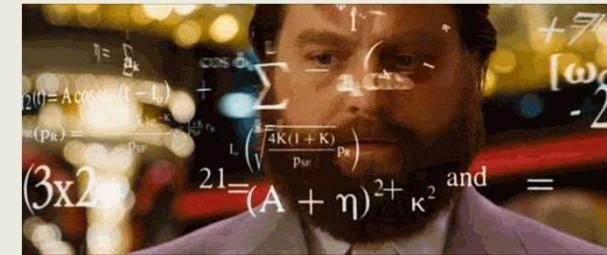
- C'est un décorateur permettant de rendre une classe injectable
- Une classe est dite injectable si on peut y injecter des dépendances
- **@Component, @Pipe, et @Directive** sont des sous classes de **@Injectable()**, ceci explique le fait qu'on peut y injecter directement des dépendances.
- Si vous n'allez 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

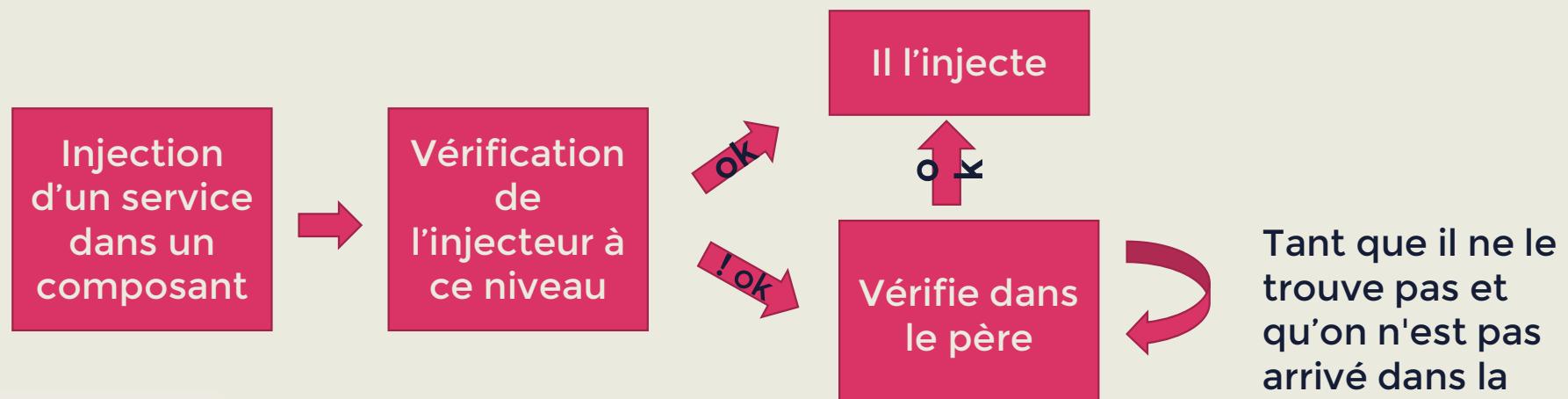


- Créons un service de Todo, le Model et le composant qui va avec. Un Todo est caractérisé par un nom et un contenu.
- Ce service permettra de faire les fonctionnalités suivantes :
  - Logger les todos
  - Ajouter un Todo
  - Récupérer la liste des Todos
  - Supprimer un Todo

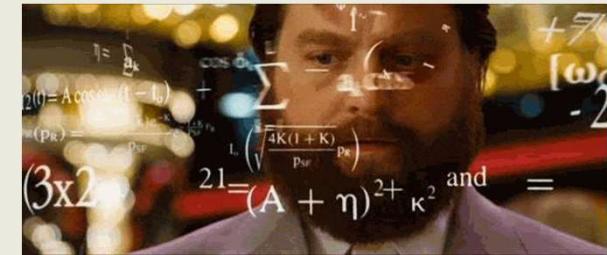
A screenshot of a web application interface. It features a form with two input fields: 'Name:' containing 'I' and 'Content:' containing an empty text area. Below the form is a blue 'Add Todo' button. The entire interface is set against a light gray background with black horizontal bars at the top and bottom.

# 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](#) [Rapis](#) [Logout](#)

[Français](#)

[English](#)



Mehdi Chaabane



Mehdi Buh



Aymen Sellaouti

"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

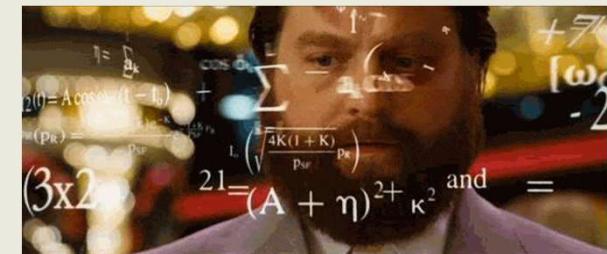
35

Projects

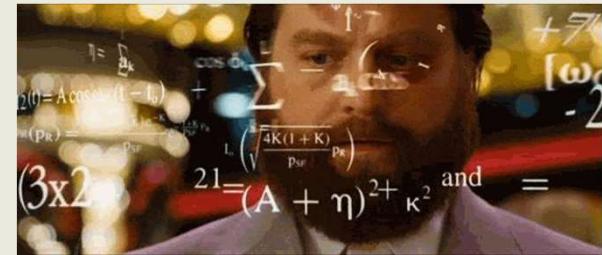
[embaucher](#)

[détails](#)

Footer



# Exercice



sellaouti aymen

sellaouti skander

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

12345678

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

235 Followers

114 Following

35 Projects

[Embaucher](#)

## Liste des cvs sélectionnés pour embauche



# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes**
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

Aymen sellaouti

# Angular Routing

# Objectifs

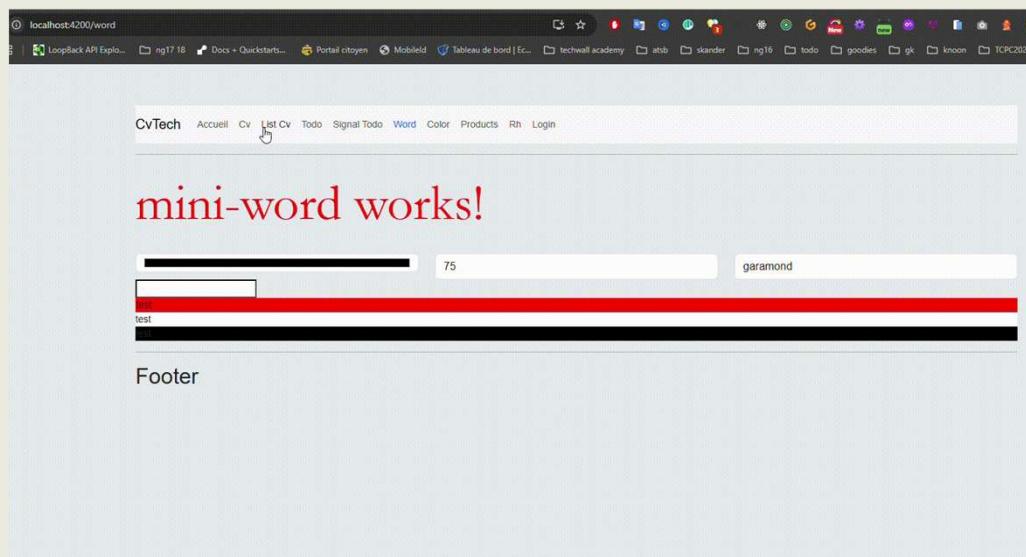
- 1. Définir le routeur d'Angular**
- 2. Définir une route**
- 3. Déclencher une route à partir d'un composant**
- 4. Ajouter des paramètres à une route**
- 5. Récupérer les paramètres d'une route à partir du composant.**
- 6. Préfixer un ensemble de routes**
- 7. Gérer les routes inexistantes**

# Qu'est ce que le routing

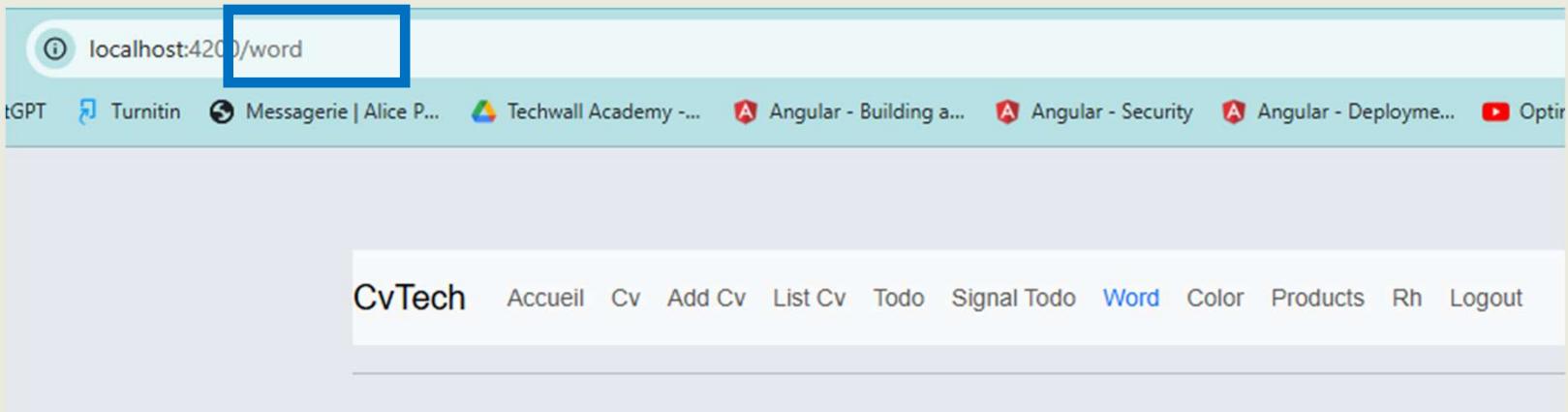
- Tout système de routing permet d'associer une URI à un traitement
- Angular est une SPA. Pourquoi parle-on de route ??
  - Séparer différentes fonctionnalités du système
  - Maintenir l'état de l'application
  - Ajouter des règles de protection
- Que risque t-on d'avoir si on n'utilise pas un système de routing ?
  - On ne peut plus rafraîchir 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 changer d'URI, Angular détecte ce changement et essaye de déterminer 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 },
];
```

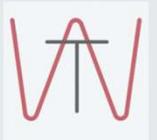
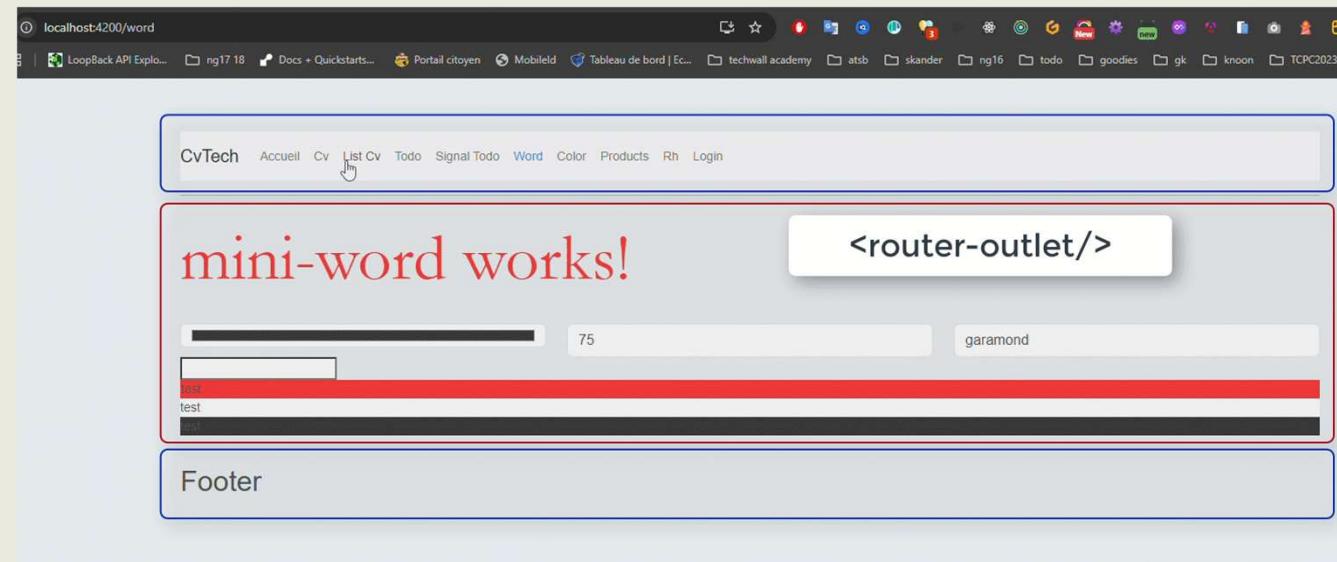
# Syntaxe minimaliste d'une route

- 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 },
];
```

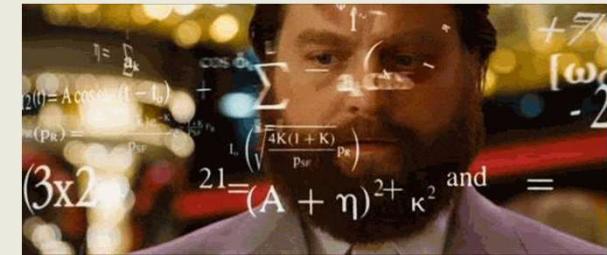
# Préparer l'emplacement d'affichage des vues correspondantes aux routes

- 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 où Angular devra afficher le composant associé à la route.
- Pour indiquer cette partie variable vous devez utiliser la directive **<router-outlet>**.



```
<app-navbar/>
<hr>
<router-outlet/>
<hr>
<h2>Footer</h2>
```

# 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**.

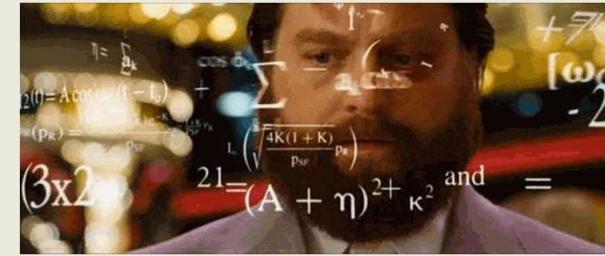
```
<a [routerLink]=[ 'todo' ]>Todo</a>
```

# 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.

The screenshot shows a web browser window with the URL `localhost:4200/cv`. The page displays a list of items under the heading "Cv Tech". The items are:

- Cv Tech
- Color
- CV
- Todo
- aymen sellaouti
- skander sellaouti
- cherche travail worker

The link "cherche travail worker" is highlighted with a red border, indicating it is the active or selected item.

# 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.

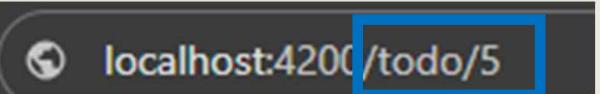
```
import { ActivatedRoute, Router } from "@angular/router";
router = inject(Router);
onLoadCv(id: number) {
    // méthode conseillée
    this.router.navigate(['cv', id]);
    //alternative
    this.router.navigate(['cv/${id}']);
}
```

# 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



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

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

Je cherche une route avec deux segments

- 1- Le premier segment doit être cv
- 2- Le deuxième segment peut contenir n'importe quoi  
c'est une variable

```
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] 2
    routerLinkActive="active" >
    Serveurs</a>
  </li>

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

# Déclencher une route routerLink

## 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});
```

# Récupérer les paramètres d'une route

- Afin de récupérer les paramètres d'une root au niveau d'un composant on doit procéder comme suit :
  1. Importer **ActivatedRoute** qui nous permettra de récupérer les paramètres de la root 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";

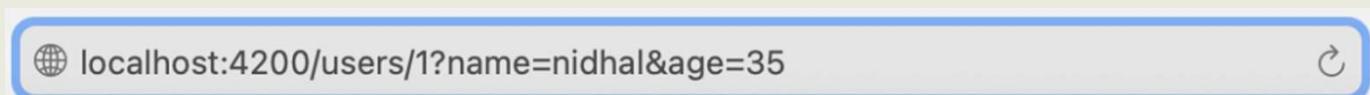
export class DetailsCvComponent {
  acr = inject(ActivatedRoute);
  constructor() {
    console.log({ 'snapshot.params': this.acr.snapshot.params });
  }
}
```

# 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;
  onNavigate() {this.router.navigate(['/about', this.id])}
}
```

# Les queryParameters

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

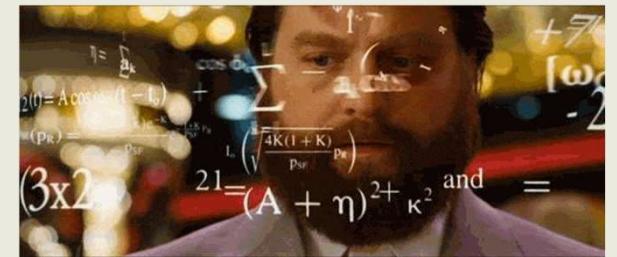
# La route joker

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

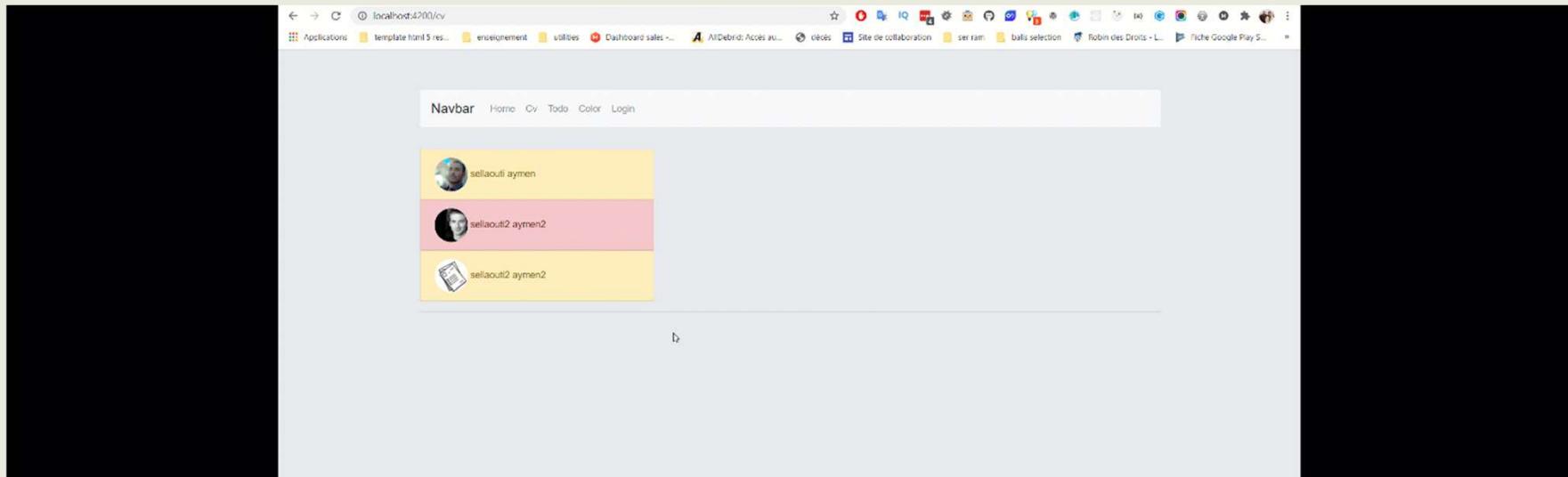
# Exemple

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

# 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.



# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

Aymen sellaouti

# Angular Form

# Approche de gestion de FORM

1. Approche basée Template
2. Approche réactive

# Objectifs

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

# Approche basée Template/ Template Driven Approach

1

Importer le module **FormsModule** dans **app.module.ts**

2

Angular détecte automatiquement un objet form à l'aide de la balise **FORM**. Cependant, il ne détecte aucun des éléments (inputs).

3

Spécifier à Angular quel sont les éléments (contrôles) à gérer.  
- Pour chaque élément ajouter la directive angular **ngModel**.  
- Identifier l'élément avec un nom permettant de le détecter et de l'identifier dans le composant.

4

Associer l'objet représentant le formulaire à une variable et la passer à votre fonction en utilisant le référencement interne # et la directive **ngForm**

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

# Approche basée Template/ Template Driven Approach

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

Template

```
export class TmeplateDrivenComponent {  
  onSubmit(formulaire: NgForm) {  
  
    console.log(formulaire);  
  }  
}
```

Component.ts

# Approche basée Template Validation

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

- required
- email

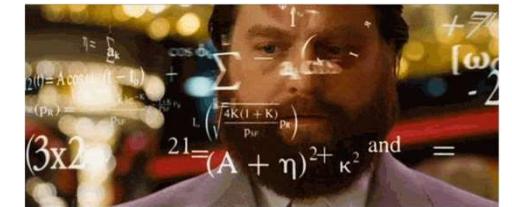
La propriété valid de ngForm permet de vérifier si le formulaire est valid ou non en se basant sur les validateurs qu'ils contiennent.

# Approche basée Template NgForm

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

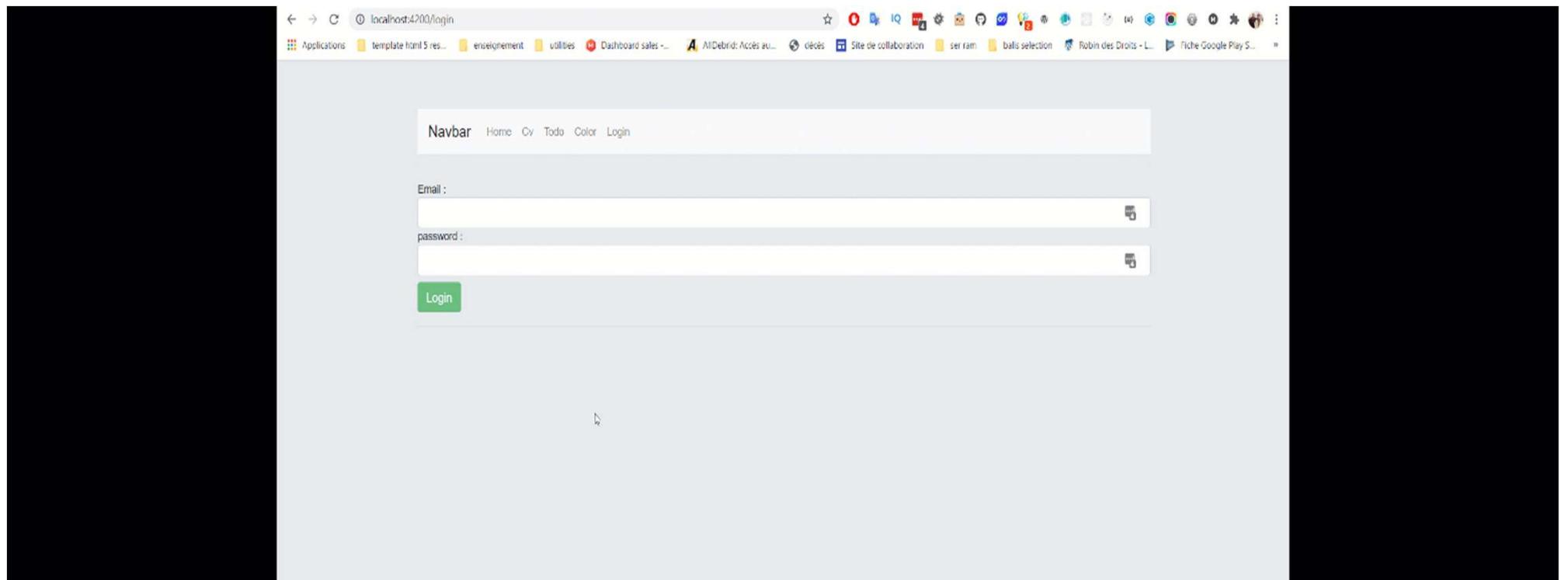
- **dirty** : informe sur le fait que l'une des propriétés du formulaire a été modifié ou non
- **Valid / invalid** : informe si le formulaire est valide ou non
- **Untouched / touched** : informe si le formulaire est touché ou non
- **pristine** : le formulaire n'a pas été touché, c'est l'opposé du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

# Exercice



- Créer un formulaire d'authentification contenant les champs suivants :
  - Email
  - Password
  - Envoyer
- Si un champ est invalide alors il devra avoir une bordure rouge.
- Les deux champs sont obligatoires
- Le password doit avoir au moins 4 caractères.
- Un champ vide et non encore modifié ne peut avoir de bordure rouge que s'il a été touché.
- Le bouton « envoyer » ne doit être cliquable que si le formulaire est valide.
- Utiliser le binding sur la propriété *disabled*.

# Exercice

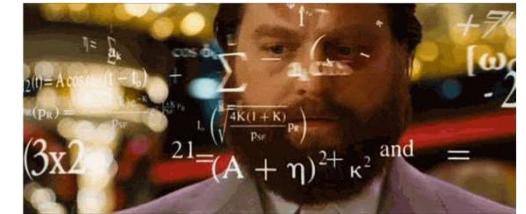


# Approche basée Template

## 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 champ de l'email s'il est invalide. Ce champ ne devra apparaître que si l'utilisateur accède ou modifie le champ email.
- Ajouter un champ d'erreur pour signaler l'erreur à l'utilisateur.

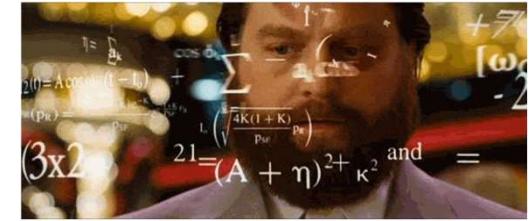
The screenshot shows a web browser window with the URL 'localhost:4200/login' in the address bar. The page title is 'Cv Tech'. Below the title, there is a navigation bar with links for 'Cv Tech', 'Color', 'CV', 'Todo', and 'Login'. The main content area contains a login form with two input fields: 'email:' and 'password:', each with a small icon to its right. Below the fields is a green 'Login' button. The background of the page is white, and the overall layout is clean and modern.

# Approche basée Template

## Associer des valeurs par défaut aux champs

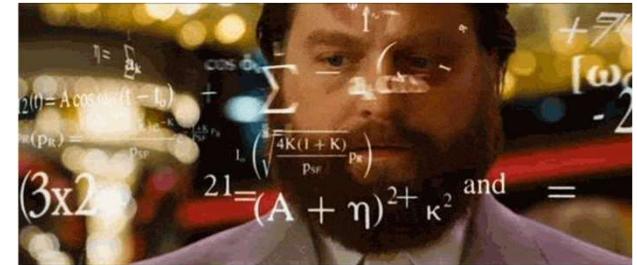
- Pour associer des valeurs par défaut aux champs d'un formulaire associé à Angular il faut le faire à partir du composant.
- Afin de gérer les valeurs du formulaire à partir du composant il faut du binding.
- Au lieu d'avoir juste la primitive ngModel associée au contrôle d'un élément on ajoute le **property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

# Exercice

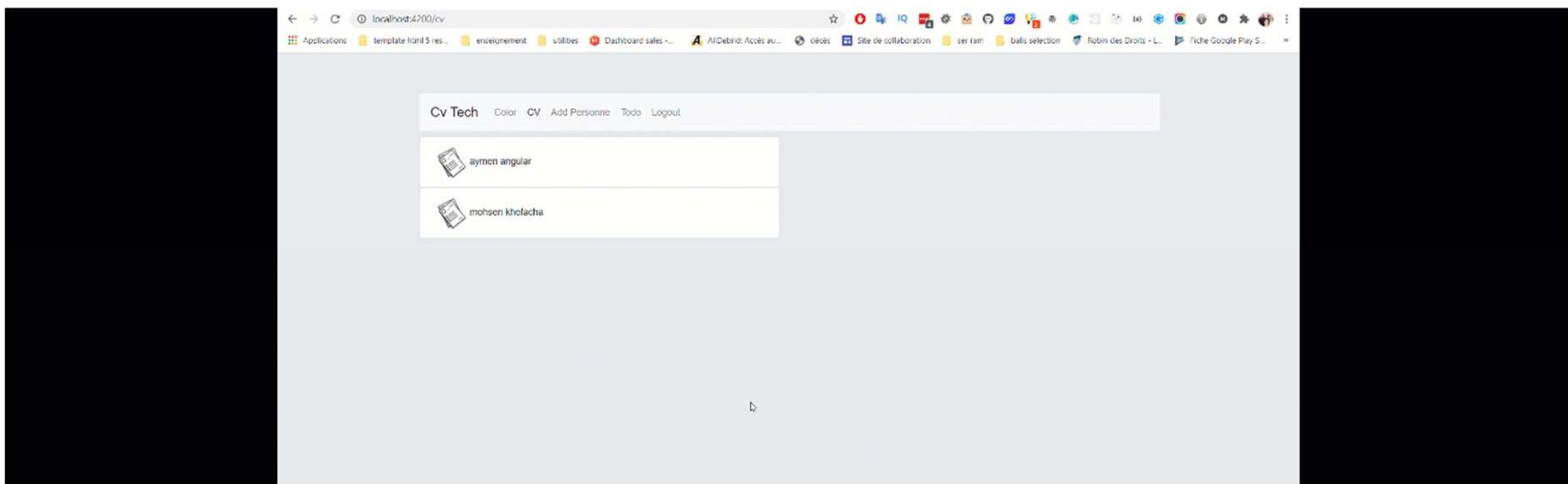


- Ajouter la valeur par défaut « myUserName » au champ username.

# Exercice



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



# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. HTTP
8. Les modules
9. Tests Unitaires et E2E

Aymen sellaouti

# Angular HTTP et Déploiement

# 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

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

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

# Qu'est-ce que la programmation réactive

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

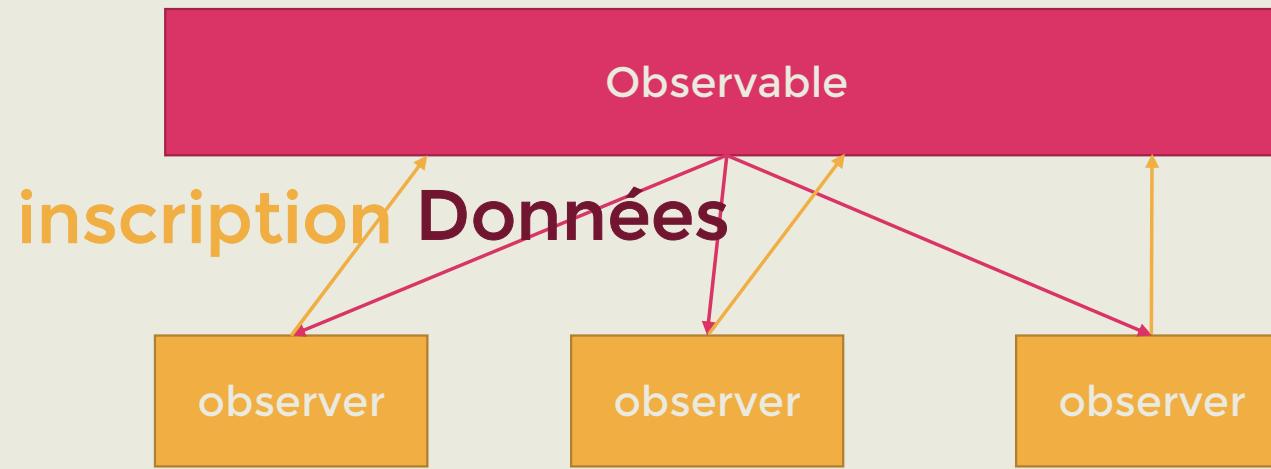
Programmation reactive =

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

# Le pattern « Observer »

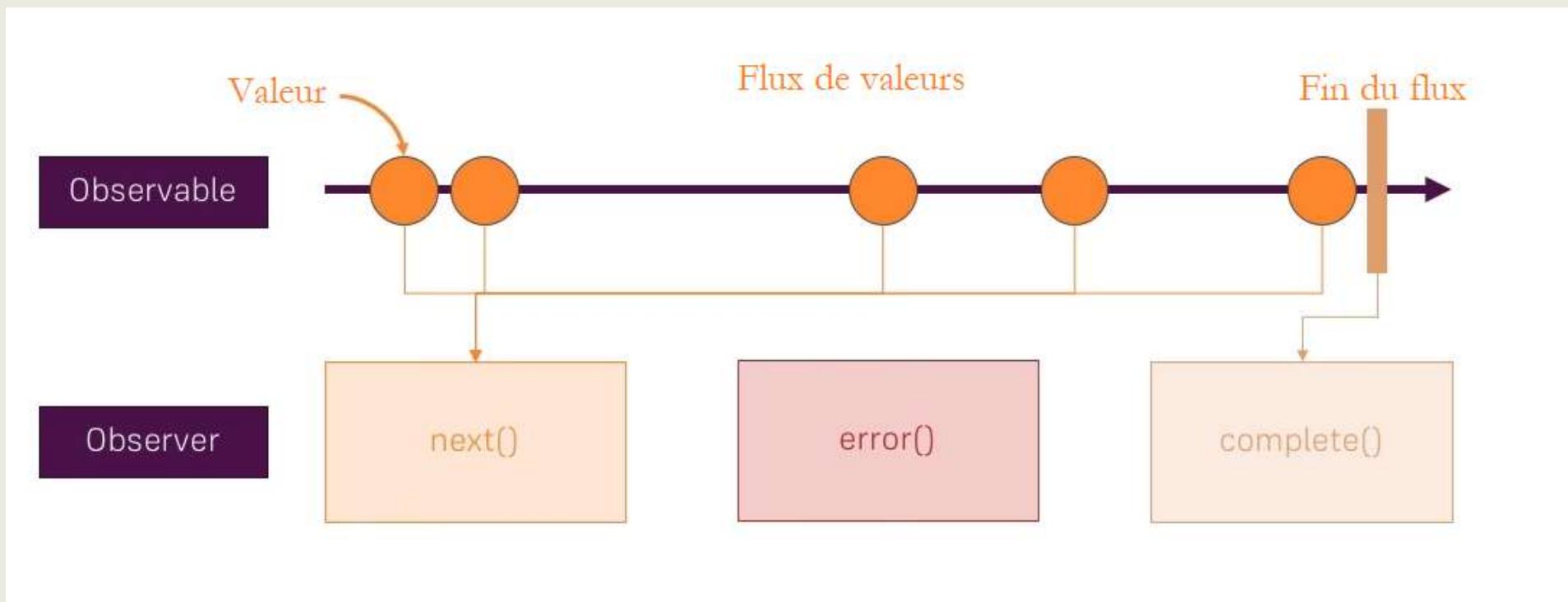
- Le patron de conception **Observateur** permet à un objet de garder la trace d'autres objets, intéressés par l'état de ce dernier.
- Il définit une relation entre objets de type un-à-plusieurs.
- Lorsque l'état de cet objet **change**, il **notifie** ces **observateurs**.

# Observables, Observer et subscriptions



## traitement

# 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 passez 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

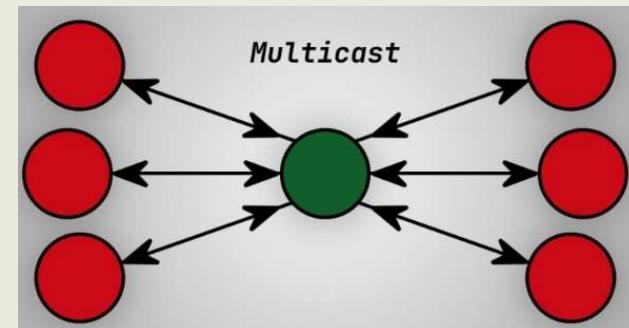
```
myObservable$.subscribe({  
    next: (data) => {  
        //TODO: implementez le comportement quand vous recevez les données  
    },  
    error: (error) => {  
        //TODO: implementez le comportement quand vous avez une erreur  
    },  
    complete: () => {  
        //TODO: implementez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



# Observable

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

# asyncPipe

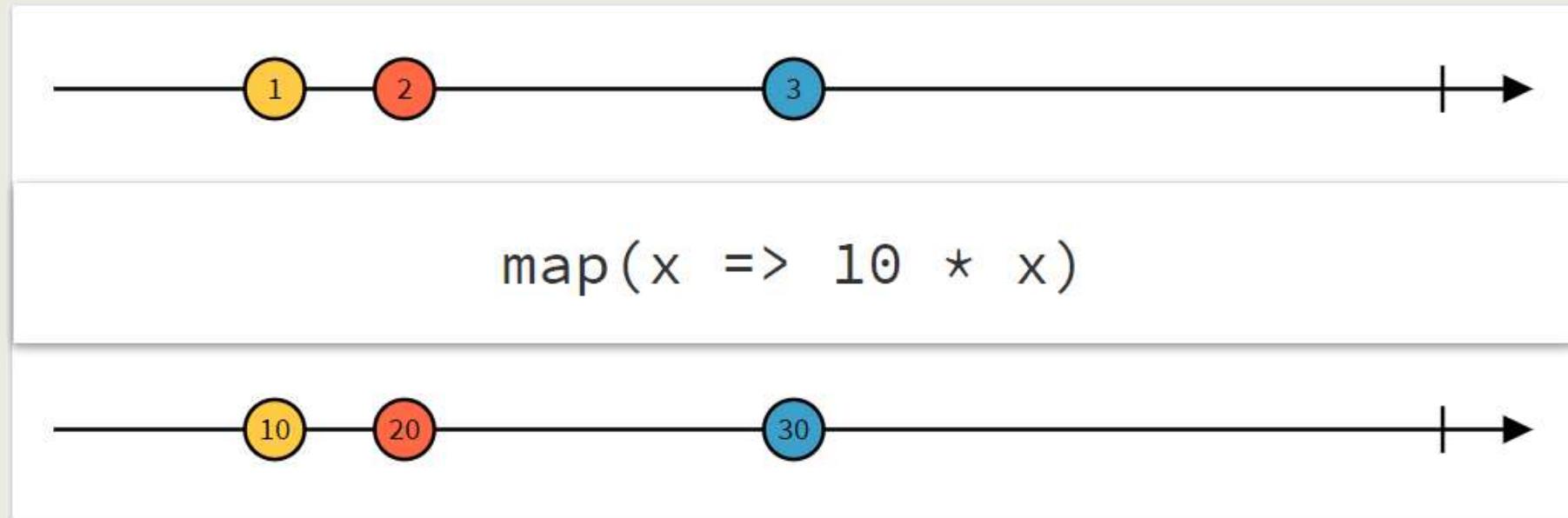
- asyncPipe est un pipe qui permet d'afficher directement un observable.
- {{ valeurSourceAsynchrone | async }}
- L'asyncPipe s'inscrit automatiquement à l'observable et affiche le dernier résultat envoyé.
- Quand le composant est détruit l'asyncPipe se désinscrit automatiquement de l'observable.

# Les opérateurs 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.

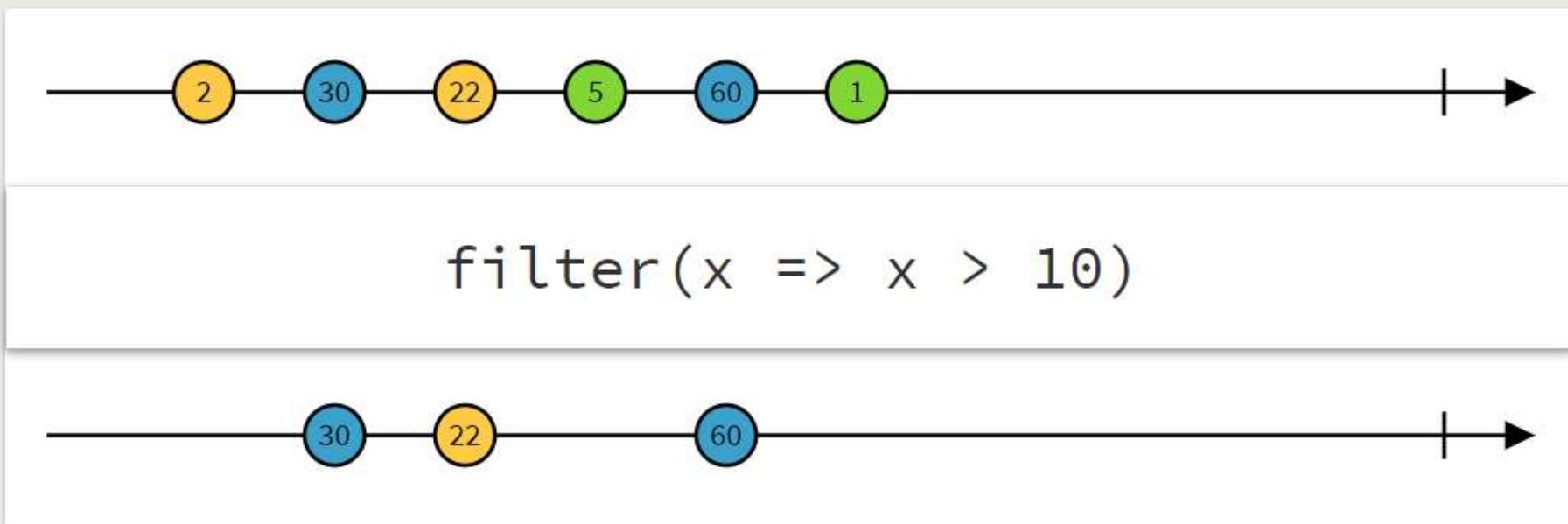
# Quelques opérateurs utiles de l'Observable

## map

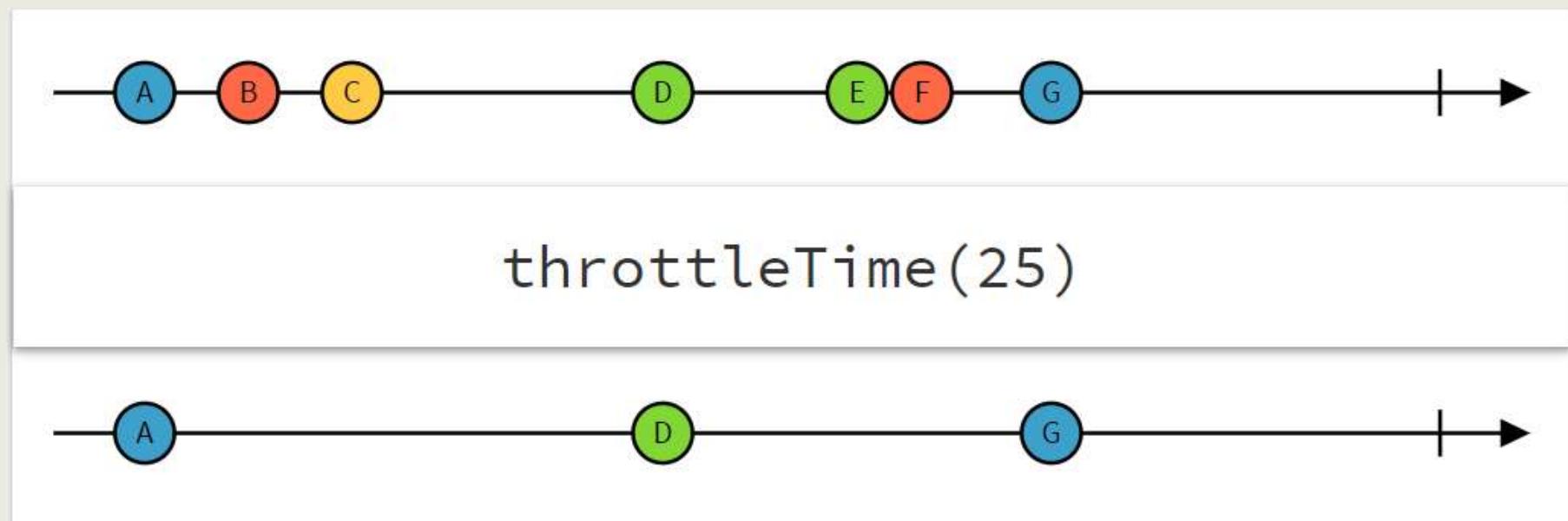


# Quelques opérateurs utiles de l'Observable

## filter



# Quelques opérateurs utiles de l'Observable



# Quelques opérateurs utiles de l'Observable

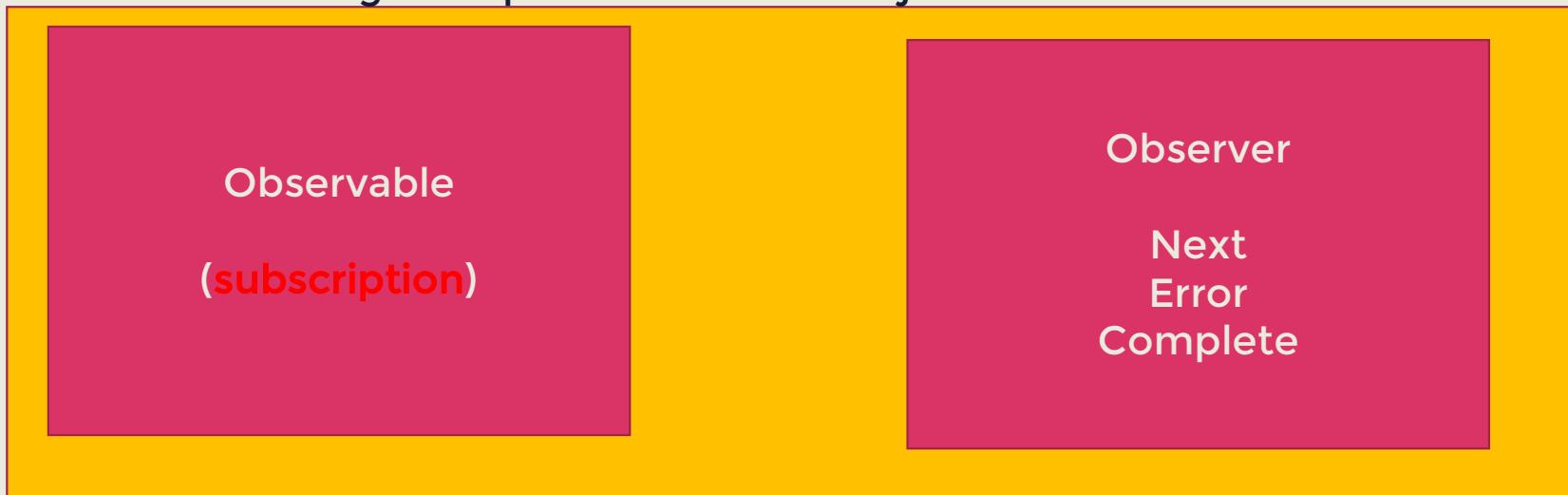
<https://angular.io/guide/rx-library>

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

<http://rxmarbles.com/>

# Les subjects

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



# Les subjects

- 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

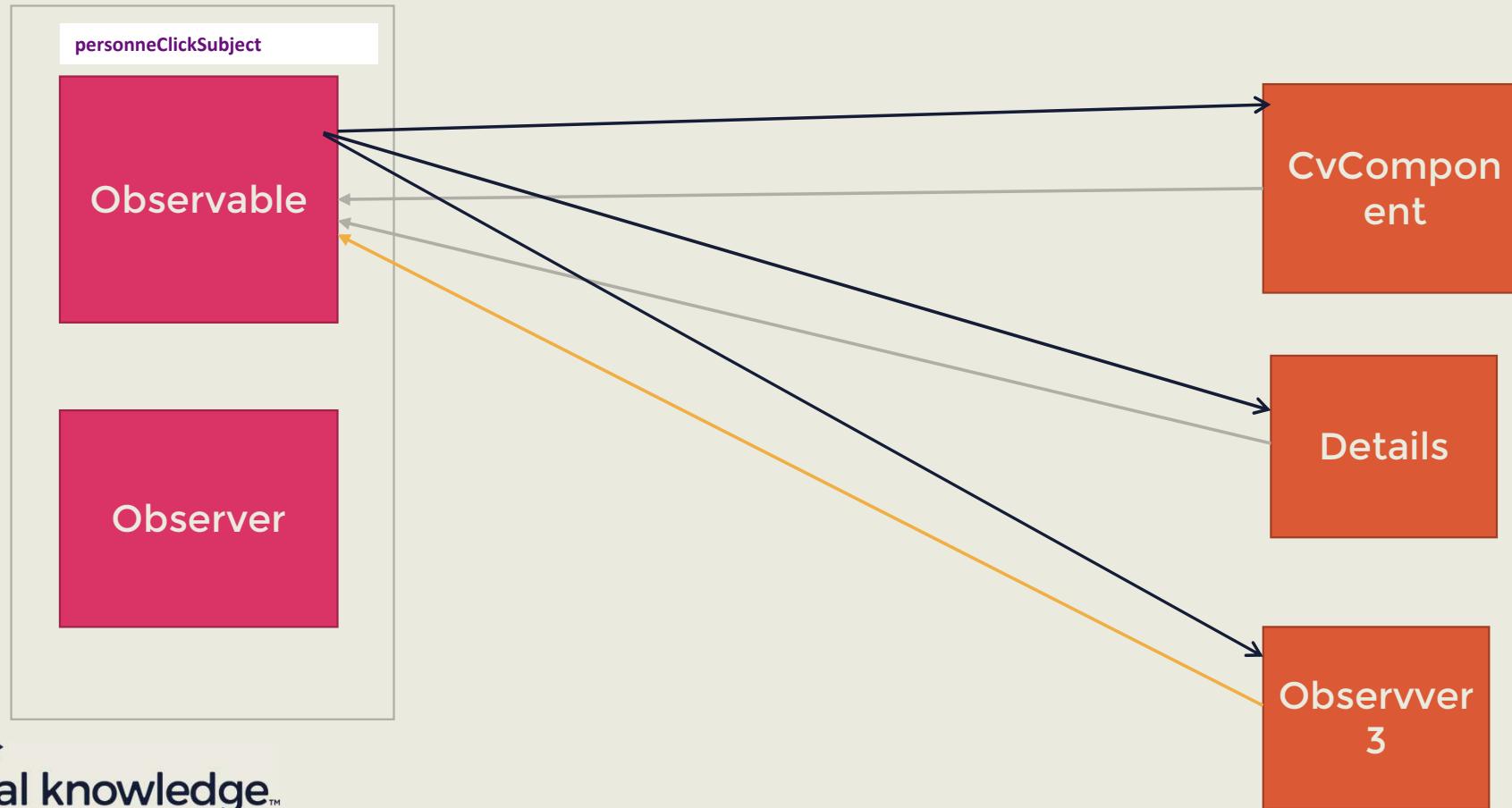
2

# Les subjects

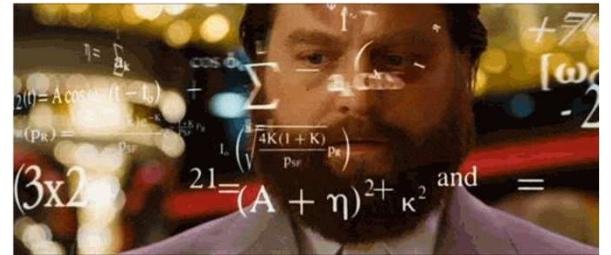
- Si vous êtes intéressé par ce flux, c'est un flux comme un autre, il vous suffit donc de vous inscrire.

```
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)
- Il faudra aussi l'ajouter dans le fichier `module.ts` dans le tableau d'imports.

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

```
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 **Observale**.
- Cet observable a 3 callback function comme paramètres.
  - Une en cas de réponse
  - Une en cas d'erreur
  - La troisième en cas de fin du flux de réponse.

# Interagir avec une API Get Request

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

# Interagir avec une API POST Request

- Afin d'exécuter une requête POST le module http nous offre une méthode post.
- Cette méthode retourne un Observale.
- Diffère de la méthode get avec un attribut supplémentaire : body
- Cette observable a 3 callbacks fonction comme paramètres.
  - Une en cas de réponse
  - Une en cas d'erreur
  - La troisième en cas de fin du flux de réponse.

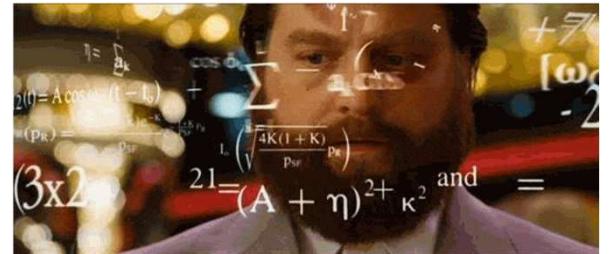
# Interagir avec une API POST Request

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

## Documentation

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

# Exercice



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

# Les headers

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

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

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

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

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

# Les paramètres

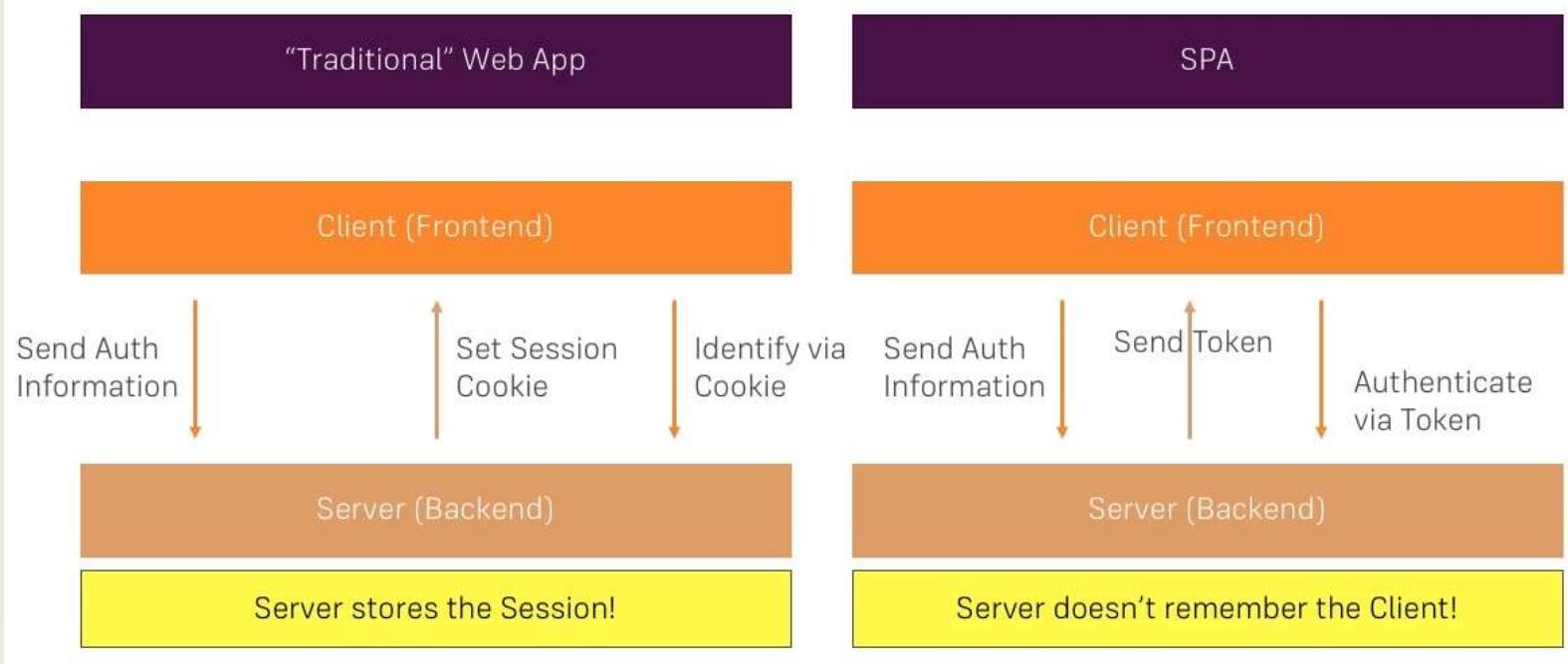
- Afin d'ajouter des paramètres à vos requêtes, le HttpClient vous offre la classe HttpParams.
- Cette classe est une classe immutable (read Only).
- Elle propose une panoplie de méthode helpers permettant de la manipuler.
- `set(clé,valeur)` permet d'ajouter des headers. Elle écrase les anciennes valeurs.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

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

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

# Authentification

## How does Authentication work?



# Ajouter le token dans la requête

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

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

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

# Ajouter le token dans la requête

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

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

# Sécuriser vos routes

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

# Guard

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

# Guard / canActivate

- Afin d'utiliser le guard **canActivate** (de même pour les autres), vous devez créer un classe qui implémente l'interface **CanActivate** et donc qui doit implémenter la méthode **canActivate** de sorte qu'elle retourne un booléen permettant ainsi l'accès ou non à la route cible. 1
- Vous devez ensuite ajouter cette classe dans le provider. 2
- Finalement pour l'appliquer à une route, ajouter la dans la propriété **canActivate**. Cette propriété prend un tableau de guard. Elle ne laissera l'accès à la route qu'esi la totalité des guard retourne true. 3
- 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 le futur état du routeur de l'application qui devra passer la validation du guard
  // https://vsavkin.com/routeur-angular-comprendre-1%C3%A9tat-du-routeur-5e15e729a6df
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    if (// your condition) {
      return true;
    }
    return false;
  }
}
```

# Guard / canActivate

2

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

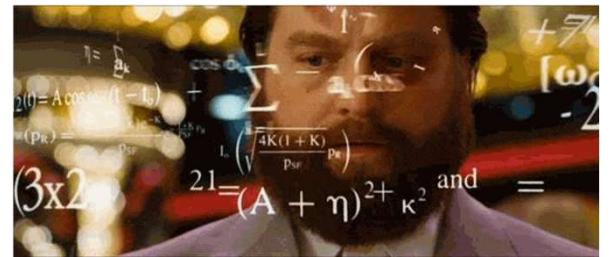
App.module.ts

# Guard / canActivate

3

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

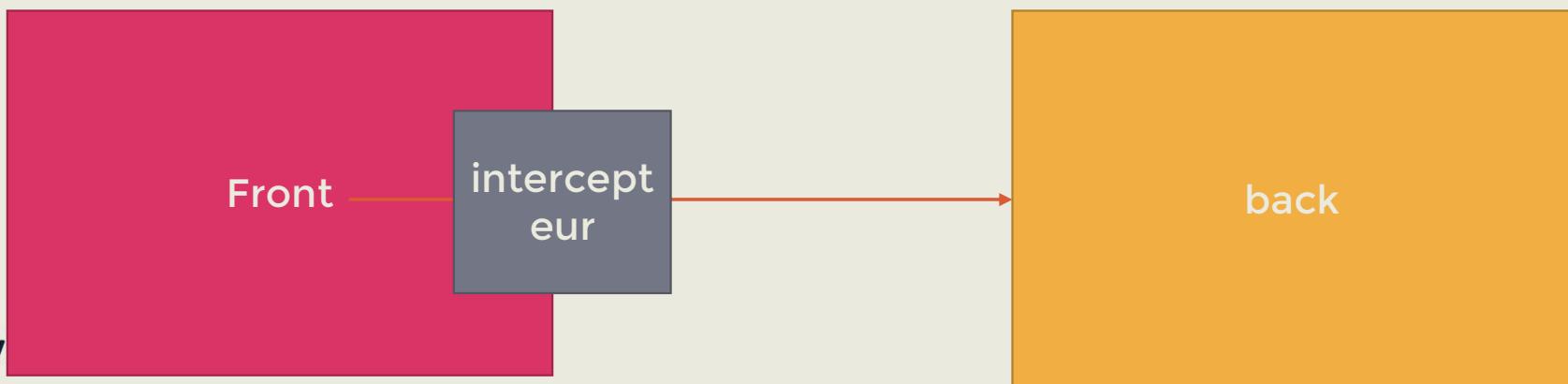
# Exercice



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

# Les intercepteurs

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



# Les intercepteurs

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

# Les intercepteurs

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

```
export const  
AuthentificationInterceptorProvider = {  
  provide: HTTP_INTERCEPTORS,  
  useClass: AuthentificationInterceptor,  
  multi: true,  
};
```

```
providers: [  
  AuthentificationInterceptorProvider  
],
```

# Les intercepteurs : changer la requête

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

```
export const  
AuthentificationInterceptorProvider = {  
  provide: HTTP_INTERCEPTORS,  
  useClass: AuthentificationInterceptor,  
  multi: true,  
};
```

```
providers: [  
  AuthentificationInterceptorProvider  
],
```

# Cloner une requête

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

# Intercepter les erreurs

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

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

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

# Déploiement

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

`ng build --prod`

- Un dossier dist sera créer 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`

# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. Angular: Reactive Form
8. HTTP
9. Les modules
10. Tests Unitaires et E2E

Aymen sellaouti

# Angular Reactive Form

# Reactive form

- Les Reactive Form sont une deuxième méthode de gérer vos formulaires avec Angular.
- Au contraire des Template Driven Form, ses formulaires sont **générés programmatiquement** dans la partie TS.
- Ceci permet **d'alléger le template** des validateurs.
- D'autre part ça permet d'avoir un **formulaire testable**
- Finalement ceci permet de plus facilement **générer des Validateurs personnalisés**.

# Reactive form

## Créer un formulaire

- Commencer par ajouter le module **ReactiveFormsModule**.
- Afin de créer un formulaire avec l'approche réactive, vous devez créer un objet **FormGroup**.
- Un **FormGroup** prend en paramètre un objet décrivant le formulaire. Chaque champ de l'objet a comme **première propriété le nom et comme valeur un objet définissant les champs associés à ce formulaire**. Ce sont les **FormControl**.
- Chaque **FormControl** définit un **champ du formulaire**. Il prend en paramètre, **la valeur initiale, un Validator ou un tableau de Validator et en troisième paramètre, un AsyncValidator ou un tableau d'AsyncValidator**

# Reactive form

## Créer un formulaire

```
FormGroup.constructor(  
  controls: {[p: string]: AbstractControl},  
  validatorOrOpts?: ValidatorFn | ValidatorFn[] | AbstractControlOptions | null,  
  asyncValidator?: AsyncValidatorFn | AsyncValidatorFn[] | null)
```

```
ngOnInit() {  
  this.form = new FormGroup({  
    name: new FormControl(null),  
    firstname: new FormControl(null),  
    age: new FormControl(null),  
  });
```

# Reactive form

## Créer un formulaire

```
constructor(controls: TControl, validatorOrOpts?: ValidatorFn | ValidatorFn[] | AbstractControlOptions | null, asyncValidator?: AsyncValidatorFn | AsyncValidatorFn[] | null);  
  controls: eTypedOrUntyped<TControl, TControl, {  
    [key: string]: AbstractControl<any>;  
  }>;
```

```
export declare interface AbstractControlOptions {  
  validators?: ValidatorFn | ValidatorFn[] | null;  
  asyncValidators?: AsyncValidatorFn | AsyncValidatorFn[] | null;  
  /**  
   * @description  
   * The t name for control to update upon.  
   */  
  updateOn?: 'change' | 'blur' | 'submit';  
}
```

# Reactive form

## Créer un formulaire

```
form = new FormGroup({  
    name: new FormControl(null),  
    firstname: new FormControl("Aymen"),  
    age: new FormControl(0, {  
        nonNullable: true,  
        validators: Validators.required,  
        updateOn: "blur",  
    }),  
});
```

# Reactive form

## Créer un formulaire

- Vous pouvez aussi passer par le **service FormBuilder** et sa méthode **group**.
- Elle prend en **paramètre un objet** avec comme **clé** le nom du **formControl** et comme **valeur** un **tableau** avec comme **première propriété la valeur initiale du champ**.
- Ceci est **moins verbeux** et plus simple à gérer pour les grands formulaires.

```
this.form = this.formBuilder.group({  
  email: [null],  
  username: [null],  
  userCategory: ['employee'],  
});
```

```
constructor(  
  private formBuilder: FormBuilder  
) {}
```

# Reactive form

## Associer le FormGroup à votre form

- Une fois le formulaire défini, nous devons l'associer au form de votre template.
  - Pour ce faire, vous devez ajouter la directive **formGroup** (qui se trouve dans le module **ReactiveFormsModule**) au niveau de la balise form et la binder à votre objet de type **FormGroup** au niveau de votre fichier TS.

```
reactive.component.ts x reactive.component.html x
7 styleUrls: ['./style.css']
8 })
9 export class Reactive
10
11 constructor(private
12 form: FormGroup;
```

```
<h3 class="text-center">User Details</h3>
1 <hr>
2 <form [FormGroup]="form">
3 <div>
4 <label for="email">Email</label>
5 <input class="form-control" type="email"
```

# Reactive form

## Associer les FormControl à vos input

- Afin d'associer votre FormControl à votre champ input, utiliser la directive **formControlName** et associer le à l'**identifiant** du **FormControl**

```
  7 ↑ ↓ 4   <div>
  8   <label for="email">Email</label>
  9   <input
 10     class="form-control" type="email" id="email"
 11     formControlName="email" placeholder="Enter email"/>
 12 </div>
 13 <div>
 14   <label for="username">Username</label>
 15   <input
    class="form-control" type="text" id="username"
    formControlName="username" placeholder="Username"/>
 16 </div>
```

skills  
glob

284

# Reactive form

## Récupérer les FormControl dans le HTML

- Afin de récupérer les FormControl dans le HTML, utiliser la méthode `get` de votre `form group` et passer lui l'identifiant du FormControl à récupérer.

```
<button  
    class="btn btn-primary"  
    [disabled]="form.get('password').valid"  
    (click)="process()>  
    Submit  
</button>
```

## Reactive form

### Récupérer les erreurs du FormControl dans le HTML

- Afin de récupérer les erreurs de votre control dans le HTML, utiliser l'attribut errors.

```
<div  
  *ngIf="form.get('name')?.errors && form.get('name')?.touched"  
  class="alert alert-danger"  
>
```

# Reactive form

## Récupérer les FormControl dans le HTML

- Une deuxième méthode pour avoir un code moins verbeux est de créer des getters pour vos champs.

```
get name(): AbstractControl {  
    return this.form.get("name")!;  
}
```

```
<div *ngIf="name.errors && name.touched">  
    Ce champ est obligatoire  
</div>
```

# Reactive form

## Soumettre le formulaire

- A l'inverse de l'approche 'template driven', vous n'avez pas besoin de récupérer la référence de l'objet form puisque vous l'avez déjà créée dans votre ts.
- Il suffit donc d'écouter le submit avec ngSubmit ou le clic sur un bouton pour déclencher la méthode du composant qui générera l'envoi du formulaire.

```
process() {  
    console.log(this.form);  
}  
  
34   <button  
35     class="btn btn-primary"  
36     (click)="process()"  
37     Submit  
38   </button>
```

# Reactive form

## Les Validateurs

- Un validateur est une fonction qui retourne **false si un champ est valide** selon une certaine condition **sinon elle retourne une information sur l'erreur.**
- Afin de valider votre formulaire, passer en deuxième paramètre de **FormControl** une **référence à une méthode de la classe Validators** ou un tableau de ces méthodes.

```
FormControl(formState: null, validatorOrOpts: [Validators.]),  
  ↪email(control: AbstractControl)  
  ↪required(control: AbstractControl)  
  ↳compose(validators: null)  
  ↪nullValidator(control: AbstractControl)  
  ↪max(max: number)  
  ↪composeAsync(validators: (AsyncValidatorFn | null)[])  
  ↪maxLength(maxLength: number)  
  ↪min(min: number)  
  ↪minLength(minLength: number)  
  ↪pattern(pattern: string | RegExp)
```

# Reactive form

## Les Validateurs

- Afin de valider votre formulaire, passer en deuxième paramètre de **FormControl** une **référence à une méthode de la classe Validators** ou un **tableau de ces méthodes**.

```
FormControl(formState: null, validatorOrOpts: [Validators.]),
  ↪email(control: AbstractControl)
  ↪required(control: AbstractControl)
  ↳compose(validators: null)
  ↪nullValidator(control: AbstractControl)
  ↪max(max: number)
  ↪composeAsync(validators: (AsyncValidatorFn | null)[])
  ↪maxLength(maxLength: number)
  ↪min(min: number)
  ↪minLength(minLength: number)
  ↪pattern(pattern: string | RegExp)
```

# Reactive form

## Les Validateurs

- **Validateurs synchrones** : Ce sont des fonctions qui contrôle votre élément et retournent un **ensemble d'erreurs de validation ou null**. C'est ce qu'on passe en deuxième paramètre lors de l'instanciation d'un FormControl.
- **Validateurs asynchrones** : Ce sont des fonctions asynchrones qui contrôle votre élément et retournent une **Promise ou un Observable** qui émettent un **ensemble d'erreurs de validation ou null**. C'est ce qu'on passe en troisième paramètre lors de l'instanciation d'un FormControl.
- Pour des raisons de **performance**, Angular commence par les validateurs synchrones, s'ils passent il déclenche les validateurs asynchrones.

# Reactive form

## Les Validateurs offerts par Angular

- Vous pouvez utiliser des validateurs offerts par angular ou créer vos propres validateurs.
- Les validateurs de base sont les mêmes que ceux de l'approche basée Template.

```
new FormGroup({  
    email: new FormControl(null, [Validators.required, Validators.email]),  
    username: new FormControl(null, [Validators.minLength(3)]),  
    age: new FormControl(null, [  
        Validators.required, Validators.pattern('[0-1]?\\d{1,2}')  
    ]),  
});
```

Méthode 1

# Reactive form

## Les Validateurs offerts par Angular

```
class Validators {  
    static min(min: number): ValidatorFn  
    static max(max: number): ValidatorFn  
    static required(control: AbstractControl<any, any>): ValidationErrors | null  
    static requiredTrue(control: AbstractControl<any, any>): ValidationErrors | null  
    static email(control: AbstractControl<any, any>): ValidationErrors | null  
    static minLength(minLength: number): ValidatorFn  
    static maxLength(maxLength: number): ValidatorFn  
    static pattern(pattern: string | RegExp): ValidatorFn  
    static nullValidator(control: AbstractControl<any, any>): ValidationErrors | null  
    static compose(validators: ValidatorFn[]): ValidatorFn | null  
    static composeAsync(validators: AsyncValidatorFn[]): AsyncValidatorFn | null  
}
```

# Reactive form

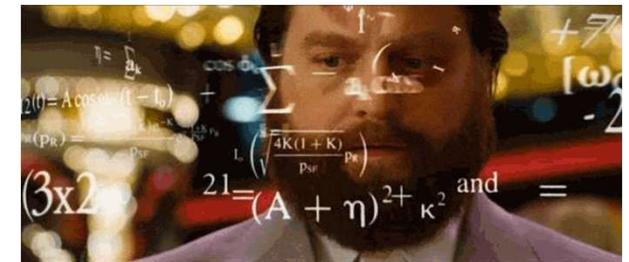
## Les Validateurs offerts par Angular

- Vous pouvez utiliser des validateurs offerts par angular ou créer vos propres validateurs.
- Les validateurs de base sont les mêmes que ceux de l'approche basée Template.

```
formGroup: FormGroup = new FormGroup({  
    name: new FormControl(null, {  
        validators: [Validators.required, Validators.minLength(3)],  
        asyncValidators: [],  
        updateOn: "change"  
    }),  
    age: new FormControl(null),  
});
```

Méthode 2

# Exercice



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

# Reactive form

## Manipulez les valeurs de votre form

- Afin de **mettre à jour la valeur** d'un **FormControl ou d'un FormGroup**, vous pouvez utiliser la méthode **setValue()** qui met à jour la valeur du contrôle de formulaire et valide la structure de la valeur fournie par rapport à la structure du contrôle.
- Vous pouvez utiliser la méthode **patchValue** si vous modifiez uniquement une partie.

```
this.form.setValue({ name: "Sellaouti", firstname: "Aymen" });
```

```
this.form.patchValue({firstname: "Aymen" });
```

# Reactive form

## Manipulez les valeurs de votre form

- En deuxième paramètre de ces deux fonctions, vous pouvez passer un objet d'options avec deux propriétés:
  - **onlySelf**: Angular vérifie l'état de validation du formulaire, chaque fois qu'il y a un changement de valeur. La **validation commence à partir du contrôle** dont la valeur a été modifiée et **se propage au FormGroup** de niveau supérieur. Il s'agit du comportement par défaut. Si vous ne souhaitez pas **qu'Angular vérifie la validité de l'ensemble du formulaire, chaque fois** que vous modifiez la valeur à l'aide de `setValue` ou `patchValue`, définissez **onlySelf à true**.

# Reactive form

## Manipulez les valeurs de votre form

- En deuxième paramètre de ces deux fonctions, vous pouvez passer un objet d'options avec deux propriétés:
  - **emitt**: Les forms Angular émettent deux événements. L'un est **ValueChanges** et l'autre est **StatusChanges**. L'événement ValueChanges est émis chaque fois que la valeur du formulaire est modifiée. L'événement StatusChanges est émis chaque fois qu'angular calcule l'état de validation du formulaire. C'est le comportement par défaut Nous pouvons empêcher que cela se produise, en définissant l'emitt à false

# Reactive form

## Manipulez les valeurs de votre form

```
this.form.patchValue(  
  { firstname: "Aymen" },  
  {  
    emit: false,  
    onlySelf: true,  
  }  
)
```

## Reactive form

# Suivre les modifications de vos FormControls et de vos FormGroup

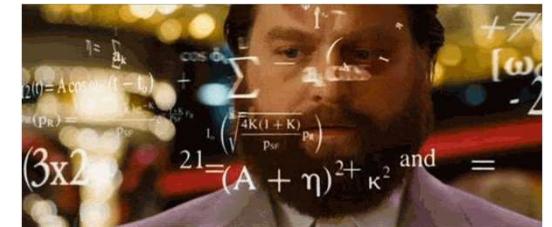
- **FormControl** et **FormGroup**, vous fournissent deux Observables permettant le suivi des changements de valeur et de statut.
- **ValueChanges** est un événement déclenché par les formulaires Angular chaque fois que la valeur de FormControl, FormGroup ou FormArray change. L'observable obtient la dernière valeur du contrôle. Il nous permet de suivre les modifications apportées à la valeur en temps réel et d'y répondre. Par exemple, nous pouvons l'utiliser pour valider la valeur, calculer les champs calculés, ...

## Reactive form

# Suivre les modifications de vos FormControl et de vos FormGroup

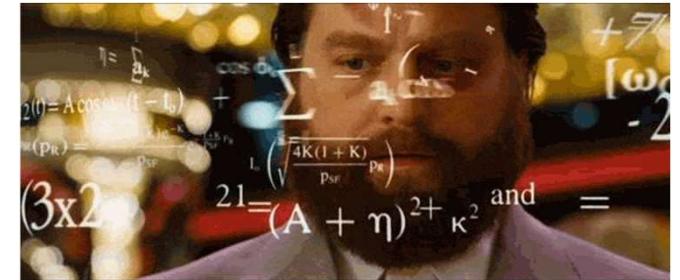
- `statusChanges` est un événement déclenché par les formulaires Angular **chaque fois que Angular calcule le statut de validation** de FormControl, FormGroup ou FormArray. Il renvoie un observable afin que vous puissiez vous y abonner. L'observable obtient le dernier état du contrôle.

# Exercice



- Afin de protéger les informations personnelles des mineurs, faites-en sortes que lorsque l'âge de la personne possédant le Cv est inférieur à 18 ans, il ne puisse pas renseigner le path de l'image.

# Exercice



- Etant donné que notre formulaire est assez volumineux et pour permettre une meilleure expérience utilisateur, nous voulons faire en sorte que si l'utilisateur saisisse un formulaire valide et qu'il sorte de la page sans l'envoyer, il puisse le retrouver rempli lorsqu'il retourne à la page d'ajout d'un cv.

# Reactive form

## Customiser vos validateurs

- Un custom validator est une fonction de type `ValidatorFn` qui prend en paramètre un control de type `AbstractControl` (qui contient une propriété `value` représentant la `valeur du champ à valider`) et retourne `null` si la valeur `est valide` ou un objet de type `ValidationErrors` s'il y a des `erreurs de validation`.

```
export declare interface ValidatorFn {  
  (control: AbstractControl): ValidationErrors | null;  
}
```

```
export declare type ValidationErrors = {  
  [key: string]: any;  
};
```

# Reactive form

## Customiser vos validateurs

- Le Validator, renvoyée par la fonction de création de validateur, doit suivre ces règles :
- Un seul argument d'entrée est attendu, qui est de type `AbstractControl`. La fonction validateur peut obtenir la valeur à valider via la propriété `control.value`
- La fonction de validation doit renvoyer null si aucune erreur n'a été trouvée dans la valeur du champ, ce qui signifie que la valeur est valide
- Si des erreurs de validation sont trouvées, la fonction doit renvoyer un objet de type `ValidationErrors`.

# Reactive form

## Customiser vos validateurs

- L'objet **ValidationErrors** peut avoir comme propriétés **les multiples erreurs trouvées** (généralement une seule) et comme valeurs les détails de chaque erreur.
- Si nous voulons simplement indiquer qu'une erreur s'est produite, **sans fournir plus de détails**, nous pouvons simplement **renvoyer true** comme valeur d'une propriété error dans l'objet **ValidationErrors**.

```
return !passwordValid ? {passwordStrength: true} : null;
```

# Reactive form

## Customiser vos validateurs

```
export function createPasswordStrengthValidator(): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const value = control.value;
    if (!value) {
      return null;
    }
    const hasUpperCase = /[A-Z]+/.test(value);
    const hasLowerCase = /[a-z]+/.test(value);
    const hasNumeric = /[0-9]+/.test(value);
    const passwordValid = hasUpperCase && hasLowerCase && hasNumeric;
    return !passwordValid ? {passwordStrength: true} : null;
  };
}
```

# Reactive form

## Customiser vos validateurs

### Validateurs Asynchrones

- Dans certains cas d'utilisation, la validation de votre champ ne se fait pas d'une façon **asynchrone**.
- Le cas le plus répandu est la **validation par votre backend**.
- Imaginez que vous avez un champ email et qu'il ne doit pas être redondant. La solution est d'avoir une API qui vérifie ça et qui vous retourne la réponse.
- Ce traitement étant Asynchrone, le Validateur synchrone ne fait plus affaire.
- Il faut donc créer un validateur **ASYNCHRONE**.

# Reactive form

## Customiser vos validateurs

### Validateurs Asynchrones

- Le Validator, renvoyée par la fonction de création de validateur, doit suivre ces règles :
- Un seul argument d'entrée est attendu, qui est de type `AbstractControl`. La fonction validateur peut obtenir la valeur à valider via la propriété `control.value`
- La fonction de validation doit renvoyer une `Promise<null>`, un `Observable<null>` si aucune erreur n'a été trouvée dans la valeur du champ, ce qui signifie que la valeur est valide
- Si des erreurs de validation sont trouvées, la fonction doit renvoyer une `Promise` ou un `Observable` d'un objet de type `ValidationErrors`.

# Reactive form

## Customiser vos validateurs

### Validateurs Asynchrones

```
export function userExistsValidator(authService: AuthService): AsyncValidatorFn {  
  return (control: AbstractControl) => {  
    return authService.findUserByEmail(control.value);  
  };  
}
```

# Reactive form

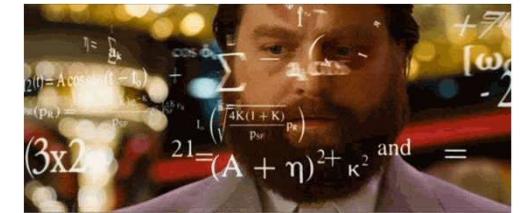
## Customiser vos validateurs

### Validateurs Asynchrones

```
findUserByEmail(value: any): Observable<ValidationErrors | null> {
    return new Observable<ValidationErrors | null>(
        (observer) => {
            setTimeout(
                () => {
                    const date = new Date();
                    if (date.getTime() % 2) {
                        observer.next(null);
                    } else {
                        observer.next({userExists: true});
                    }
                }, 1500);
        }
    );
}
```

# Exercice

- Le champ Cin étant unique, créer un validateur asynchrone permettant de gérer cette contrainte et tester le.



# Reactive form

## Customiser vos validateurs

### Valider un form

- Afin de valider un form, vous devez faire la même chose que pour le validateur d'un FormControl.

```
export function createDateRangeValidator(): ValidatorFn {
  return (form: AbstractControl): ValidationErrors | null => {
    const start: Date = form.get('startAt').value;
    const end: Date = form.get('endAt').value;
    if (start && end) {
      const isRangeValid = (end.getTime() - start.getTime() > 0);
      return isRangeValid ? null : {dateRange: true};
    }
    return null;
  };
}
```

# Reactive form

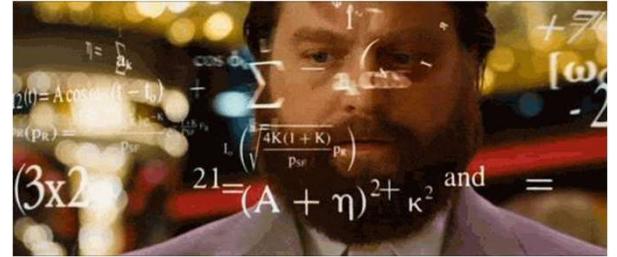
## Customiser vos validateurs

### Valider un form

- Maintenant et pour l'appliquer à votre FormGroup ajouter à la création de votre FormGroup un second paramètre qui est un objet options et utiliser la propriété validators

```
this.form = new FormGroup(  
  {},  
  {  
    validators: VotreValidateur  
  }  
);  
this.form = this.formBuilder.group(  
  {},  
  {  
    validators: VotreValidateur  
  }  
);
```

# Exercice



- Le champ Cin étant composé de 8 caractères numériques, il présente une corrélation entre l'age de la personne et les deux premiers caractères.
- Si la personne a un age  $\geq 60$  ans, les deux premiers caractères numériques doivent être entre 00 et 19.
- Sinon ça doit être supérieur à 19.
- Créer le validateur permettant de faire ça.

# Plan du Cours

1. Introduction à NodeJS
2. Express avec NestJs
3. Introduction à Angular
4. Les composants
5. Les directives
- 5Bis. Les pipes
4. Service et injection de dépendances
5. Le routage
6. Form
7. Angular: Reactive Form
8. HTTP
9. Les modules
10. Tests Unitaires et E2E

Aymen sellaouti

# Angular Les modules

# 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.  
Cependant dès que votre application grandit penser à la séparer en Modules.
- Chaque module **vie séparée** des autres modules. Par défaut il **n'expose rien**,  
tous 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 tous 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 {}
```

# Déclarations

- Dans la partie **declarations**, faites en sorte que chaque composant, directive ou pipe soit associé à **un et un seul module**. **Ne déclarer pas un même composant dans deux modules différents.**
- Ne déclarer que les composants, directives et les pipes dans cette partie.
- Tous les composants, directives et les pipes déclarés sont **privés par défaut**. Ils ne sont accessibles que pour les composants, directives et les pipes déclarés dans le même module.
- Pour utiliser un de ces éléments 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 { TodoRoutingModule } from "./todo.routing";

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

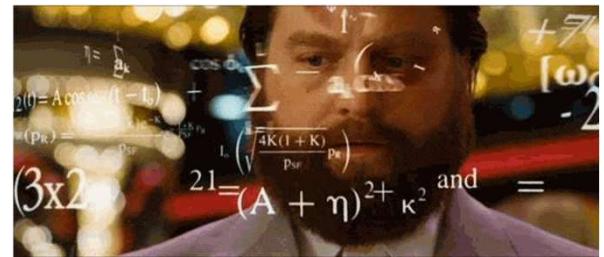
```
import { NgModule } from "@angular/core";
import { Route, RouterModule } from "@angular/router";
import { NF404Component } from "../nf404/nf404.component";
import { TodoComponent } from "./todo.component";

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

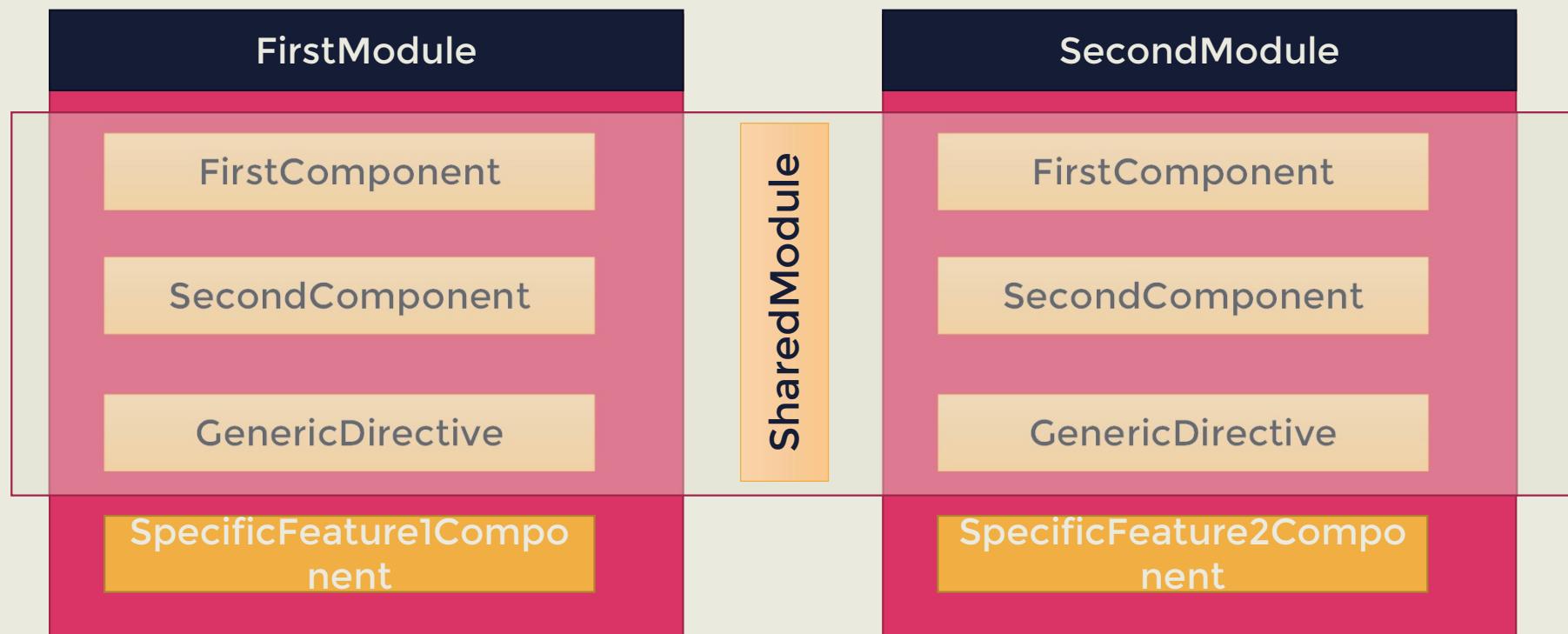
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
export class TodoRoutingModule {}
```

# Exercice

- Implémentez le CvModule.



# Shared Module

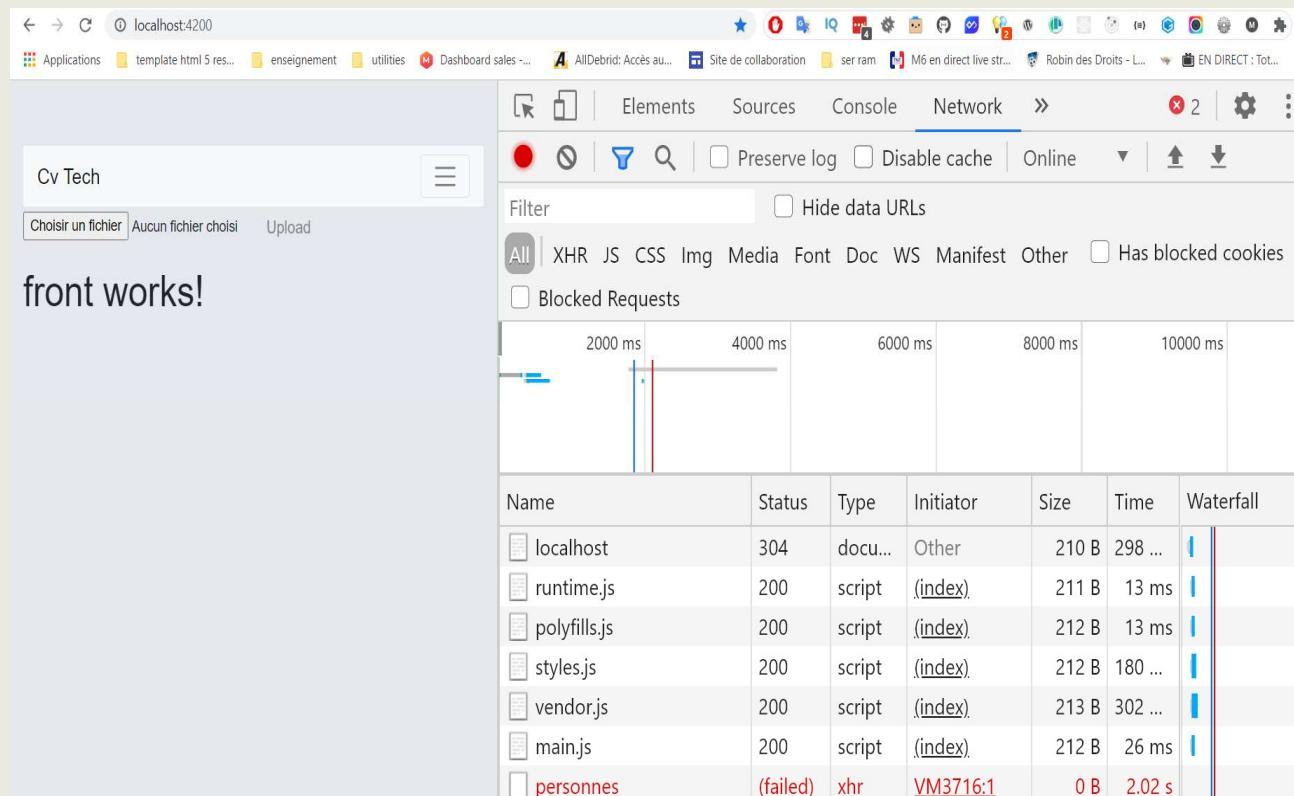


# Shared Module

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

# Lazy Loading

- Par défaut, tous les modules que vous déclarez au niveau du AppModule sont chargé 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 principal et puis de ne charger un module que si on appelle l'une de ses routes.
- Ceci va nous faire gagner en performance.



# Lazy Loading

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

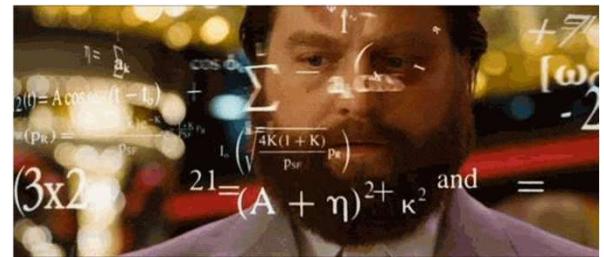
1. Le **module** à charger doit **lui-même gérer sa partie routing**
2. Au niveau du routing principal (AppRoutingModule), créer une **route**, ajouter lui un path ‘ça sera le **préfixe** de toutes les routes du module’, et ajouter une nouvelle clé qui est **loadChildren**. Cette clé va informer angular et lui demander de ne charger le module associé que lorsque on appelle le path défini.
3. Ce **paramètre** prend ou une **chaine de caractère** qui spécifie le module à charger ou un **callback fonction**.
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 `preloadStrategy`.
- 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 nul**.

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

# Pourquoi les NgModules

- La principale raison de l'introduction initiale de NgModules était pragmatique : **regrouper des blocs de construction qui sont utilisés ensemble.**
- Ceci permet non seulement d'augmenter la commodité pour les développeurs, mais également pour **le compilateur Angular**.
- En effet le NgModule permet de **définir le contexte de compilation**. A partir de ce contexte, le compilateur apprend où le programme est autorisé à appeler quels composants.
- Les deux parties imports et declarations permettent d définir ce contexte de compilation.

[https://www.youtube.com/watch?v=DA3efofhpq4&ab\\_channel=ngVikings](https://www.youtube.com/watch?v=DA3efofhpq4&ab_channel=ngVikings)

# Pourquoi les standalone Components ?

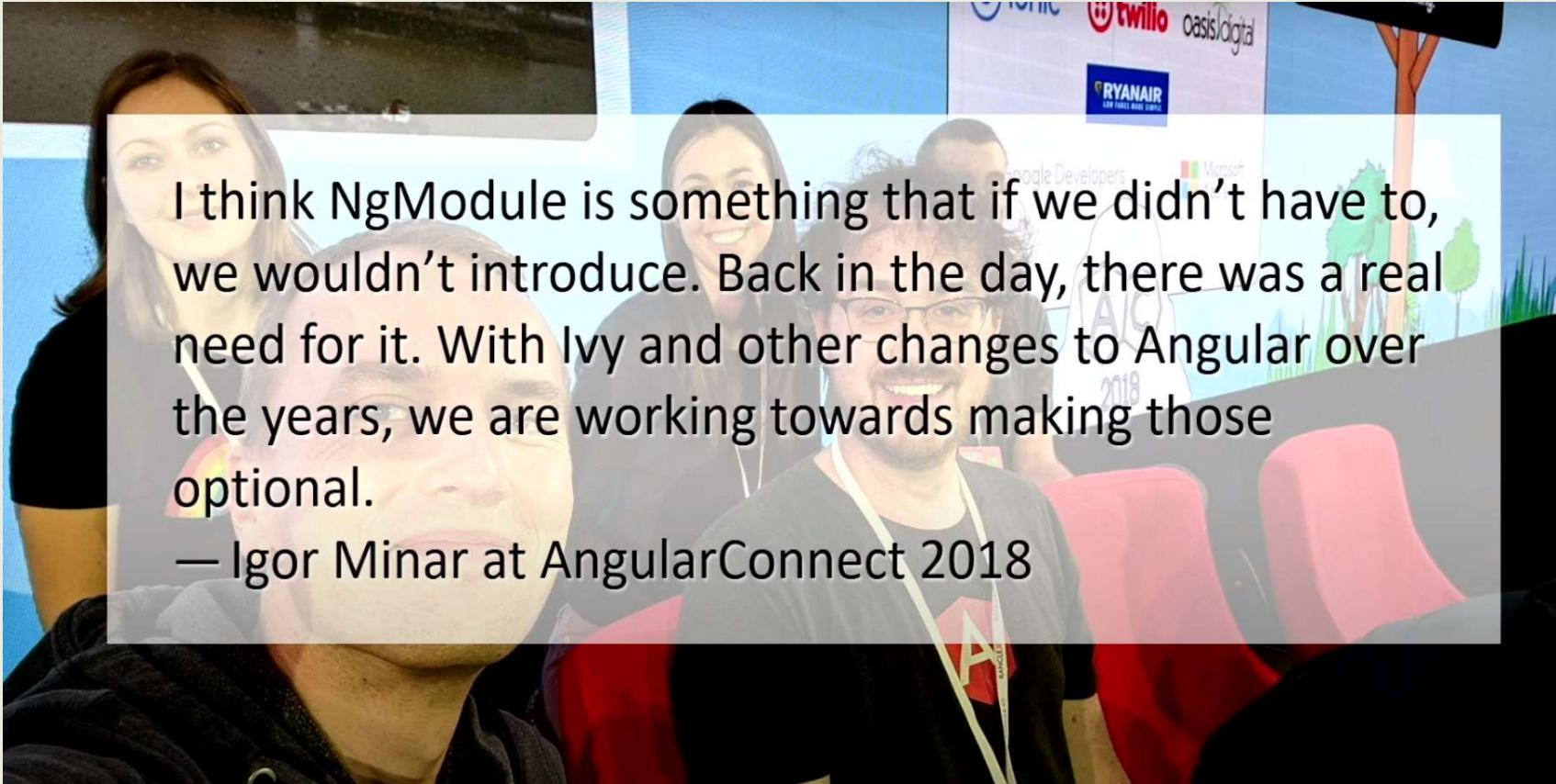
- Le premier problème qui s'est posé est l'ambiguïté que pose ce deuxième système de module en parallèle à celui de EcmaScript.
- Ceci a compliqué l'apprentissage pour les nouveaux apprenants.
- Ceci a poussé la Team Angular à faire en sorte qu'avec Ivy, le nouveau compilateur d'Angular, d'avoir un contexte de compilation par composant.
- C'est ce qui a permis la naissance des standalone component.

[https://www.youtube.com/watch?v=DA3efofhpq4&ab\\_channel=ngVikings](https://www.youtube.com/watch?v=DA3efofhpq4&ab_channel=ngVikings)

# Pourquoi les standalones Components ?

## Single Component Angular Module

### SCAM



I think NgModule is something that if we didn't have to, we wouldn't introduce. Back in the day, there was a real need for it. With Ivy and other changes to Angular over the years, we are working towards making those optional.

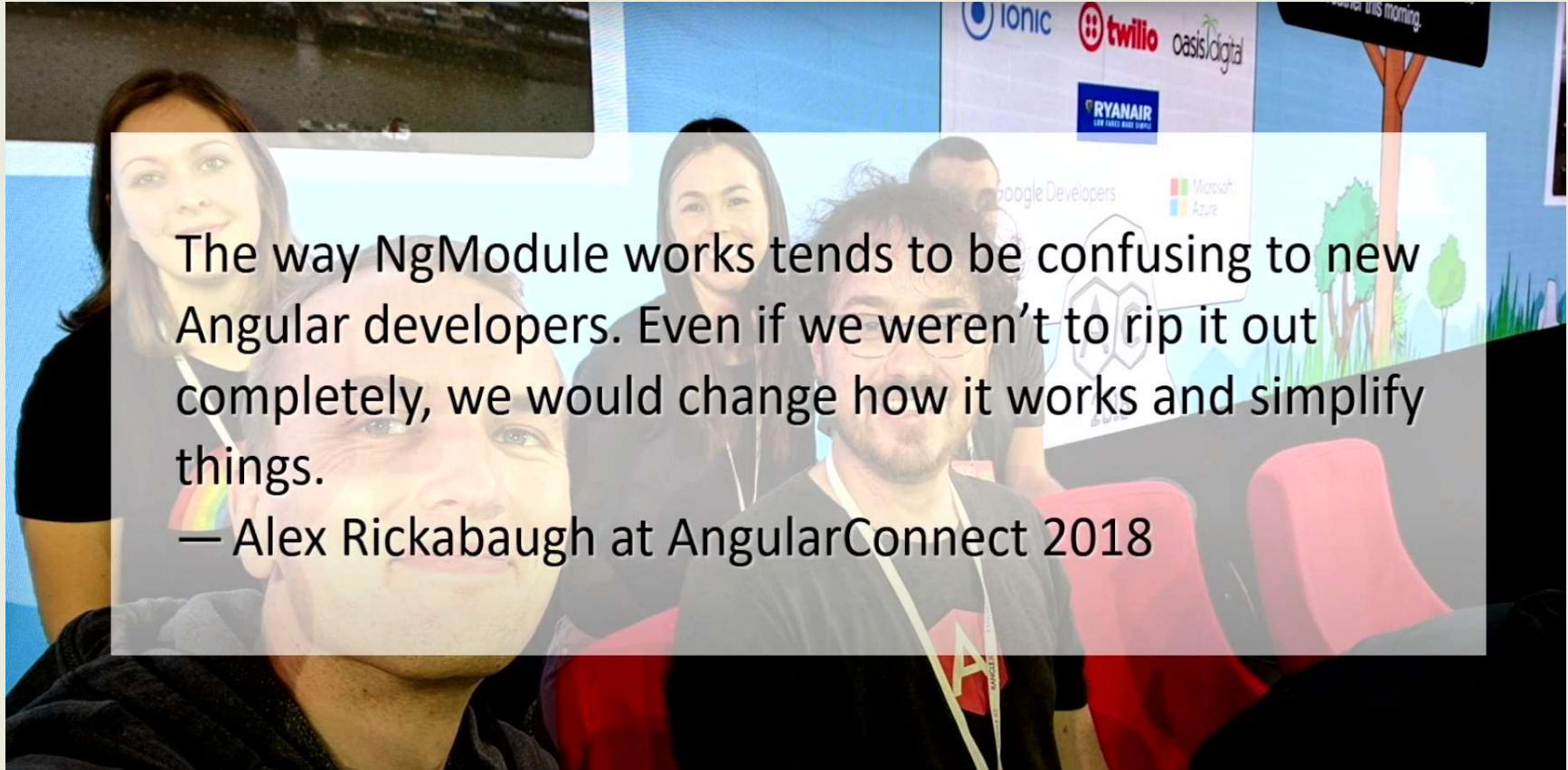
— Igor Minar at AngularConnect 2018

[https://www.youtube.com/watch?v=DA3efofhpq4&ab\\_channel=ngVikings](https://www.youtube.com/watch?v=DA3efofhpq4&ab_channel=ngVikings)

# Pourquoi les standalones Components ?

## Single Component Angular Module

### SCAM



The way NgModule works tends to be confusing to new Angular developers. Even if we weren't to rip it out completely, we would change how it works and simplify things.

— Alex Rickabaugh at AngularConnect 2018

[https://www.youtube.com/watch?v=DA3efofhpq4&ab\\_channel=ngVikings](https://www.youtube.com/watch?v=DA3efofhpq4&ab_channel=ngVikings)

# Standalone Component

## Les composants autonomes

- Un standalone Component est un **composant indépendant d'un module**.
- Les composants standalone offrent un **moyen simplifié** de créer des applications Angular.
- Les applications existantes peuvent éventuellement et progressivement adopter le nouveau style autonome sans aucune modification majeure.
- Ceci est aussi **applicable** aux **directives** et aux **pipes**.
- L'utilisation d'une telle vision simplifie l'apprentissage d'Angular
- Elle facilite aussi plusieurs aspects comme le lazy loading

# Standalone Component

## Les composants autonomes

- Une nouvelle propriété a été introduite au niveau de l'objet de configuration, c'est la propriété **standalone**.
- Si elle est à **true**, l'élément est **considéré standalone**.
- Ces éléments **ne doivent plus être associés** à un **tableau déclaration** d'un NgModule.
- Une nouvelle clé **import** permet aussi **d'importer les dépendances nécessaires**, que ce soit d'autres standalone components ou d'autres NgModule.

# Standalone Component

## Les composants autonomes

- Afin de créer un *standalone* component procéder comme vous lefaite d'habitude avec un composant et ajouter l'option **--standalone**

```
@Component({  
  selector: 'app-standalone',  
  standalone: true,  
  imports: [CommonModule],  
  templateUrl: './standalone.component.html',  
  styleUrls: ['./standalone.component.css']  
})  
export class StandaloneComponent {}
```

# Standalone Component

## Les composants autonomes

### Utilisation dans les NgModule

- Si vous voulez utiliser un **Standalone Component** **dans un NgModule**, importer le **comme** vous importer **un module**.

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

# Standalone Component

## Bootstraper une application avec un Standalone Component

- Afin de **bootstraper** votre application avec un **standalone component**, utilisez la fonction **bootstrapApplication** dans **main.ts**, à la place de **platformBrowserDynamic().bootstrapModule**, en lui passant le **Standalone Component que vous voulez déclencher au lancement de l'application**.
- Dans **index.html** appelez **ce Standalone Component**.

```
bootstrapApplication(StandaloneComponent);
```

main.ts

```
<body>
<div class="container">
  <app-standalone></app-standalone>
</div>
</body>
</html>
```

index.html

# Standalone Component

## Les composants autonomes

### Configurez l'injection de dépendance

- Afin de fournir un **provider** pour **toute l'application**, vous pouvez ajouter en **deuxième paramètre de la fonction bootstrapApplication**, un **objet d'options**.
- Cet objet contient une **clé providers** qui prend en paramètre un **tableau de providers**.
- Vous pouvez aussi **récupérer des providers offerts** par un **module** avec la fonction **importProvidersFrom(moduleCible)**.
- A partir de la **version 15**, vous avez **certaines fonctions spécifiques** pour les **modules les plus utilisés** comme le **http** avec sa fonction **provideHttpClient()**, ou **provideRouter(APP\_ROUTES)**.

# Création d'un système de Routing

- Si vous utilisez une application standalone, la configuration change un peu.
- Vous n'avez plus à définir un module. Il suffit de provider vos route avec la fonction **provideRouter** au niveau de la fonction **bootstrapApplication**

```
bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes)
  ]
}
```

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
]
```

# Lazyload Standalone Component

- Vous pouvez faire du **lazy loading** pour les **standalone component** avec la clé **loadComponent**.

```
{  
  path: 'track',  
  loadComponent: () => import('./track-by/track-by.component')  
    .then(c => c.TrackByComponent)  
}
```

- Vous pouvez faire du **lazy loading** pour vos fichiers de routes

```
{  
  path: "todo",  
  loadChildren: () => import("./todo/todo.routes").then((m) => m.TODO_ROUTES),  
},
```

```
export const TODO_ROUTES: Routes = [  
  {  
    path: "",  
    component: TodoComponent,  
    canDeactivate: [canLeaveGuard],  
  },  
];
```

# Migration

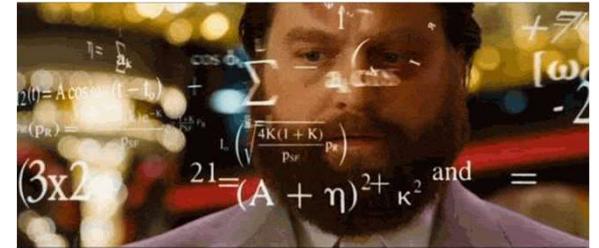
- Angular offre un moyen **de migrer automatiquement une partie** de votre application Angular modulaire (15.2.0) et plus vers une application standalone.
- Vous pouvez suivre les différentes étapes du Guide qui présente des parties automatiques et d'autres manuelles.
- Exécutez la migration **dans l'ordre indiqué ci-dessous, en vérifiant que votre code est généré et exécuté entre chaque étape :**
  1. Exécutez `ng g @angular/core:standalone` et sélectionnez "Convert all components, directives and pipes to standalone"
  2. Exécutez `ng g @angular/core:standalone` et sélectionnez "Remove unnecessary NgModule classes" , **cette étape risque de garder plusieurs modules.**
  3. Exécutez `ng g @angular/core:standalone` et sélectionnez "Bootstrap the project using standalone APIs".

# Migration

- Une partie de votre application sera convertie en **autonome (standalone)**.
- Cependant, **il vous restera du travail à faire manuellement** :
- Rechercher les **imports manquants**, votre compilateur va vous aider dans cette tâche.
- **Recherchez et supprimez toutes les déclarations NgModule restantes** : étant donné que l'étape "Supprimer les NgModules inutiles" ne peut pas supprimer automatiquement tous les modules, vous devrez peut-être supprimer les déclarations restantes manuellement.
- Exécutez les tests unitaires du projet et corrigez les éventuels échecs.

# Exercice

- Cloner ce projet angular17 Modulaire :  
<https://github.com/aymensellaouti/ng17ModuleToConvert>
- Migrer le vers un projet standalone.

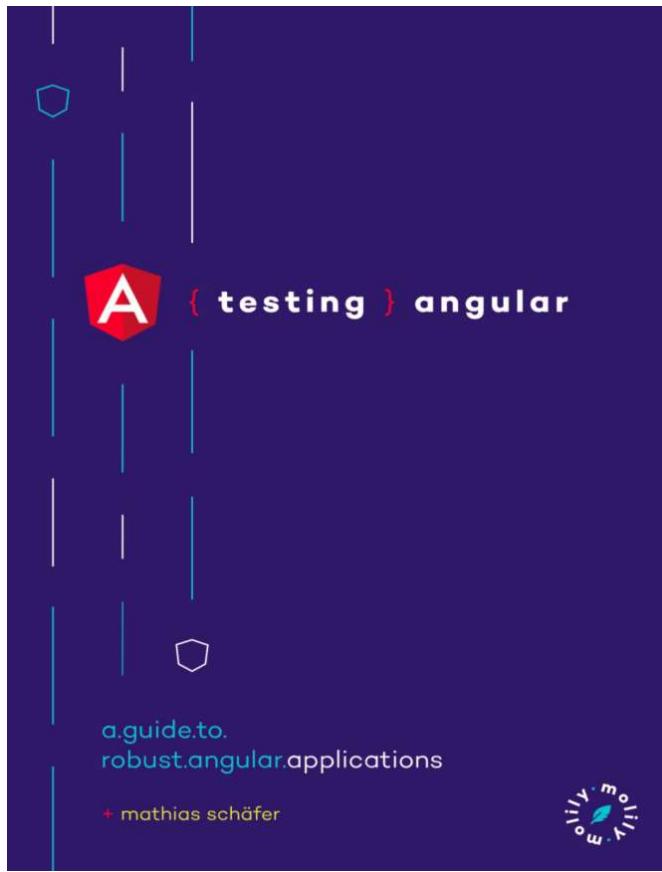


# Angular Tests Unitaires

Aymen sellaouti



# Références



# Tests unitaires : Introduction

- La plus petite unité de test possible
- Couvre une petite fonctionnalité et ne s'occupe pas de comment les différentes unités testées travaillent ensemble.
- Il est isolé et ne doit pas dépendre d'autres tests.
- Rapide, fiable et pointe directement sur le bug en question.

# Tests unitaires : Pourquoi

- Rend votre **code plus robuste**, le bug sera identifié plus tôt
- Vous permettez de **formaliser et de documenter** vos besoins
- Un bon test décrit clairement comment le code d'implémentation doit se comporter
- Un bon test doit couvrir **les scénarios les plus importants**
- Les tests rendent le changement sûr en **empêchant les régressions**

# Tests unitaires : Partout ?

- Alors devriez-vous écrire des tests automatisés pour tous les cas possibles afin de garantir l'exactitude ?
- Non, disent les principes de l'ISTQB : "**Les tests exhaustifs sont impossibles**".
- Il n'est **ni techniquement faisable ni utile** d'écrire des tests pour toutes les entrées et conditions possibles.
- Au lieu de cela, vous devez **évaluer les risques** d'un certain cas et rédiger d'abord des **tests pour les cas à haut risque**.
- Même s'il était viable de couvrir tous les cas, cela vous donnerait un **faux sentiment de sécurité**.

# Angular et les tests unitaires

- Angular avec sa structuration et ses différentes couches se marie parfaitement avec les tests unitaires.
- Chaque couche ayant un rôle unique et une tache bien spécifique, les tests unitaires seront donc propres à chaque partie, composant, pipe, service, directive, ...
- L'utilisation de l'injection de dépendance implique le couplage faible et donc des tests isolés.
- Les providers qui permettent de fournir des classes fictives facilitent aussi cette notion d'isolation.

# Jasmin et Karma

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

```
"@types/jasmine": "~4.3.0",
"jasmine-core": "~4.5.0",
"karma": "~6.4.0",
"karma-chrome-launcher": "~3.1.0",
"karma-coverage": "~2.2.0",
"karma-jasmine": "~5.1.0",
"karma-jasmine-html-reporter": "~2.0.0",
```

# Jasmin et Karma

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



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



# Lancement d'un test

- Afin de lancer un test, vous avez juste besoin d'une seule commande et Karma fait le reste. Il exécutera les tests, ouvrira le navigateur, et affichera un rapport sur l'ensemble des tests.

`ng test`

# Concepts de base de jasmin describe (suite)

- Pour ce qui est de Jasmine, un test se compose d'une ou plusieurs suites. Une suite est déclarée avec un bloc describe :

```
describe('Suite description', () => {  
  /* ... */  
});
```

- Chaque suite décrit un morceau de code, le code à tester.

# Concepts de base de jasmin describe (suite)

- describe est une fonction qui prend deux paramètres.
- Une chaîne. généralement le nom de la fonction ou de la classe testée. Par exemple, describe('TestExampleComponent')
- Une fonction contenant la définition de la suite
  - On peut avoir des describe imbriqué afin de diviser des gros describe en des sections logiques
  - Les blocs de description imbriqués ajoutent une description lisible à un groupe de spécifications. Ils peuvent également héberger leur propre logique de configuration.

```
describe('Suite description', () => {  
  describe('One aspect', () => {  
    /* ... */  
  });  
  describe('Another aspect', () => {  
    /* ... */  
  });  
});
```

# Concepts de base de jasmin

## Spécification (it)

- Chaque **describe** se compose d'une ou plusieurs **spécifications**.
- Une **spécification** est déclarée avec un bloc **it** :

```
describe('description de la suite', () => {
  it('description de la spécification', () => {
    /* ... */
  });
});
```

- **it** est une **fonction** qui prend deux paramètres.
  - Le premier paramètre est une **chaîne** avec une **description lisible**
  - Le second paramètre est une **fonction** contenant **le code de votre test**

# Concepts de base de jasmin

## Spécification (**it**)

- Le pronom **it** fait référence au code testé.
- La phrase doit donc **être lisible et affirme le comportement du code testé**.
- Le code de votre spécification **prouve alors cette affirmation**.
- Ce style d'écriture des spécifications provient du concept de Behavior-Driven Development (**BDD**).
- L'un des objectifs du BDD est de **décrire le comportement du logiciel dans un langage naturel**.
- Chaque partie prenante doit être capable de lire les phrases et de **comprendre comment le code est censé se comporter**.

# Concepts de base de jasmin Specification (it)

- Pour écrire le titre de votre test, le **it**..., demandez-vous ce que doit faire le code que vous testez.
- Pour une LampeComponent par exemple, il doit allumer et éteindre la lampe, on aura donc :

```
it('switch on the lamp', () => {
    /* ... */
});
it('switch off the lamp', () => {
    /* ... */
})
```

- Après **it**, un verbe suit généralement, comme **switch on**, une deuxième famille de testeur préfère suivre le **it** par **should**

# Concepts de base de jasmin Specification (it)

- À l'intérieur du bloc **it** se trouve le code de test réel.
- Indépendamment du framework de test, le code de test se compose généralement de trois phases :
  - **Arrange**
  - **Act**
  - **Assert.**
- **Arrange** est la **phase de préparation** et de mise en place. Par exemple, la classe testée est instanciée. Les dépendances sont mises en place. Des espions (spy) et des faux sont créés.
- **Act** est la **phase où l'interaction** avec le code testé. Par exemple, une méthode est appelée ou un élément HTML du DOM est cliqué.
- **Assert** est la **phase où le comportement du code est contrôlé** et vérifié. Par exemple, la sortie réelle est comparée à la sortie attendue.

# Concepts de base de jasmin

## Spécification (it)

- Imaginons que nous voulons tester un service qui permet d'additionner et de soustraire des entiers.
- Nous commençons par tester l'addition.
  - **Arrange**
    - Nous devons créer une instance du service et de ses dépendances s'ils existent
  - **Act**
    - Appeler la fonction add avec deux paramètres
  - **Assert.**
    - Vérifier que la fonction retourne le bon résultat

# Concepts de base de jasmin

## Attente (Expectation)

- Dans la phase **d'affirmation (Assert)**, le test **compare** la **sortie** ou la **valeur de retour réelle** à la **sortie ou à la valeur de retour attendue**. S'ils sont **identiques**, le test **réussit**. S'ils **diffèrent**, le test **échoue**.
- Afin de gérer ça, jasmine nous offre la fonction **expect**.
- Cette **fonction** est **associée** à un ensemble de **matchers** permettant de faciliter la **validation** de vos **attentes ou expectations**.

```
const expectedValue = 5;
const actualValue = MathService.add(2, 3);
expect(actualValue).toBe(expectedValue);
```

# Jasmin matchers

- Les **matchers** de Jasmine sont des **fonctions** qui permettent de **tester si une valeur donnée correspond à une condition spécifique**. Ils permettent donc de vérifier que les fonctionnalités de l'application se comportent comme prévu.
- **toBe()** : vérifie si deux valeurs sont strictement égales (utilisant l'opérateur "===")
- **toEqual()** : vérifie si deux objets ont les mêmes propriétés et les mêmes valeurs
- **toMatch()** : vérifie si une chaîne de caractères correspond à une expression régulière
- **toBeDefined()** : vérifie si une variable est définie
- **toBeUndefined()** : vérifie si une variable n'est pas définie
- **toBeNull()** : vérifie si une variable est null

# Jasmin matchers

- **toBeTruthy()** : vérifie si une expression est vraie
- **toBeFalsy()** : vérifie si une expression est fausse
- **toContain()** : vérifie si un tableau ou une chaîne de caractères contient un élément spécifié
- **toBeLessThan()** : vérifie si une valeur est inférieure à une autre
- **toBeGreaterThan()** : vérifie si une valeur est supérieure à une autre
- ...

# Concepts de base de jasmin

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

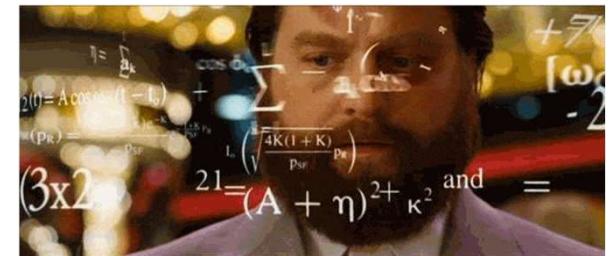
Exemple **expect(etatActuel).toBe(etatExpecté)**

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

# Concepts de base de jasmin

- **xit** permet d'exclure un test individuel
- **xdescribe** permet d'exclure tout le bloc
- **fit** permet de spécifier le test individuel à exécuter
- **fdescribe** permet de spécifier le bloc à exécuter.

# Exercice



➤ Récupérer le Répo suivant :

<https://github.com/aymensellaouti/startinTest>

➤ Créer les tests nécessaires pour le service MathService

# Concepts de base de jasmin

- Lorsque vous écrivez plusieurs spécifications dans une suite, vous réalisez rapidement que **la phase d'arrangement (Arrange) est similaire**, voire identique, dans toutes ces spécifications.
- Par exemple, lors du test du MathService, la phase Arrange consiste toujours à créer une instance de MathService.
- Afin de centraliser ces traitements répétitifs, Jasmine propose quatre fonctions : **beforeEach**, **afterEach**, **beforeAll** et **afterAll**. Ils sont **appelés à l'intérieur d'un bloc describe**.
- Ils attendent un **paramètre**, une fonction qui est appelée **aux étapes données**.

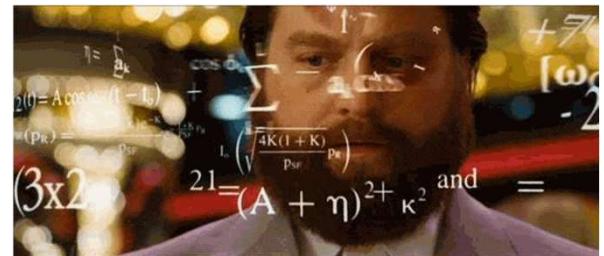
# Concepts de base de jasmin

Jasmin offre des handlers permettant de répéter certaines fonctionnalités.

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

# Exercice

➤ Mettez à jour vos tests.

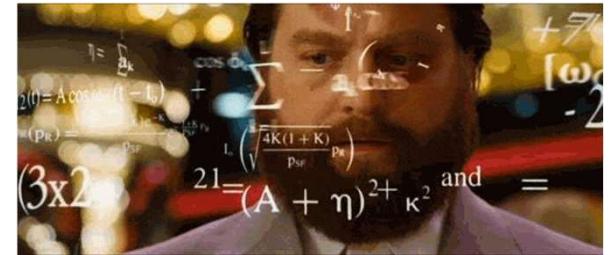


# Angular TestBed

- **TestBed** est l'outil le plus important des utilitaires de test Angular.
- Il vous **facilite** l'étape de préparation de l'environnement de Test (**Arrange**).
- Il vous donne la possibilité **d'injecter un service via la méthode inject**.

```
// Remplacant de new LoggerService()  
loggerService = TestBed.inject(LoggerService);
```

# Exercice



➤ Utilisez les TestBed pour découpler votre test du LoggerService

# Tests E2E

- Ces tests permettent de **SIMULER L'UTILISATION RÉELLE** de votre application.
- Certains tests ont une **vue d'ensemble de haut niveau sur l'application**.
- Ils simulent un **utilisateur interagissant avec l'application** :
  - navigation vers une adresse,
  - lecture de texte,
  - clic sur un lien ou un bouton,
  - remplissage d'un formulaire,
  - déplacement de la souris ou saisie au clavier.

# Tests E2E

- Ces tests font des **attentes** sur ce que l'utilisateur voit.
- Du point de vue de l'utilisateur, **peu importe que votre application soit implémentée dans Angular**.
- L'**expérience complète est testée => TESTS DE BOUT EN BOUT**
- Les tests de bout en bout constituent également la **partie automatisée des tests d'acceptation** puisqu'ils indiquent si l'application fonctionne pour l'utilisateur.

# Tests E2E

## Comment ça marche ?

- Les tests **E2E** vont donc simuler les interactions de l'utilisateur avec votre application.
- Vous allez donc lancer le **navigateur**, et le contrôler afin de simuler un scénario d'interactions.
- Une fois le **scénario exécuté**, vous allez avoir des **attentes** (expectations), exactement comme avec les tests unitaires :
  - Est-ce que les éléments des pages sont correct
  - Est-ce que pour donner suite au click j'ai le bon affichage
  - ...

# Tests E2E

## Cypress

➤ Cypress est un Framework pour les Test E2E dont les avantages sont :

1. Interface utilisateur facile à utiliser
2. Temps de développement rapide
3. Intégration parfaite avec le développement front-end
4. Exécution rapide des tests
5. Capacité à tester directement dans le navigateur
6. Possibilité de déboguer facilement les tests
7. Prise en charge native de la manipulation du DOM et de l'Ajax
8. Documentations et communauté actives
9. Tests fiables et reproductibles.

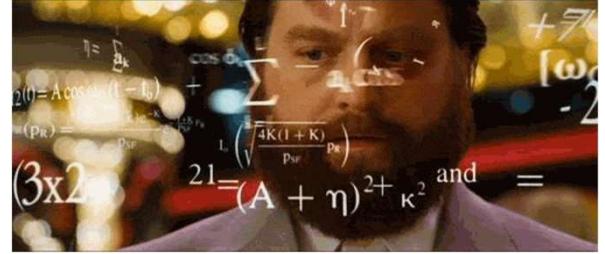
# Tests E2E

## Cypress

- Afin d'installer Cypress utiliser la commande **npm i cypress -save-dev**.
- Avec **angular**, utilisez la commande **ng add @cypress/schematic**
- L'utilisation de cette commande permet d'automatiser la configuration en ajoutant
  - **Cypress** et les packages npm auxiliaires à **package.json**.
  - Le fichier de configuration Cypress **cypress.config.ts**.
  - Modifiez le fichier de configuration **angular.json** afin d'ajouter des commandes d'exécution ng.
- Créez un **sous-répertoire** nommé **cypress** avec des **templates pour vos tests**.

# Exercice

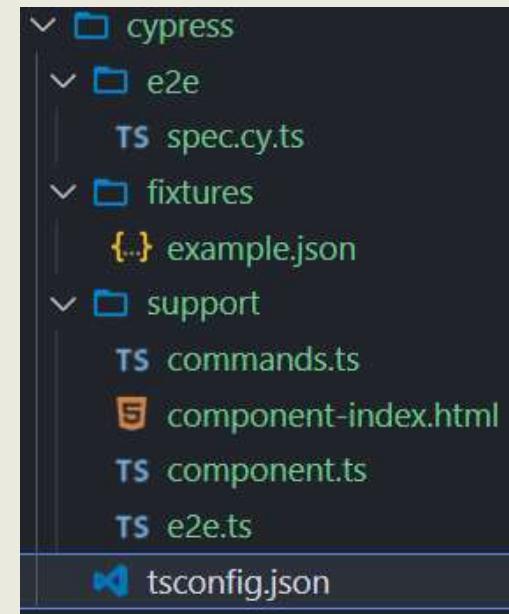
- Installez Cypress et regarder les différents fichiers ajoutés



# Tests E2E

## Le dossier cypress

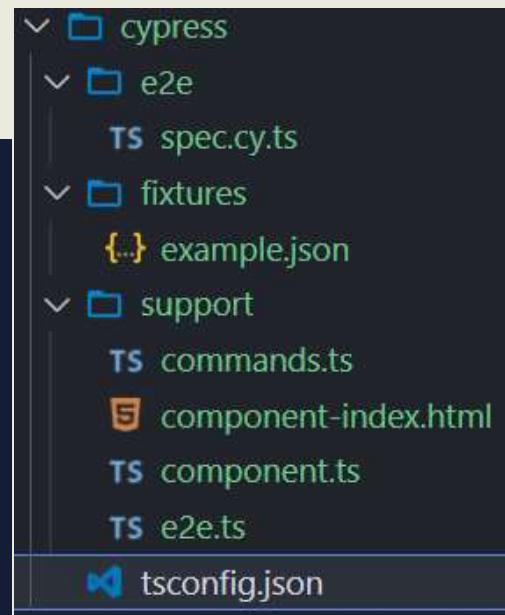
- Le dossier **cypress** généré contient :
  - Une configuration **tsconfig.json** pour tous les fichiers TypeScript spécifiquement dans ce répertoire,
  - Un répertoire **e2e** pour les **tests E2E**,
  - Un répertoire de **support** pour les **commandes personnalisées** et autres assistants de test,
  - un répertoire **fixtures** pour les **données de test**.



# Tests E2E Configuration

- Il y a aussi le fichier cypress.config.ts au niveau de la racine de votre projet.
- Ce fichier vous permet de configurer cypress

```
import { defineConfig } from 'cypress'
export default defineConfig({
  e2e: {
    baseUrl: 'http://localhost:4200'
  },
  component: {
    devServer: {
      framework: 'angular',
      bundler: 'webpack',
    },
    specPattern: '**/*cy.ts'
  }
})
```



## Tests E2E

### Lancer les tests E2E

- Dans package.json on peut identifier deux commandes:
  - **cypress:open** qui exécute la commande **cypress open**
  - **cypress:run** qui exécute la commande **cypress run**

```
"cypress:open": "cypress open",
"cypress:run": "cypress run"
```

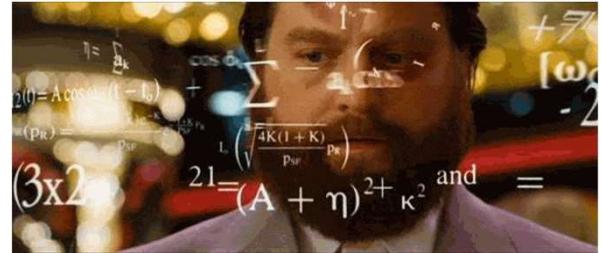
## Tests E2E

### Lancer les tests E2E

- **cypress open** est le mode interactif. Elle ouvre une fenêtre dans laquelle vous pouvez sélectionner le navigateur à utiliser et les tests à exécuter. A chaque changement, tout est mis à jour.
- **cypress run**, c'est le mode non interactif. Exécute les tests dans un navigateur "headless". Cela signifie que la fenêtre du navigateur n'est pas visible. Les tests sont exécutés une fois, puis le navigateur est fermé et la commande shell se termine.
- Cette commande est généralement utilisée dans un environnement d'intégration continue.

# Exercice

➤ Lancez cypress en mode interactif et suivez les étapes



# Tests E2E

## Ecrire des tests E2E

- Vous devez lancer votre **serveur dans un terminal** et **cypress dans l'autre**
- Vos **tests** doivent être dans le **dossier e2e**
- Chaque groupement de test, généralement par page sera représenté par un fichier dont **l'extension est .cy.ts.**
- En règle générale, un fichier contient un bloc de description **describe.**
- On peut avoir **des blocs de description imbriqués.**
- À l'intérieur, les blocs **beforeEach, afterEach, beforeEach, afterEach** peuvent être utilisés de la même manière que les tests Jasmine.
- À l'intérieur des blocs on peut avoir **un ou plusieurs attentes.**

# Tests E2E

## Visiter une page

- Afin d'accéder à une page vous pouvez utiliser visit
- Si vous avez défini votre baseUrl dans la config comme nous l'avons spécifié (Qui est une bonne pratique : <https://docs.cypress.io/guides/references/best-practices#Setting-a-global-baseUrl>), ajoutez l'URI vers lequel vous voulez naviguer.

```
cy.visit('/') // visits the baseUrl
cy.visit('index.html') // visits the local file "index.html" if baseUrl is null
cy.visit('http://localhost:3000') // specify full URL if baseUrl is null or the domain is different
//the baseUrl
cy.visit({
  url: '/pages/hello.html',
  method: 'GET',
})
```

# Tests E2E

## Sélectionner des éléments

- Certaines méthodes jouent le **double rôle de sélecteur et d'assertions** comme `get` qui vérifie que l'élément existe et qui le sélectionne
- `cy.get()`: Cette méthode permet de sélectionner un élément spécifique en utilisant un sélecteur CSS. **Exemple:** `cy.get('#bouton-submit').click()`
- `cy.contains()`: Cette méthode permet de sélectionner un élément en fonction du texte qu'ils contient.

**Exemple:** `cy.contains('Submit').click()`

- `cy.focused()`: Cette méthode permet de sélectionner l'élément qui a le focus actuellement.

**Exemple:** `cy.focused().should('have.class', 'form-input-focused')`

# Tests E2E

## Sélectionner des éléments

➤ **cy.first():** Cette méthode permet de sélectionner le premier élément d'une liste d'éléments.

**Exemple:** `cy.get('.liste-éléments').first()`

➤ **cy.last():** Cette méthode permet de sélectionner le dernier élément d'une liste d'éléments.

**Exemple:** `cy.get('.liste-éléments').last()`

➤ **cy.parent():** Cette méthode permet de sélectionner le parent d'un élément donné.

**Exemple:** `cy.get('.élément-enfant').parent()`

# Tests E2E

## Sélectionner des éléments

➤ **cy.root():** Cette méthode permet de sélectionner la racine du document HTML.

**Exemple:** cy.root().should('have.class', 'racine')

➤ **cy.children():** Cette méthode permet de sélectionner les enfants d'un élément donné.

**Exemple:** cy.get('.élément-parent').children().should('have.length', '3')

➤ **cy.next():** Cette méthode permet de sélectionner l'élément suivant d'un élément donné.

**Exemple:** cy.get('.élément-précédent').next()

➤ **cy.prev():** Cette méthode permet de sélectionner l'élément précédent d'un élément donné. **Exemple:** `cy.get('.élément-suivant').prev()`.

# Tests E2E

## Sélectionner des éléments

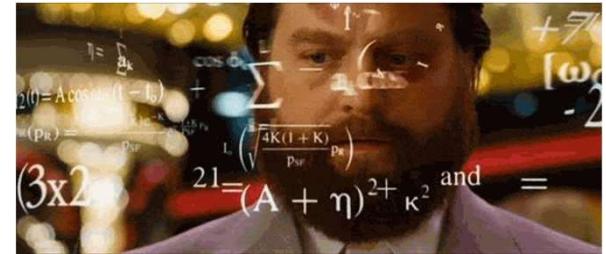
### Les bonnes pratiques

- Cypress déconseille d'utiliser les sélecteurs susceptibles d'être modifiés fréquemment comme les **classes** ou les **ids** c'est un Anti-Pattern.
- La bonne pratique est d'utiliser des **attributs** avec ce **pattern data-\*** permettant de donner un contexte à vos sélecteurs et de les **isoler des changements css et js**.
- De plus en utilisant cette technique le **selector Playground de cypress va préférer ces sélecteurs et les mettre en avant** :
  - data-cy
  - data-test
  - data-testid

<https://docs.cypress.io/guides/references/best-practices#Selecting-Elements>

# Exercice

- Dans la page Cv, vérifiez l'existence de la liste des cvs.
- Vérifiez qu'il n'existe pas de cvCard au départ (utiliser l'assertion `.should('not.exist')`).



# Tests E2E

## Test des requêtes HTTP

### API Mockés

- Cypress vous permet de **remplacer une réponse** et de **contrôler le corps, l'état, les en-têtes ou même le délai**.
- `cy.intercept()` est utilisé pour contrôler le comportement des requêtes HTTP. Vous pouvez **définir de manière statique le corps**, le **status HTTP**, les **en-têtes** et d'autres caractéristiques de réponse.
- Elle peut **prend en paramètre un grand nombre de combinaison selon votre cas d'utilisation**.

# Tests E2E

## Test des requêtes HTTP

### API Mockés

```
// spying
cy.intercept('/users/**')
cy.intercept('GET', '/users*')
cy.intercept({
  method: 'GET',
  url: '/users*',
  hostname: 'localhost',
})
// spying and response stubbing
cy.intercept('POST', '/users*', {
  statusCode: 201,
  body: {
    name: 'Peter Pan',
  },
})
// spying, dynamic stubbing, request modification, etc.
cy.intercept('/users*', { hostname: 'localhost' }), (req) => {
  /* do something with request and/or response */
}
```

## Tests E2E

### Test des requêtes HTTP

#### API Mockés / intercept, mockez une réponse

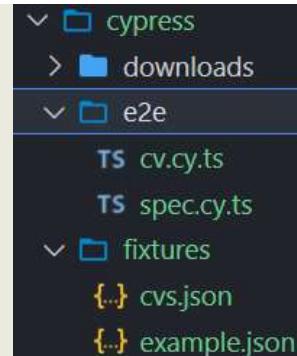
➤ Lorsque vous utilisez intercept suivez les étapes suivantes:

1. Préparer l'interception
2. Lancer l'opération souhaité
3. Lancez vos Assertions

# Tests E2E

## Test des requêtes HTTP

### API Mockés / intercept, mockez une réponse



- Vous pouvez moquer la réponse de votre api avec des fixtures.
- Les fixtures peuvent êtres de plusieurs types et vous avez le dossier fixtures pour les stocker.

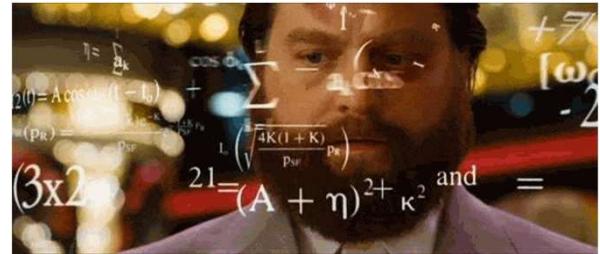
```
// requests to '/update' will be fulfilled
// with a body of "success"
cy.intercept('/update', 'success')
// requests to '/users.json' will be fulfilled
// with the contents of the "users.json" fixture
cy.intercept('/users.json', { fixture: 'users.json' })
cy.intercept('/projects', {
  body: [{ projectId: '1' }, { projectId: '2' }],
})
```

```
cy.intercept('/not-found', {
  statusCode: 404,
  body: '404 Not Found!',
  headers: {
    'x-not-found': 'true',
  },
})
```

```
cy.intercept(
{
  method: 'GET',
  url: API.cv,
},
{
  fixture: 'cvs',
}
)
```

# Exercice

- Faites en sorte d'avoir des fixtures pour la liste des cvs permettant de tester cette liste.
- Vérifier que l'affichage utilise vos fixtures
- Ajouter des fixtures pour la sélection d'un cv par son id.



# Tests E2E

## Les assertions

- Cypress intègre plusieurs assertions de diverses bibliothèques d'assertions JS telles que Chai, jQuery, etc.
- Nous pouvons globalement classer toutes ces assertions en deux segments en fonction du sujet sur lequel nous pouvons les invoquer :
  - Les assertions implicites
  - Les assertions explicites

## Tests E2E

### Les assertions implicites

- Lorsque l'assertion s'applique à l'objet fourni par la **commande chaînée parente**, elle s'appelle une assertion **implicite**.
- Cette catégorie d'assertions inclut généralement des commandes telles que `".should()" et ".and()"`.
- Comme ces commandes **ne sont pas indépendantes** et dépendent toujours de la commande parente précédemment chaînée, elles **héritent et agissent automatiquement sur l'objet généré par la commande précédente**.
- Généralement, nous utilisons des assertions implicites lorsque nous voulons :
  - Affirmer plusieurs validations sur le même sujet.
  - Changez de sujet avant de faire des affirmations sur le sujet.

# Tests E2E

## Les assertions

```
cy.get('[data-cy=list-cvs]').should('have.length',2);
```

```
cy.get('.assertion-table')
  .find('tbody tr:last')
  .should('have.class', 'success')
  .find('td')
  .first()
// valider le contenu d'un élément
  .should('have.text', 'Column content')
  .should('contain', 'Column content')
  .should('have.html', 'Column content')
  .should('match', 'td')
```

```
<table class="table table-bordered assertion-table">
  <thead>
    <tr><th>#</th><th>Column heading</th><th>Column heading</th></tr>
  </thead>
  <tbody>
    <tr><th scope="row">1</th><td>Column content</td><td>Column content</td></tr>
    <tr><th scope="row">2</th><td>Column content</td><td>Column content</td></tr>
    <tr class="success"><th scope="row">3</th><td>Column content</td><td>Column content</td></tr>
  </tbody>
</table>
```

#	Column heading	Column heading
1	Column content	Column content
2	Column content	Column content
3	Column content	Column content

// Pour vérifier qu'un texte valide une expression régulière,  
// préférer l'utilisation de contains

```
cy.get('.assertion-table')
  .find('tbody tr:last')
// finds first element with text content matching regular
expression
  .contains('td', /column content/i)
  .should('be.visible')
```

# Tests E2E

## Les assertions

### have

- **exist** : pour vérifier l'existence d'un élément
- **be.visible** : pour vérifier la visibilité d'un élément
- **be.enabled** : pour vérifier l'état activé/désactivé d'un élément
- **be.checked** : pour vérifier l'état coché/décoché d'un élément
- **have.value** : pour vérifier la valeur d'un élément
- **have.text** : pour vérifier le texte d'un élément
- **have.css** : pour vérifier la présence d'un attribut de style sur un élément
- **have.class** : pour vérifier la présence d'une classe sur un élément

# Tests E2E

## Les assertions

- **have.length** : pour vérifier la longueur d'un objet.
- **be.true / be.false** : pour vérifier si une expression est vraie ou fausse
- **eq / equal / eql** : pour vérifier l'égalité de deux valeurs
- **contain** : pour vérifier la présence d'une valeur dans un tableau ou une chaîne de caractères
- **match** : pour vérifier si une chaîne de caractères correspond à une expression régulière
- **be.greaterThan / be.lessThan** : pour vérifier si une valeur est supérieure ou inférieure à une autre valeur.

# Tests E2E

## Location

- Afin d'avoir des informations sur la localisation actuelle, donc l'url actif, vous pouvez utiliser la commande location.
- Avec l'assertion should, vous pouvez lui passez un callback qui prend en paramètre la location et appelle les exceptions que vous voulez valider.

```
cy.location().should((location) => {
    expect(location.pathname).to.equal('/cv/1');
});
```

```
Location : ▾ Object [1]
  auth: ""
  authObj: undefined
  hash: ""
  host: "localhost:4200"
  hostname: "localhost"
  href: "http://localhost:4200/cv/1"
  origin: "http://localhost:4200"
  pathname: "/cv/1"
  port: "4200"
  protocol: "http:"
  search: ""
  superDomain: "localhost"
  superDomainOrigin: "http://localhost:4200"
  ▶ toString: f wrapper()
  ▶ [[Prototype]]: Object
```

# Tests E2E

## Déclencher des actions

- Cypress vous permet de simuler des fonctions.
- Pour écrire dans un élément DOM, utilisez la commande `.type()`.
- Vous pouvez effacer le champ avant de taper avec `clear()`

```
it('Visits the initial project page', () => {
  cy.visit('/');
  cy.contains('Faurecia');
  cy.get('[data-cy=email-input]')
    .type('aymen@email.com')
    .should('have.value', 'aymen@email.com')
});
```

# Tests E2E

## Déclencher des actions

- Pour avoir le **focus sur un élément du DOM**, utilisez la commande **focus()**
- Pour **perdre le focus sur un élément du DOM**, utilisez la commande **blur()**
- Pour **soumettre un formulaire**, utilisez la commande **cy.submit()**
- Pour **cliquer sur un élément du DOM**, utilisez la **commande click()**. Si l'élément **n'est pas visible ou n'est pas enabled** ajouter en paramètre {force:true}
- Pour **cocher une case ou une radio**, utilisez la commande **check()**.
- Pour **sélectionner une option dans un select**, utilisez la commande **select()**.

```
.click({ force: true });
```

<https://docs.cypress.io/guides/core-concepts/interacting-with-elements>

# Tests E2E

## Déclencher des actions

- Afin de simuler le click sur un caractère spécial, entre, insert, delete, utilisez la syntaxe suivante:

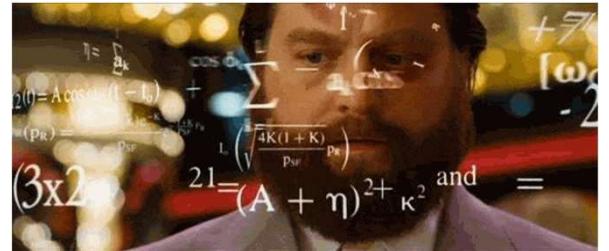
```
// Special characters:  
cy.get('input').type('{enter}')  
cy.get('input').type('{backspace}')  
cy.get('input').type('{del}')  
cy.get('input').type('{esc}')  
cy.get('input').type('{end}')  
cy.get('input').type('{home}')  
cy.get('input').type('{insert}')  
cy.get('input').type('{moveToEnd}') // Move cursor to the end of  
// typeable element  
cy.get('input').type('{moveToStart}') // Move cursor to the start of  
// typeable element  
cy.get('input').type('{pageDown}') // Scroll down  
cy.get('input').type('{pageUp}') // Scroll up  
cy.get('input').type('{selectAll}') // Select the entire input value
```

```
// Arrows:  
cy.get('input').type('{upArrow}')  
cy.get('input').type('{downArrow}')  
cy.get('input').type('{leftArrow}')  
cy.get('input').type('{rightArrow}')
```

```
// Modifier keys:  
cy.get('input').type('{shift}')  
cy.get('input').type('{ctrl}')  
cy.get('input').type('{alt}')
```

# Exercice

- Ecrivez un test qui permet de tester le fonctionnement de la sélection d'un cv via le click sur le bouton détails.
- Utilisez des fixtures
- Simulez un click sur le bouton détails du premier élément.
- Vérifiez qu'en cliquant sur le bouton, vous accéder à la bonne url
- Vérifier que les infos du cv sont bien affichées



**Merci pour votre  
attention**