



LI262

# Applications web : Angular





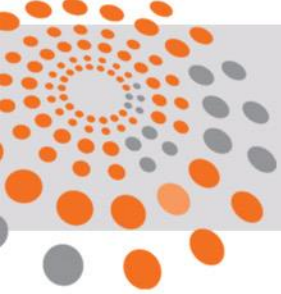
# Introduction

**Nicolas Lethuillier**

Dev web @ Coaxys

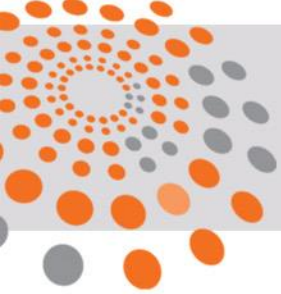
Expérience dans diverses technos web  
Développement front-end





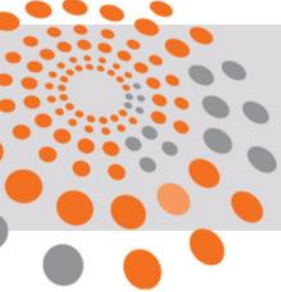
# Organisation

- Durée : 5 jours
- Horaires : 9h - 17h
- Lieu : Dixisoft



# Prérequis

- Avoir participé à un projet web
- Être autonome en HTML, CSS, Javascript (ES5)
- Être muni(e) d'un ordinateur avec MacOS, Windows ou Linux



# Objectifs

- Comprendre ce qu'est une application web
- Être autonome sur **Angular** (version 8)
- Être autonome sur **Bootstrap**
- Être capable de prendre en main le développement front d'une application



# Déroulement

- **Lundi** : présentation d'Angular, TP au fil de l'eau
- **Mardi** : Tour of Heroes (tutoriel Angular)
- **Mercredi** : Tour of Heroes (tutoriel Angular)
- **Jeudi** : présentation de Bootstrap, application à ToH
- **Vendredi** : fin du design et des fonctionnalités de ToH
- **Votre participation est importante !**  
*Cette formation doit être interactive, n'hésitez pas à poser des questions ou faire des commentaires*

# Au menu...



I.

## Contexte

*Les applications web, les frameworks, les outils*



II.

## Langages

*HTML5, CSS3, ES6, TypeScript*



III.

## TypeScript

*Intérêt, fonctionnement*



IV.

## Angular : initialisation

*Présentation, nouveau projet, déploiement, modules*



V.

## Angular : composants

*Premier composant, directives, binding, lifecycle hooks*

# Au menu...



**VI.**     **Angular : routing**  
*Navigation dans une SPA*



**VII.**    **Tour of Heroes (1/2)**  
*Initialisation du projet*



**VIII.**   **Outils avancés**  
*Services, pipes, directives, EventEmitter*



**IX.**     **HTTP & RxJS**  
*Appels serveurs, Observables*



**X.**      **Tour of Heroes (2/2)**  
*Fin du tutoriel officiel*



**XI.**     **Dernier round**  
*Subject, RxJS avancé, AOT*





# CONTEXTE



- Au XX<sup>ème</sup> siècle, on avait des sites web et des applications :



### Sites web

*Vitrines, sites simples...*



### Applications

*Logiciels installés destinés  
à effectuer des actions*

- En 2019, on a aussi des applications web :



**Applications web**  
*Vitrines, sites simples...*



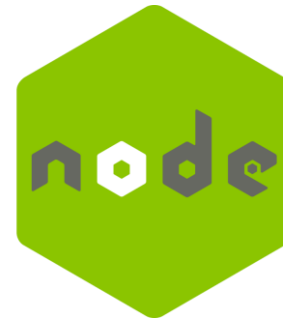
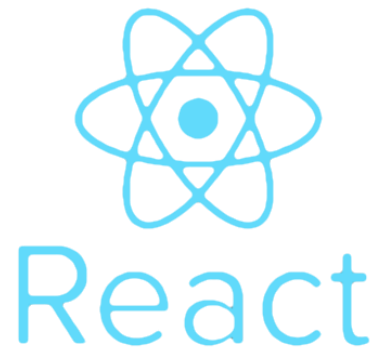
**Applications**  
*« Clients lourds » pour exploiter  
toute la puissance de la machine*

- Une **application web** est donc un site web qui ne se contente pas d'afficher de l'information, mais qui permet à l'utilisateur :
  - De consulter, modifier des informations qui lui sont propres
  - Effectuer des actions qui ont un impact sur sa navigation et/ou celle des autres



- Avec la mode du Cloud, de nombreux éditeurs font des versions « web » de leurs logiciels

*Word Online, Photopea, etc.*





- Angular permet de créer des applications « *Single Page* » (SPA)
- Il utilise le paradigme de la **programmation orientée composants**
- Il utilise les langages et extensions de fichiers bien connus du web (HTML, CSS, JavaScript)... en ajoutant quelques bonus










- Une application Angular est donc composée :
  - De fichiers `.html`
  - De fichiers `.css`
  - De fichiers `.js`
  - De fichiers `.ts`
  - De fichiers `.conf`, `.properties`, etc.
- Elle inclut également un `index.html` à la racine
- L'arborescence, elle, est (quasiment) libre !



- Une application Angular est une application Node.js
- La racine du projet possède toujours les éléments suivants :

	<code>node_modules</code>	Code des dépendances (bibliothèques, etc.)
	<code>src</code>	Code de l'application
	<code>angular-cli.json</code>	Configuration d'angular-cli
	<code>package.json</code>	Liste des dépendances
	<code>tsconfig.json</code>	Configuration de TypeScript





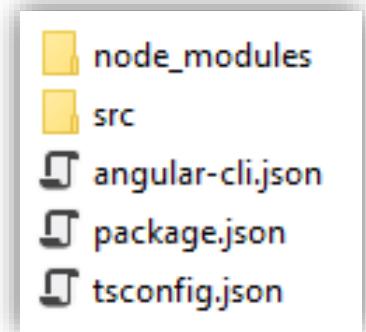


# webpack

- Webpack est un outil couteau-suisse, indépendant d'Angular mais requis pour en faire
- Il gère le découpage en modules, le packaging, ...
- Bonne nouvelle : vous n'aurez pas (trop) à vous en soucier  
*C'est aussi une de ses qualités !*



- Npm est le « gestionnaire de paquets officiel de Node.js » (*Wikipedia*)
- Il permet d'installer, désinstaller et administrer les dépendances de nos projets
- Celui-là, par contre, c'est un incontournable !



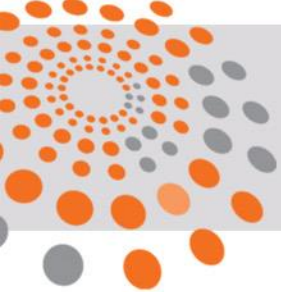
- Exemples :

`npm install`

*// Téléchargement et installation de toutes les dépendances du projet  
// Cette commande lit le fichier « **package.json** », télécharge les paquets,  
// et les installe dans le répertoire « **node\_modules** »*

`npm update`

*// Mise à jour des dépendances du projet*



- Exemples :

```
npm install html2pdf
```

*// Installation de la bibliothèque **html2pdf** pour le projet actuel*

```
npm install -s ng2-social-share
```

*// Installation de **ng2-social-share** pour le projet actuel*

*// avec sauvegarde dans le « **package.json** » du projet*

```
npm install -g ng-youtube-embed
```

*// Installation globale de **ng-youtube-embed***

*// Tous les projets en local peuvent désormais l'utiliser*



- Angular-cli (*Command Line Interface*) permet d'administrer son projet Angular en ligne de commande.
- Exemples :

```
ng new my-project
```

```
// Création d'un nouveau projet Angular  
// avec les paramètres par défaut
```

```
ng serve
```

```
// Lancement de l'application « en live »
```



**webpack**



**Angular CLI**

- On va donc commencer par installer tout ça !



- Etape 1/3 : **Node.js / npm**

<https://nodejs.org/>

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

Download for Windows (x64)

**10.16.3 LTS**

Recommended For Most Users

**12.9.0 Current**

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

Or have a look at the [Long Term Support \(LTS\) schedule](#).

Sign up for [Node.js Everywhere](#), the official Node.js Monthly Newsletter.



- Etape 1/3 : **Node.js / npm**

```
C:\> Invite de commandes

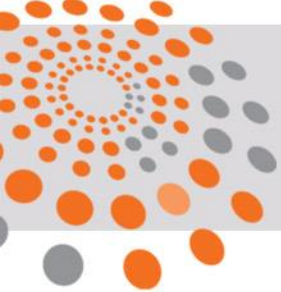
Microsoft Windows [version 10.0.18362.295]
(c) 2019 Microsoft Corporation. Tous droits réservés.

C:\Users\NicolasLETHUILLIER>node -v
v10.16.3

C:\Users\NicolasLETHUILLIER>npm -v
6.9.0

C:\Users\NicolasLETHUILLIER>
```





- Etape 2/3 : **angular-cli**

**npm install -g @angular/cli**

```
C:\Users\NicolasLETHUILLIER>npm install -g @angular/cli
C:\Users\NicolasLETHUILLIER\AppData\Roaming\npm\ng -> C:\Users\NicolasLETHUILLIER\AppData\Roaming\npm\node_modules\@angular\cli\bin\ng

> @angular/cli@8.2.2 postinstall C:\Users\NicolasLETHUILLIER\AppData\Roaming\npm\node_modules\@angular\cli
> node ./bin/postinstall/script.js

? Would you like to share anonymous usage data with the Angular Team at Google under
Google's Privacy Policy at https://policies.google.com/privacy? For more details and
how to change this setting, see http://angular.io/analytics. No
+ @angular/cli@8.2.2
added 240 packages from 185 contributors in 19.928s

C:\Users\NicolasLETHUILLIER>
```

```
C:\Users\NicolasLETHUILLIER>
```

- Etape 3/3 : **l'environnement !**

Installer l'IDE de votre choix

Mon conseil :



Visual Studio Code

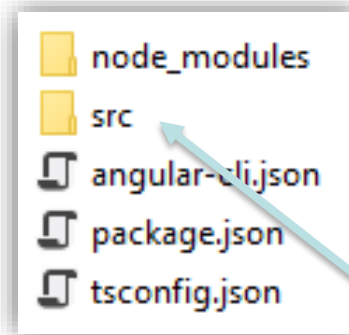
*#Microsoft*

*#Libre*

*#bourré\_d'extensions*

*#idéal\_pour\_le\_web*

- Vous avez désormais toutes les briques nécessaires pour générer, développer et packager un projet Angular
- Nous avons vu ce qu'il y a à la racine d'un projet :



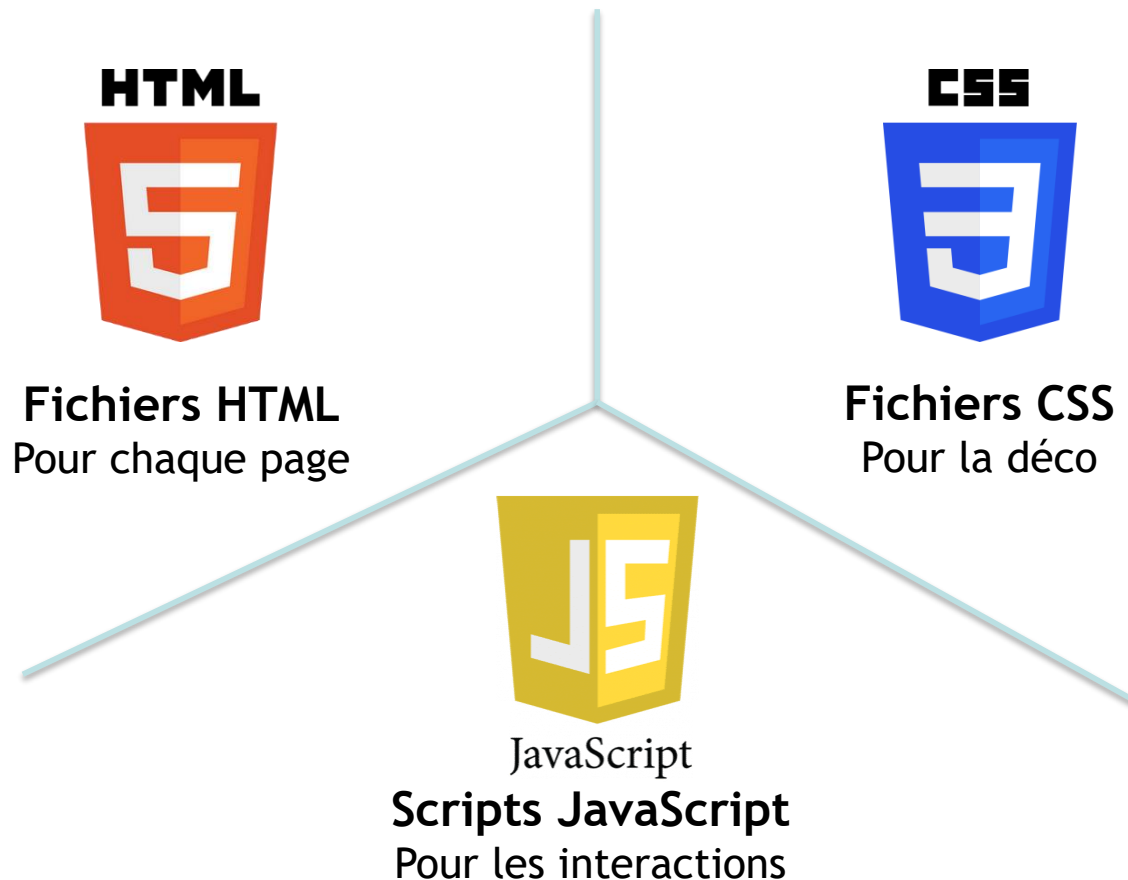
- Nous allons maintenant parler des sources, et en particulier des langages mis en jeu.
- Des questions ?



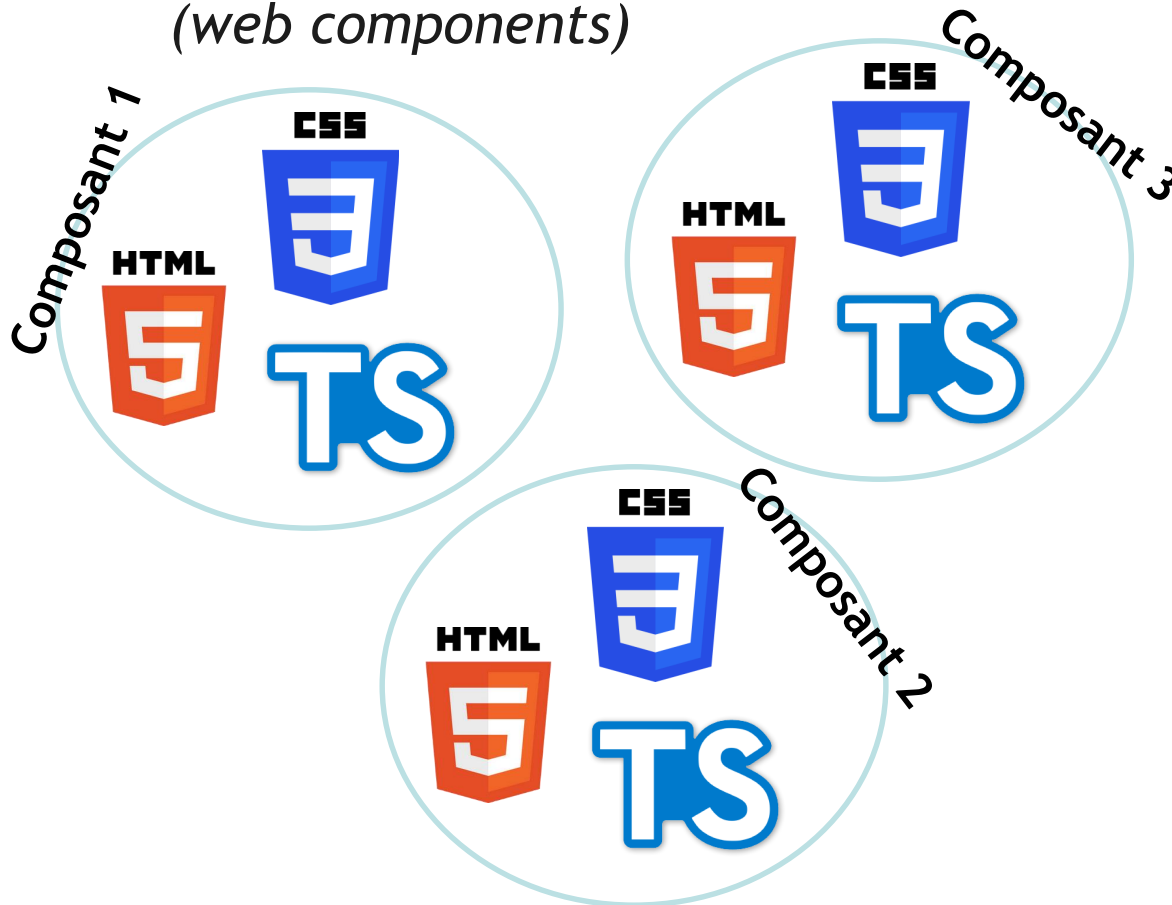
# LANGAGES



- Un projet web est en général composé de :



- Un projet Angular est, lui, composé de composants web (*web components*)



# TS

Fichiers TypeScript

# HTML

index.html



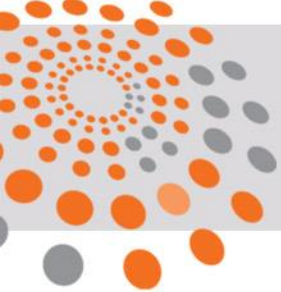
- La programmation **orientée composants** consiste à découper son application en une multitude de composants web indépendants
- Les composants sont des briques qui contiennent chacune :
  - Son code (fichier TypeScript)
  - Son template (fichier HTML)
  - Son style (fichier CSS)



- Exemple :

The screenshot displays the 'L'Œuvre Cinéphile' website. The header includes navigation links: 'ORCHESTRE À VENTS', 'ORCHESTRE À CORDES', 'CHAMBRE', 'DUO', and 'CHERCHER'. Social media icons for Twitter and Facebook are also present. The main banner features a collage of Marvel characters (Captain America, Iron Man, Doctor Strange) with the text 'Marvel, Captain America: First Avenger, Iron Man 3, Doctor Strange, The Avengers' and 'Marvel Suite'. Below the banner, the 'FICHE TECHNIQUE' section provides details: 'Composé par Michael Giacchino, Alan Silvestri, Brian Tyler', 'Orchestre à cordes', 'Créé le 09/07/2019', and tags 'Cinéma/TV', 'Marvel', and 'Medley'. A play button icon and a 4:37 duration are shown. The musical score section, titled 'MARVEL SUITE', includes the conductor 'Conducteur', the composers 'M. Giacchino, A. Silvestri, B. Tyler', and the arranger 'Arr. Nicolas Lethuillier'. The score is for Violon I, with a tempo of 164 and dynamics like *mp* and *mf*.





- Conclusion :
  - L'intérêt principal des composants est la **réutilisabilité**  
*On ne duplique plus le code !*
  - Un autre intérêt : les **scopes CSS**  
*Chaque composant a son fichier CSS, qui n'impacte que lui*  
→ *On peut donc clarifier le code, avec des standards pour les classes, etc.*  
*sans craindre les dommages collatéraux*
  - On peut créer des composants de toutes tailles :
    - Un lecteur PDF de 2000px de haut  
avec un bouton « Enregistrer » et un bouton « Imprimer »
    - Une note de 1 à 5 étoiles
  - L'important est de rationnaliser



- Quelle que soit la technologie web employée, des connaissances en HTML sont primordiales
- **Attention** : créer des composants permet d'« étendre » le HTML en rajoutant des balises



- Exemple :

```
▶ <div _ngcontent-ekv-c5 class="length">...</div>
▶ <div _ngcontent-ekv-c5 class="band">...</div>
  <!--bindings={
    "ng-reflect-ng-if": "4"
  }-->
▶ <star-mark _ngcontent-ekv-c5 _ngghost-ekv-c7 ng-reflect-mark="
  <div _ngcontent-ekv-c5 class="created">Crée le 20/08/2019</di
▶ <div _ngcontent-ekv-c5 class="tagsbox">...</div>
</div>
▼ <div _ngcontent-ekv-c5 class="fieldset listen">
  <div _ngcontent-ekv-c5 class="legend">Ecouter</div>
▶ <audio-player _ngcontent-ekv-c5 _ngghost-ekv-c8 ng-reflect-aud
  </audio-player>
</div>
▶ <div _ngcontent-ekv-c5 class="fieldset download">...</div>
```



- Il faut donc :
  - Faire attention au nommage (par défaut préfixé « app- »)
  - Bien connaître ses balises HTML
- Allez, un petit quiz ?



### *HTML5 ou composant ?*


 fieldset


 viewer


 video


 datalist

 h7


 star

 legend


 bucket

 icon

 progress

 link

 article

 audio

 code

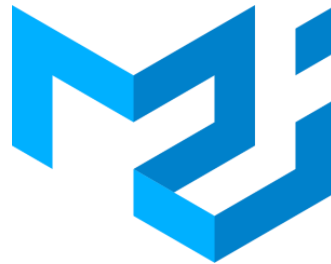
 pre



- Chaque composant a son propre fichier CSS
- On peut donc avoir deux éléments avec les mêmes classes mais des styles différents (scopes CSS)
- On peut également utiliser le fichier **style.css** à la racine du répertoire « src »
  - Celui-ci impacte l'ensemble du site (composants inclus)



- On peut utiliser n'importe quel framework CSS existant



- Angular n'apporte aucun style et/ou pratique quelconque



- Angular permet l'utilisation de **préprocesseurs CSS**



- Mais ce n'est pas l'objet de ce cours, nous en resterons à CSS 😊





- Angular est développé en **TypeScript**
- C'est donc le langage principalement utilisé dans les applications Angular
- TypeScript est une « *surcouche* » de **JavaScript** :  
*il apporte des fonctionnalités supplémentaires  
et surtout des contraintes de développement*
- Qualité +++



- Créé par Microsoft en 2012  
*par Anders Hejlsberg, l'inventeur du C#*
- Supporte la spécification ECMAScript 6
- Utilise donc beaucoup, notamment, les fonctions fléchées :

```
myArray.forEach(function(item) {  
    // Do something with item  
});
```



```
myArray.forEach(item => {  
    // Do something with item  
});
```



- Le principal apport de **TypeScript**, comme son nom l'indique, est la **gestion de types** que n'a pas JavaScript
- On va donc manipuler des **types** et des **classes** comme on le fait en **Java** ou en **C#**
- Dès lors, toute variable pourra être typée  
*On s'appliquera donc à le faire systématiquement*



- Au build, le code TypeScript est compilé en code JavaScript, pour se retrouver avec la triade du web habituelle :
  - *HTML*
  - *CSS*
  - *JavaScript*
- Google Chrome permet cependant de debugger le TypeScript (voir plus tard)



- Si vous avez **Visual Studio**, il est possible que **TypeScript** soit déjà installé.

```
C:\Users\NicolasLETHUILLIER>tsc -v  
Version 3.5.3
```

- Sinon :

```
npm install -g typescript
```

```
C:\Users\NicolasLETHUILLIER>npm install -g typescript  
C:\Users\NicolasLETHUILLIER\AppData\Roaming\npm\tsserver -> C:\Users\NicolasLETHUILLIER\AppData\Roaming\npm\typescript\bin\tsserver  
C:\Users\NicolasLETHUILLIER\AppData\Roaming\npm\tsc -> C:\Users\NicolasLETHUILLIER\AppData\Roaming\npm\typescript\bin\tsc  
+ typescript@3.5.3  
added 1 package from 1 contributor in 1.123s
```

9qqeq J b9ck986 fLOW J coufLipnfoL Jn J'TJ32



# TYPESCRIPT





- On l'a vu : **TypeScript** est compilé pour donner du **JavaScript**
- On peut donc utiliser, dans du code **TypeScript**, tout ce que l'on connaît en **JavaScript**
- Cependant, on veillera à respecter les bonnes pratiques et usages d'**ES6**



- Bons usages ES6 (1/3) :

## Fonctions fléchées

```
myArray.forEach(function(item) {  
  // Do something with item  
});
```



```
myArray.forEach(item => {  
  // Do something with item  
});
```

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



```
sum = (a, b) => a + b;
```





- Bons usages ES6 (2/3) :

### Egalité de types

```
"2" == 2
```

TRUE

Puisqu'on développe avec un langage fortement typé, on bannit ==

```
"2" === 2
```

FALSE

Je suis toujours censé savoir quel type je manipule

```
"2" != 2
```

FALSE

Si je dois comparer une chaîne et un entier, j'utilise des fonctions comme `parseInt()`

```
"2" !== 2
```

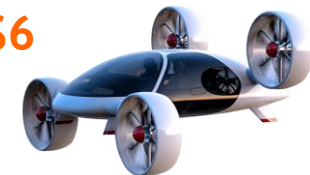
TRUE

- Bons usages ES6 (3/3) :

### Déclaration de variables



ES5



ES6

```
var counter = 12;

if (counter > 10) {
  var message = "C'est long...";

  if (counter < 20) {
    message = "ça va quand même.";
  }
}

console.log(message);
```

```
const counter = 12;

let message;

if (counter > 10) {
  message = "C'est long...";

  if (counter < 20) {
    message = "ça va quand même.";
  }
}

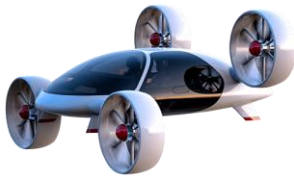
console.log(message);
```



- Bons usages ES6 (3/3) :

## Déclaration de variables

ES6



```
const counter = 12;

let message;

if (counter > 10) {
  message = "C'est long...";
}

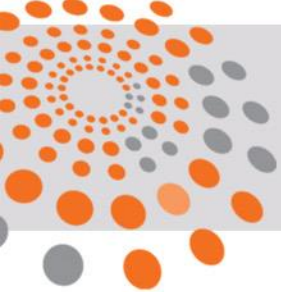
if (counter < 20) {
  message = "ça va quand même.";
}

console.log(message);
```

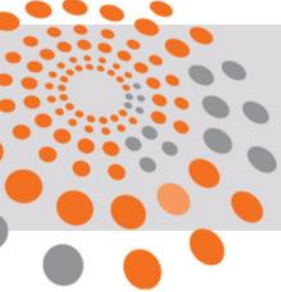
✗ **var** : à bannir  
*C'est le « fourre-tout » version Préhistoire*

➡ **const** : pour les constantes  
*On ne peut pas modifier la valeur après l'initialisation*

➡ **let** : pour les variables  
*On déclare une variable (avec éventuellement un type, voir ci-après)*



- La caractéristique principale de TypeScript est d'ajouter à JavaScript des **types**
- Jusqu'alors, JavaScript avait quelques types basiques (entiers, chaînes, etc.) mais faisait preuve de beaucoup de souplesse dans les opérations quotidiennes ; quant aux classes...
- Avec **TypeScript**, la rigolade, c'est terminé



- TypeScript met à disposition quelques types basiques :
  - `number`
  - `string`
  - `boolean`
  - `void`
  - `undefined`
- Ces types existent en JavaScript...  
mais on a rarement besoin de s'en préoccuper
- Les classes plus sophistiquées qui existent en JavaScript  
sont accessibles : `Date`, `Array`, etc.



- En TypeScript, il n'est pas obligatoire de typer ses variables quand on les déclare
  - Mais ce serait bête de s'en priver, non ?
- Exemples :

```
let value; // Sans type, ça marche, mais on ne sait pas ce que c'est  
let value: number; // Voilà : value est un nombre (entier ou décimal)  
let value = 12; // Lorsqu'on initialise la variable dès la déclaration, le type est facultatif, car déduit
```

- On peut également typer « à la volée » :

```
let human: { name: string, gender: string };  
let human = { name: "Roberto", gender: "male" };
```

Ici, **human** est typé

Ici aussi, déduit des propriétés



- **Objectif** : typer toutes les variables, y compris dans les fonctions

```
function formatTime(hours: number, minutes: number): string {  
    return hours + ':' + minutes;  
}
```

```
function createHuman(name: string): { name: string, age: number } {  
    // Miracle of birth  
    return { name, age: 0 };  
}
```

ES6 : il n'est pas nécessaire d'écrire  
**name: name**  
si le nom est le même



- On peut, en TypeScript, créer des classes, de la même manière qu'en C# ou en Java
- Les classes peuvent avoir :
  - Des attributs
  - Un ou plusieurs constructeurs
  - Des méthodes
- Pour pouvoir y accéder en-dehors du fichier de définition, on la préfixe avec le mot-clé « **export** »



- Exemple :

```
export class Human {  
    private _name: string;  
    get name(): string {  
        return this._name;  
    }  
    set name(value: string) {  
        this._name = value;  
    }  
    constructor(birthName: string) {  
        this.name = birthName;  
    }  
    public eat(food: Course[]): void {  
        // eat  
    }  
    public pray(): void {  
        // pray  
    }  
    public love(): void {  
        // love  
    }  
}
```

Un attribut avec ses getter et setter

Un constructeur

Une méthode avec un argument (typé)

D'autres méthodes...



- Exemple :

```
public eat(food: Course[]): void {  
    // eat  
}
```

Je ne peux pas appeler  
`eat()`

→ Too few parameters

Je ne peux pas appeler  
`eat([...], true)`

→ Too many parameters

Mais je souhaite que food soit facultatif  
Je lui donne donc une valeur par défaut :

```
public eat(food: Course[] = []): void {  
    // eat  
}
```

Ou je précise simplement  
que l'argument est facultatif

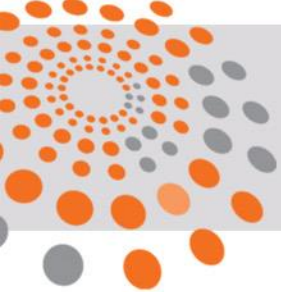
```
public eat(food?: Course[]): void {  
    // food peut ne pas être renseigné  
    // et sera undefined le cas échéant  
}
```

- Exemple :

```
function buildName(firstName: string, lastName = "Smith") {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"  
let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"  
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result4 = buildName("Bob", "Adams"); // ah, just right
```

```
function buildName(firstName = "Will", lastName: string) {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob"); // error, too few parameters  
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams"  
let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"
```

Source : <http://typescriptlang.org/docs/handbook/functions.html>



- Les arguments « indénombrables »

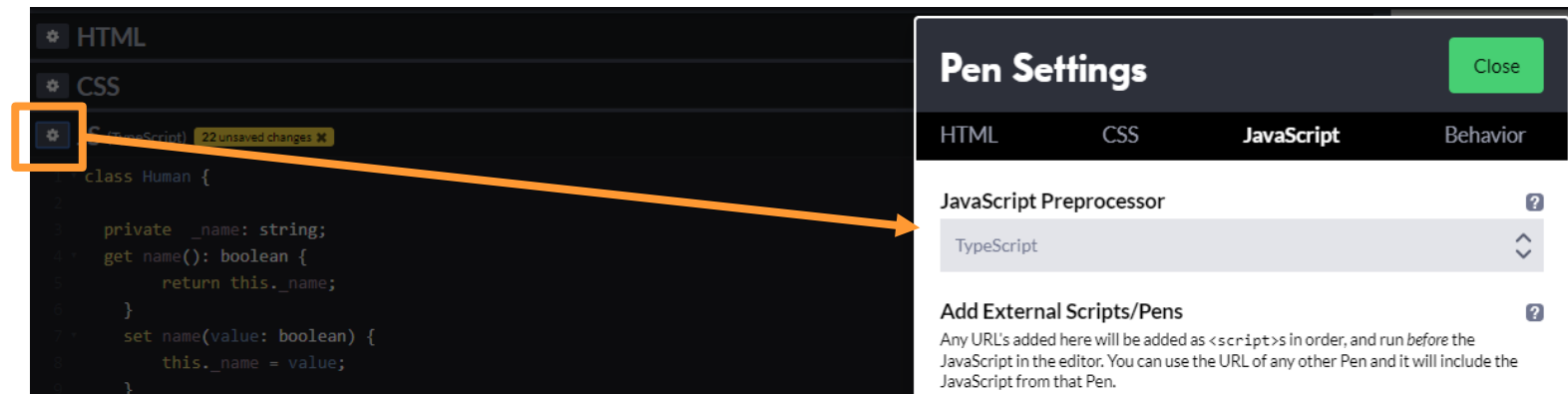
```
function buildName(firstName: string, ...restOfName: string[]) {  
    return firstName + " " + restOfName.join(" ");  
}  
  
// employeeName will be "Joseph Samuel Lucas MacKinzie"  
let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Source : <http://typescriptlang.org/docs/handbook/functions.html>

Avec « ... », on obtient un tableau  
qui contient l'ensemble des paramètres passés

- Premier exercice : nous allons créer une classe **Human** avec différents attributs et méthodes
- Pour cet exercice, nous utiliserons **CodePen** car il offre un environnement clé en main pour exécuter du TypeScript

<https://codepen.io/>





- Nous n'aurons pas besoin des volets HTML et CSS
- Dans le volet *JS (TypeScript)*, créer une classe **Human** avec :
  - Un attribut « name » privé avec un getter et un setter
  - Un constructeur qui prend une chaîne en paramètre et qui initialise le nom de cet humain
- Après le code de la classe, vous pouvez écrire du code pour tester votre classe

**C'est à vous**

```
"use strict";

class Human {

    private _name: string;
    get name(): string {
        return this._name;
    }
    set name(value: string) {
        this._name = value;
    }

    constructor(birthName: string) {
        this.name = birthName;
    }
}

const human = new Human("Homer");
document.write(human.name);
```



- Ajouter à **Human** un attribut privé **children** de type « tableau de **Human** », sans getter ni setter
- Créer la méthode publique **addChildren** qui prend en paramètre un nombre indéterminé de chaînes de caractères
  - Cette méthode créera, pour chaque nom passé, un **Human** et l'ajoutera à la liste des enfants
- Créer la méthode publique **getChildren** qui retourne une copie de la liste des enfants

**C'est à vous**



```
"use strict";

class Human {

    private _name: string;
    get name(): string {
        return this._name;
    }
    set name(value: string) {
        this._name = value;
    }

    private _children: Human[] = [];

    constructor(birthName: string) {
        this.name = birthName;
    }

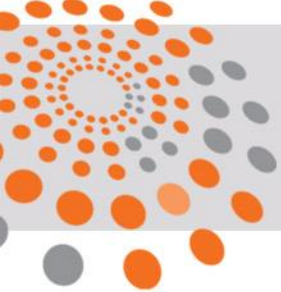
    public addChildren(...names: string[]) {
        names.forEach(name => this._children.push(new Human(name)));
    }

    public getChildren(): Human[] {
        return [...this._children];
    }
}
```



- Ajouter à **Human** une méthode publique **getChildrenHtml** qui retourne une chaîne de caractères contenant une liste HTML avec l'ensemble des noms des enfants
- Tester avec un code hors classe qui :
  - Crée un **Human** nommé *Homer*
  - Lui ajoute les enfants *Bart*, *Lisa* et *Maggie*
  - Affiche le nom d'*Homer*, puis la liste de ses enfants

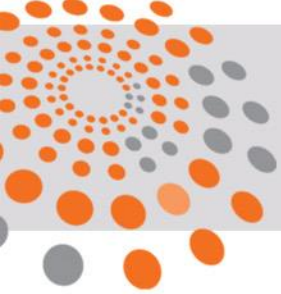
**C'est à vous**



```
public getChildrenHtml(): string {  
    return '<ul>' +  
        this.getChildren()  
            .map(child => '<li>' + child.name + '</li>')  
            .join('') +  
        '</ul>';  
}  
  
}  
  
const human = new Human("Homer");  
human.addChildren("Bart", "Lisa", "Maggie");  
document.write(human.name + human.getChildrenHtml());
```

Homer

- Bart
- Lisa
- Maggie

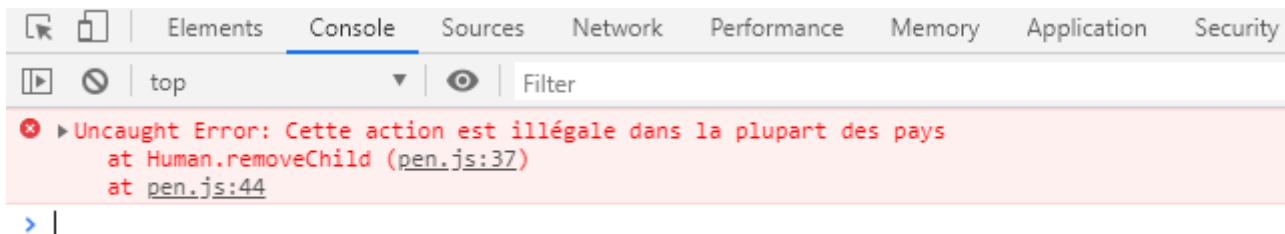


- Enfin, ajouter à Human une méthode publique **removeChildren**, qui prend un **Human** en paramètre, et qui lève une exception avec le message suivant :  
« *Cette action est illégale dans la plupart des pays* »
- Tester en appelant la méthode

**C'est à vous**

```
public removeChild(child: Human) {  
    throw new Error('Cette action est illégale dans la plupart des pays');  
}  
  
}  
  
const human = new Human("Homer");  
human.addChildren("Bart", "Lisa", "Maggie");  
document.write(human.name + human.getChildrenHtml());  
human.removeChild(human.getChildren()[0]);
```

F12





- On souhaite que l'erreur apparaisse à l'écran, et dénonce Homer
- Modifier **removeChild** et complétez son appel pour afficher, à l'écran, le message suivant :  
*« Homer a tenté d'assassiner Bart. Le commissaire Wiggum est sur le coup. »*

**C'est à vous**

```
public removeChild(child: Human) {  
    throw new Error(this.name + ' a tenté d\'assassiner ' + child.name + '. Le commissaire Wiggum est sur le coup.');
```

```
}  
  
const human = new Human("Homer");  
human.addChildren("Bart", "Lisa", "Maggie");  
document.write(human.name + human.getChildrenHtml());  
try {  
    human.removeChild(human.getChildren()[0]);  
} catch (e) {  
    document.write(e.message);  
}
```

Homer

- Bart
- Lisa
- Maggie

Homer a tenté d'assassiner Bart. Le commissaire Wiggum est sur le coup.



- On peut également écrire les chaînes de caractères avec des « backticks » (ou « backquote ») : `
- Cela rend parfois le code plus propre/plus clair
- L'important est de respecter les conventions établies sur votre projet

```
public removeChild(child: Human) {  
    throw new Error(`${this.name} a tenté d'assassiner ${child.name}. Le commissaire Wiggum est sur le coup.`);  
}
```



# TypeScript

Corrigé

```
"use strict";

class Human {

    private _name: string;
    get name(): string {
        return this._name;
    }
    set name(value: string) {
        this._name = value;
    }

    private _children: Human[] = [];

    constructor(birthName: string) {
        this.name = birthName;
    }

    public addChildren(...names: string[]) {
        names.forEach(name => this._children.push(new Human(name)));
    }

    public getChildren(): Human[] {
        return [...this._children];
    }

    public getChildrenHtml(): string {
        return '<ul>' +
            this.getChildren()
                .map(child => '<li>' + child.name + '</li>')
                .join('') +
            '</ul>';
    }

    public removeChild(child: Human) {
        throw new Error(`${this.name} a tenté d'assassiner ${child.name}. Le commissaire Wiggum est`);
    }
}

const human = new Human("Homer");
human.addChildren("Bart", "Lisa", "Maggie");
document.write(human.name + human.getChildrenHtml());
try {
    human.removeChild(human.getChildren()[0]);
} catch (e) {
    document.write(e.message);
}
```

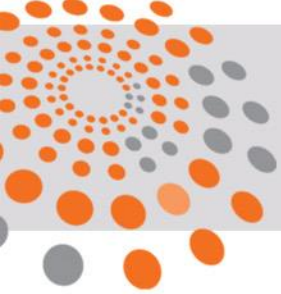


- Vous êtes désormais (quasiment) experts en TypeScript
- Il existe d'autres choses plus complexes, comme les **interfaces**, ou les **énumérations**... mais rien de dépayçant
- Nous pouvons maintenant nous lancer dans **Angular** !



# ANGULAR : INITIALISATION





- **Angular** est un framework JavaScript créé par Google
- **AngularJS** a été créé en 2009
- En 2016, Google publie la v2 d'AngularJS, si différente qu'elle est baptisée sobrement « **Angular** »



- **Angular** est donc un framework jeune, qui évolue beaucoup



Mai 2016



Mars 2017



Novembre 2017



Mai 2018



Octobre 2018

**Angular 8.0**



**Mai 2019**

<https://update.angular.io/>



- Première étape : créer un nouveau projet
- Pour ce faire, nous utiliserons angular-cli installé précédemment
- Angular-cli met à disposition différentes commandes, avec plein de paramètres bien documentés

<https://angular.io/cli/new>



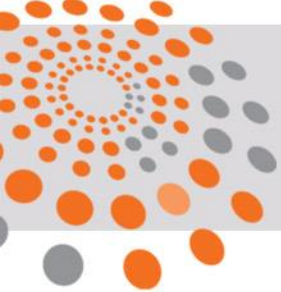
<https://angular.io/cli/new>

- `minimal`, `skipTests` : pour les projets perso  
« *Tester, c'est douter* »
- `routing` : pour de la navigation (Attention : `false` par défaut)
- `style` : pour l'extension des fichiers style (css, sass, scss...)



- Angular-cli permet également de compiler (build) un projet Angular
- On utilise alors la commande `ng build` avec différentes options
  - `baseHref` : pour déployer sur un sous-dossier
  - `configuration` : pour indiquer l'environnement cible



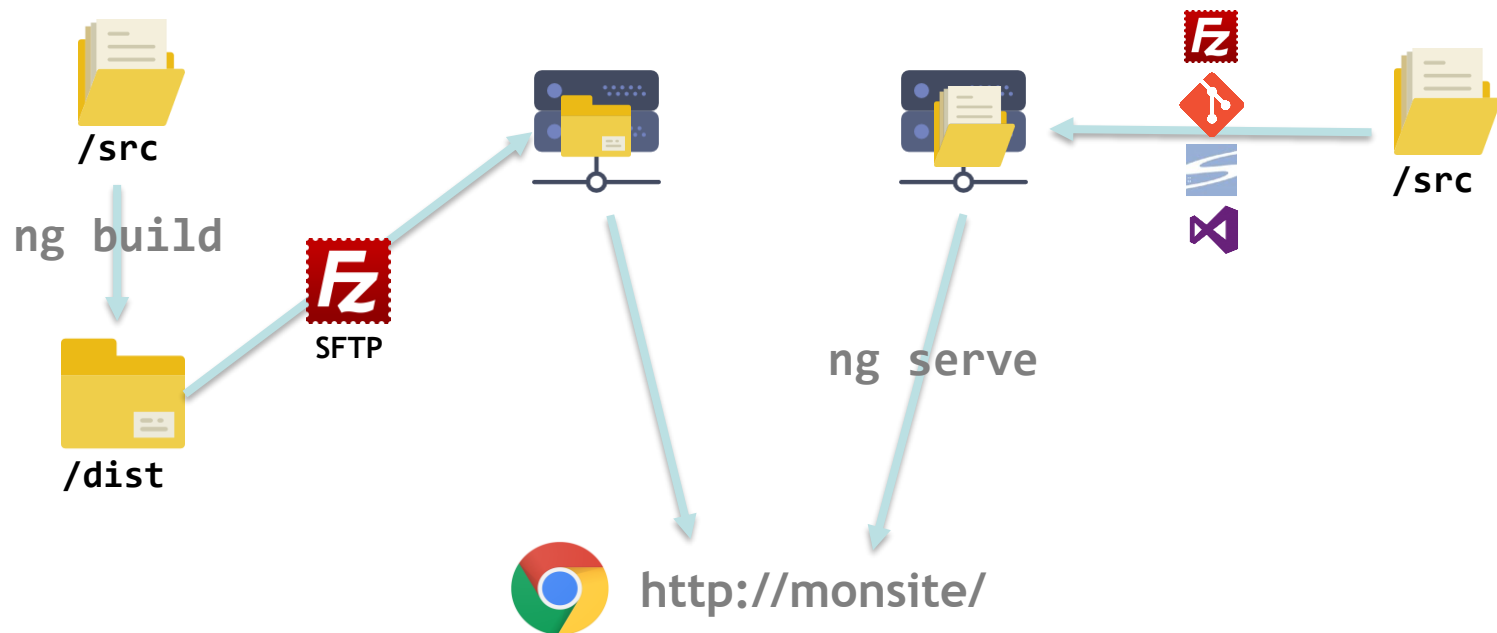


- Les sources ainsi compilées (en fichiers HTML, CSS, JS) se retrouvent dans un répertoire **dist** à la racine du projet
- Une fois le projet compilé, il suffit de transférer le contenu de ce répertoire **dist** sur un serveur web (Apache, tomcat, wamp...)
- De cette façon, il n'y a aucun prérequis à l'exécution d'une application Angular sur le serveur
  - *Puisque ce n'est que du HTML, CSS, JS*



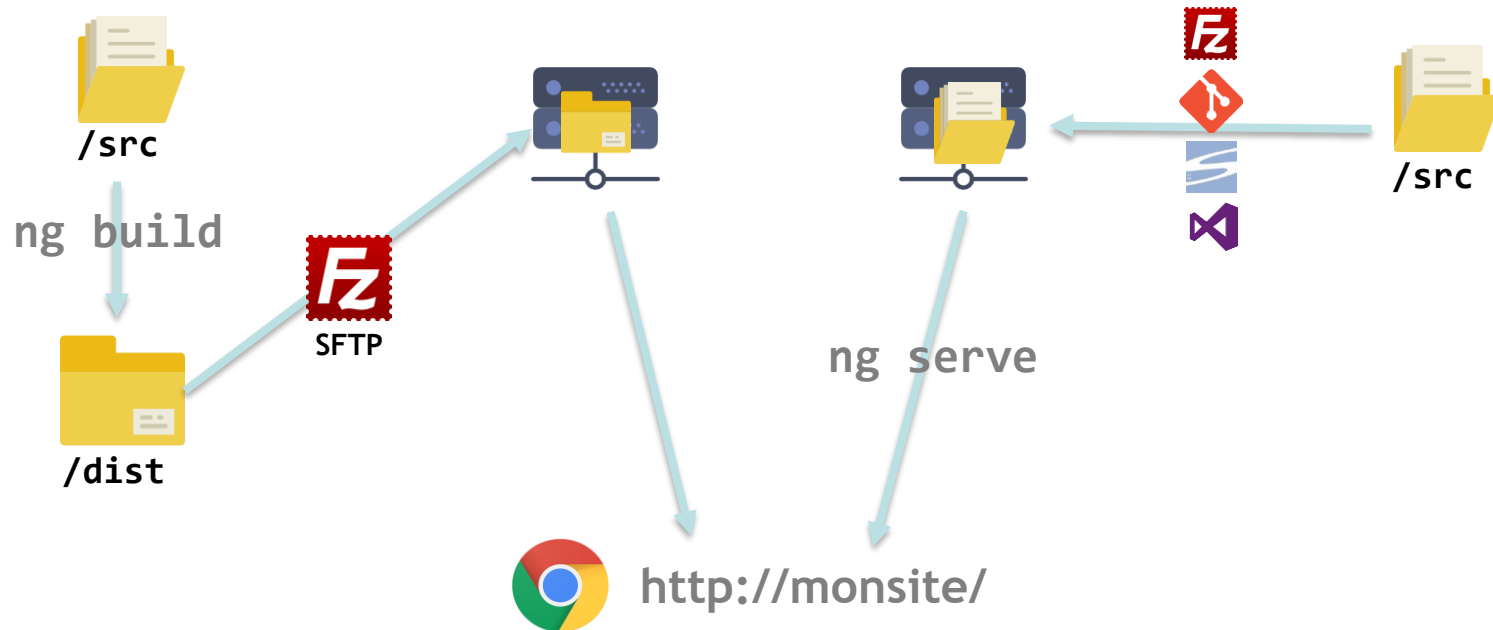
- Mais pour le développement, builder puis uploader serait un peu contraignant...
- Angular-cli intervient encore, avec la commande `ng serve`
- `ng serve` compile l'application (sans passer par le `dist`) puis l'exécute sur un serveur qui lui est propre (par défaut sur <http://localhost:4200>)
  - `baseHref`, `configuration` : voir déploiement
  - `liveReload` : recharge la page à chaque modification de fichier
  - `open` : ouvre le navigateur par défaut à l'adresse de l'application
  - `host`, `port` : pour configurer l'adresse de l'application

- Pour un déploiement externe (serveur de dev, qualif, prod...), on a deux options :



# Angular : initialisation

## Déploiement



Simple, ne nécessite aucune installation sur le serveur



Requiert des manipulations manuelles



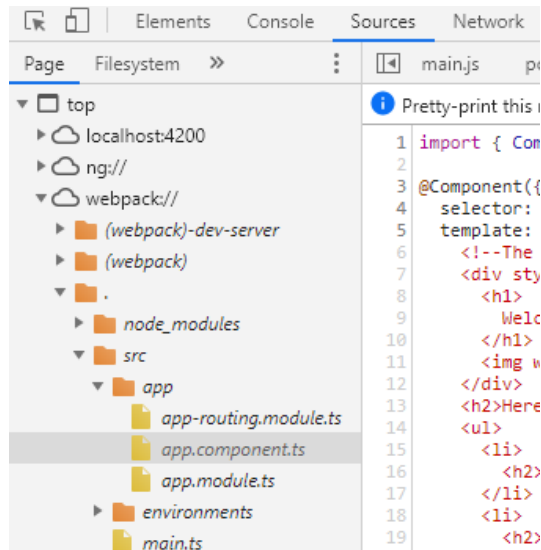
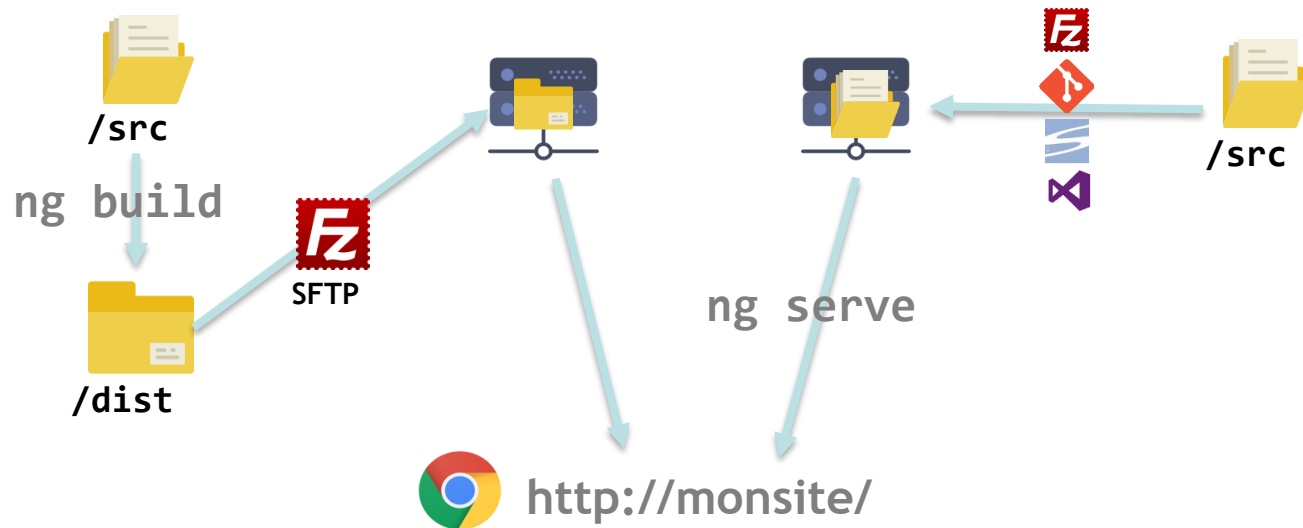
Automatisable, meilleure gestion de versions, intégration continue



Nécessite d'installer Node.js, angular-cli sur le serveur (voire Git, TFS, etc.)

# Angular : initialisation

Déploiement



Dans les deux cas  
(en fonction des options de build),  
Google Chrome permet de débogger  
le TypeScript grâce à du **SourceMapping**



# Angular : initialisation

Créons un premier projet !

- Ouvrir une invite de commandes et se placer dans le répertoire où vous souhaitez créer votre premier projet

```
C:\>  
λ cd C:\Users\NicolasLETHUILLIER\Documents\Formations\Web+angular  
  
C:\Users\NicolasLETHUILLIER\Documents\Formations\Web+angular  
λ ng new test --commit=false --minimal=true --routing=true --skipGit=true --skipTests=true --style=css|
```

- Le projet se crée et npm télécharge toutes les dépendances (répertoire **node\_modules** à la racine)



# Angular : initialisation

Créons un premier projet !

- Le projet est généré avec une page test
- On peut donc dès maintenant le lancer, et voir ce qui se passe

```
C:\Users\NicolasLETHUILLIER\Documents\Formations\Web+angular
λ cd test

C:\Users\NicolasLETHUILLIER\Documents\Formations\Web+angular\test
λ ng serve
10% building 3/3 modules 0 activei [wds]: Project is running at http://localhost:4200/webpack-dev-server/
i [wds]: webpack output is served from /
i [wds]: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 10.6 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 251 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.09 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 16.3 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.1 MB [initial] [rendered]
Date: 2019-08-22T12:38:23.410Z - hash: ac5f2940/e1541c97125 - Time: 9809ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
i [wds]: Compiled successfully.
```

```
i [mqw]: combyj6q znccezztjyl.
** Vu8nj9u g16e DeleJobwneuf 26L66u 12 j1zfeujtU8 ou j0c9jmozf:4500' obeu lonu plomzeu ou mffb:\\j0c9jmozf:4500\
D9f6: 501d-08-551T5:28:52.4T05 - H92u: 9C2f5d4016424Tc01453 - 1Twe: 880dwe
cmouk (Aeuqou) Aeuqou:12' Aeuqou:12:web (Aeuqou) 4:1 w8 [tutit9:] [Aeuqeleq]
```

# Angular : initialisation

Créons un premier projet !

**Welcome to test!**

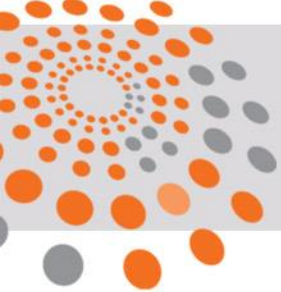


Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)







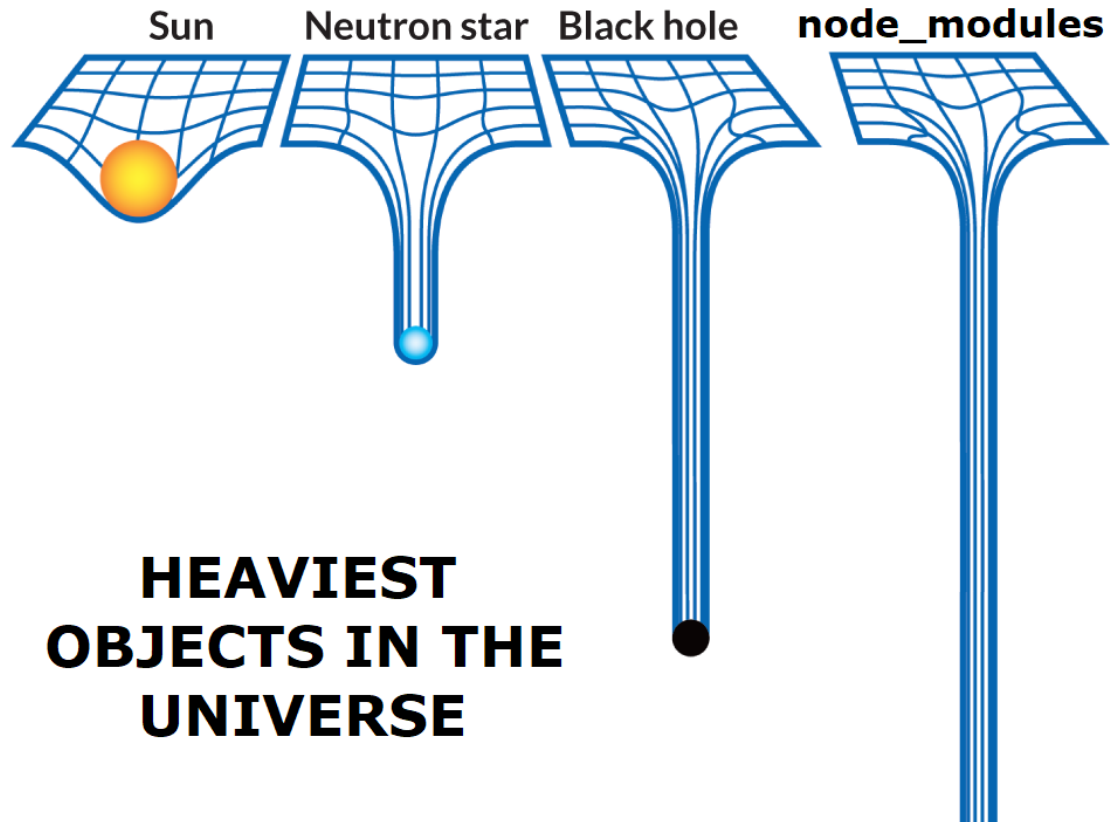
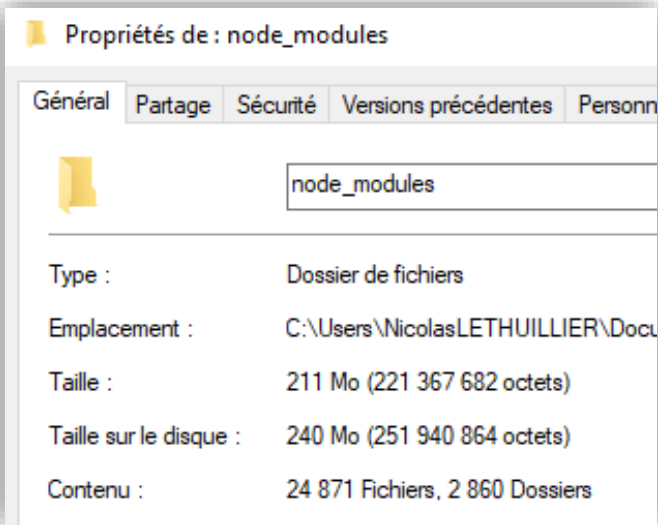
# Angular : initialisation

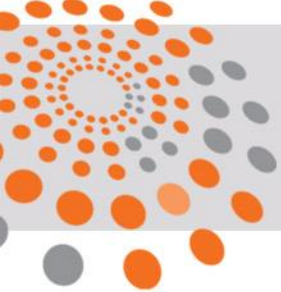
Créons un premier projet !

- Ouvrir la racine du projet avec votre IDE préféré  
Rappel : Visual Studio Code ♥
- Observons les fichiers à la racine...
  - **angular.json**
    - Configuration du projet, de build, des assets, etc.
  - **package.json**
    - Les commandes « raccourcies » npm
    - L'ensemble des dépendances de l'application  
Si on exécute un `npm install -s myLibrary`, elle viendra s'ajouter ici
  - **node\_modules**

# Angular : initialisation

Créons un premier projet !

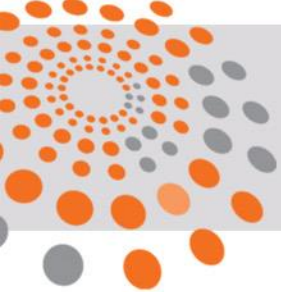




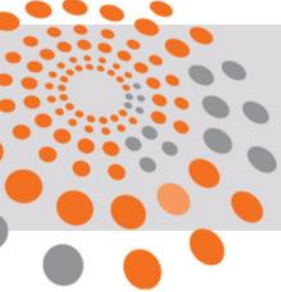
« *Il ne faut pas confondre Module et Module* »

Source : <http://www.learn-angular.fr/les-modules-angular/>

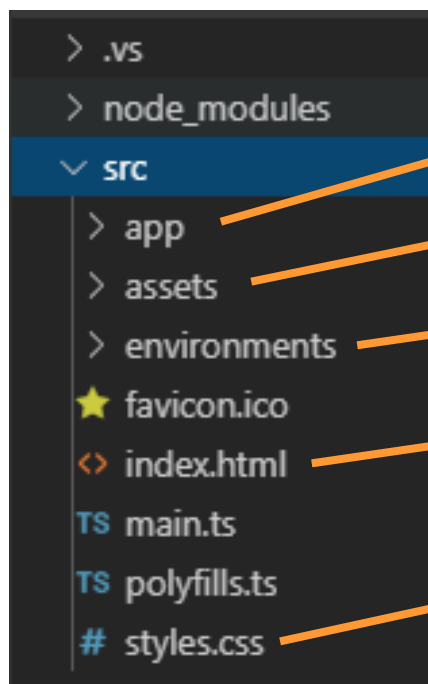
- Les modules au sens ES6
  - Ce qui est exporté et importé d'un élément à l'autre (mots-clés export, import)
- Les modules au sens Angular
  - Des classes « hub » qui regroupent l'ensemble des déclarations, imports de bibliothèques externes, etc. et indiquent le composant « de départ » de l'application



- Les modules au sens ES6
  - On retrouve notamment :
    - Les composants
    - Les services *à venir*
    - Les pipes *à venir*
    - Les directives *à venir*
- Les modules au sens Angular
  - On a en général **deux** modules
    - `AppModule` dans `app.module.ts`
    - `AppRoutingModule` dans `app-routing.module.ts`
  - Il arrive qu'on en crée d'autres pour découper l'application
    - Ex. : `admin.module.ts`, `backoffice.module.ts`



- Ouvrir le répertoire **src** du projet



Le code que vous écrivez

Les assets (images, fonts, etc.)

La configuration des environnements

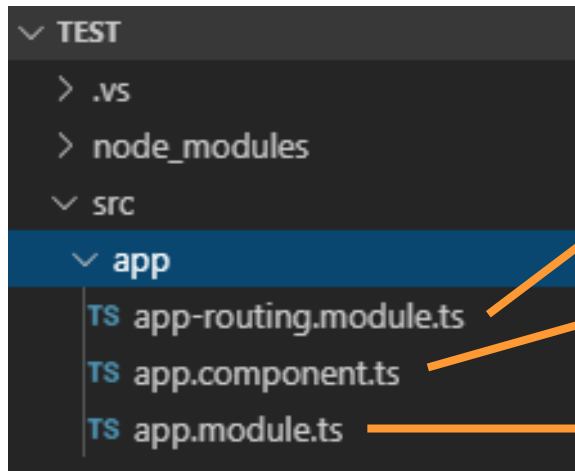
La page de base (que nous verrons plus tard)

La feuille de style qui s'appliquera  
à tous les composants et toutes les pages

## L'arborescence est libre

*Avec les fichiers de configuration,  
on peut changer cette organisation*

- Ouvrir le répertoire **app**



Les routes de l'application

Le premier composant généré

Le fameux **AppModule**

Dans une application Angular, les conventions veulent que chaque module porte son nom en extension :

```
*.component.ts / *.component.html / *.component.css  
    *.module.ts  
    *.service.ts  
    *.pipe.ts
```

- Ouvrir le fichier `app.module.ts`

Les composants,  
directives,  
pipes  
*les « vues », le front*

Les modules à importer  
*internes (routing) ou externes*

Les services  
*mis à disposition des composants*

Le composant de départ  
*la première page chargée*

```
TS app.module.ts •
src > app > TS app.module.ts > ...
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6
7  @NgModule({
8    declarations: [
9      AppComponent
10   ],
11   imports: [
12     BrowserModule,
13     AppRoutingModule
14   ],
15   providers: [],
16   bootstrap: [AppComponent]
17 })
18 export class AppModule { }
19
20
```



- On ne regarde pas le fichier de routing pour l'instant car il est vide, ce ne serait pas très parlant
- Maintenant qu'on a vu toute la mécanique d'Angular, on va s'intéresser aux **composants**, pour pouvoir créer des pages, des blocs, etc.
- Des questions ?

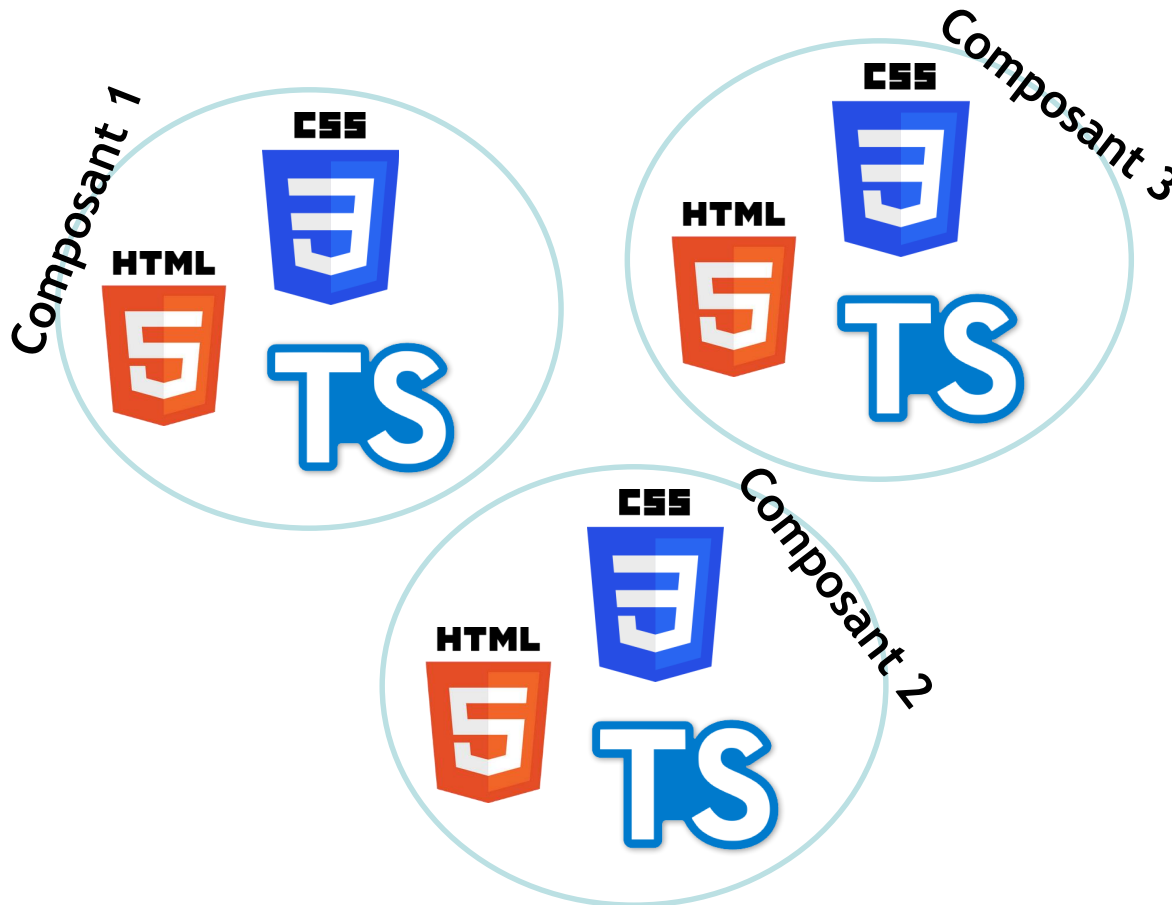




# ANGULAR : COMPOSANTS



- Pour rappel, une application web, c'est :

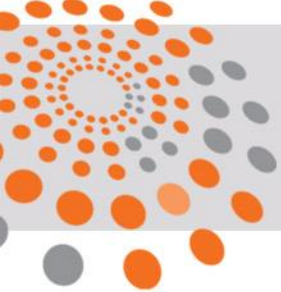


# TS

Fichiers TypeScript

# HTML

index.html



# Angular : composants

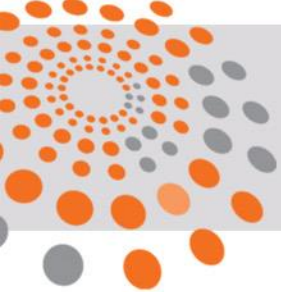
Racine de l'application

- Ouvrir le fichier `index.html` à la racine du projet

```
<> index.html x
src > <> index.html > ...
1  <!doctype html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>Test</title>
6    <base href="/">
7
8    <meta name="viewport" content="width=device-width, initial-scale=1">
9    <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
11 <body>
12   <app-root></app-root>
13 </body>
14 </html>
15
```

Le sous-dossier éventuel  
d'exécution de l'app

La racine de l'app  
→ là où le composant  
« *bootstrap* » va venir  
s'insérer au démarrage



# Angular : composants

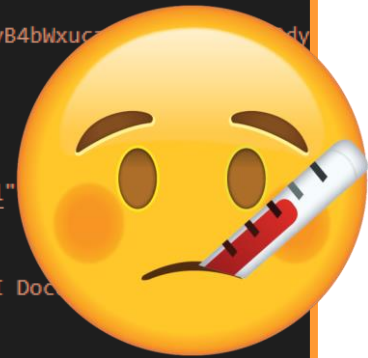
## Premier composant

- C'est `AppComponent`, qui était renseigné en tant que `bootstrap` dans le `AppModule`
- C'est donc ce composant qui est inséré dans le `app-root`
- Ouvrir `app.component.ts`

# Angular : composants

## Premier composant

```
src > app > TS app.component.ts > ...
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    template: `
6      <!--The content below is only a placeholder and can be replaced.-->
7      <div style="text-align:center">
8        <h1>
9          Welcome to {{title}}!
10        </h1>
11        
12      </div>
13      <h2>Here are some links to help you start: </h2>
14      <ul>
15        <li>
16          <a target="_blank" rel="noopener" href="https://angular.io/tutorial">
17            <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Doc</a>
18          </li>
19          <li>
20            <a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></li>
21          </li>
22        </ul>
23      </li>
24      </ul>
25      <router-outlet></router-outlet>
26    `
27    ,
28    styles: []
29  })
30  export class AppComponent {
31    title = 'test';
32  }
```





- Commençons par faire un peu de ménage, voulez-vous ?
- Dans le répertoire `app`, créer les fichiers :
  - `app.component.html`
  - `app.component.css`
- Copier le HTML de `app.component.ts` (ce qui est entre les quotes) dans `app.component.html`
- Remplacer « `template:` » par « `templateUrl:` » et indiquez le chemin du fichier
- Remplacer « `styles: []` » par « `styleUrls:` » en indiquant le chemin de la feuille de style (dans un tableau !)



# Angular : composants

## Premier composant

- Voilà à quoi ressemble, classiquement, un composant :

```
<> index.html TS app.component.ts ●
src > app > TS app.component.ts > ...
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.css']
7  })
8  export class AppComponent {
9    title = 'test';
10
11    // Constructors, methods, etc.
12  }
13
```



# Angular : composants

## Premier composant

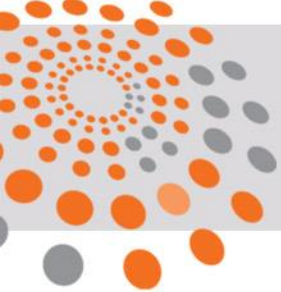
- Observons le fichier HTML du composant...

```
<> index.html    TS app.component.ts    <> app.component.html
src > app > <> app.component.html > ...
 1  <!--The content below is only a placeholder and can be replaced.-->
 2  <div style="text-align:center">
 3      <h1>
 4          Welcome to {{title}}!
 5      </h1>
 6      Tour of Heroes</a></h2>
12      </li>
13      <li>
14          <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Documentation</a></h2>
15      </li>
16      <li>
17          <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
18      </li>
19  </ul>
20  <router-outlet></router-outlet>
21
```

Appel à un attribut de la classe TS

Entrée du routing





# Angular : composants

## Binding (1/2) : les bases

- {{ ... }} permet d'utiliser un attribut (public) de la classe TS

```
1 <!--The content below is only a placeholder and can be replaced.-->
2 <div style="text-align:center">
3   <h1>
4     Welcome to {{title}}!
5   </h1>
6   
7 </div>
```

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

  // Constructors, methods, etc.
}
```

Dans les accolades, on peut même écrire du JavaScript



```
<div style="text-align:center">
  <h1>
    Welcome to {{title.toUpperCase()}}!
  </h1>
  
3      <h1>
4      | Welcome to {{getTitleFromDatabase()}}!
5      </h1>
6      
  <h1>
    Welcome to {{title}}!
  </h1>
  <img width="{{customWidth}}" alt="Angular Logo" sr
</div>
<h2>Here are some links to help you start: </h2>
<ul>
```

- Qu'on peut écrire également :

```
<!--The content below is only a placeholder and can be
<div style="text-align:center">
  <h1>
    Welcome to {{title}}!
  </h1>
  <img [width]="customWidth" alt="Angular Logo" src=
</div>
<h2>Here are some links to help you start: </h2>
<ul>
```

attr='{{ ... }}'

et

[attr]='...'

sont strictement équivalents

- Nous parlerons plus tard des **directives**, mais trois d'entre elles nous intéressent déjà :
  - \*ngIf
  - \*ngFor
  - \*ngSwitch
- Ces termes se placent dans les balises HTML pour ajouter de la mécanique à l'affichage

```
<h1 *ngIf="title">
  Welcome to {{title}}!
</h1>
```

```
<ul>
  <li *ngFor="let link of links">{{link.url}}</li>
</ul>
```

Attribut de la classe

# Angular : composants

ngIf, ngFor, ngSwitch

```
<h1 *ngIf="title">
  Welcome to {{title}}!
</h1>
```

```
<ul>
  <li *ngFor="let link of links">{{link.url}}</li>
</ul>
```

- Attention : un **\*ngIf** et un **\*ngFor** ne peuvent pas se trouver sur la même balise
- Pour pallier cette lacune, il existe la balise **ng-container**

```
<ng-container *ngIf="canDisplayTiles()">
  <div *ngFor="let tile of tiles" [width]="tile.width" [height]="tile.height">{{tile.name}}</div>
</ng-container>
```

- **ng-container** n'apparaît pas dans le HTML final et n'a aucune incidence sur le DOM



- Il est temps de créer notre premier composant !
- Pour ce faire, deux méthodes :
  - A la main, en créant les fichiers, en les remplissant *from scratch* et en mettant à jour le **AppModule**  
*Permet de bien comprendre la mécanique, mais un peu fastidieux*
  - Avec angular-cli : **ng generate component 'monComposant'**
- angular-cli crée automatiquement le composant dans **/app** suivi de l'arborescence éventuellement précisée, avec les fichiers TS, HTML et CSS qui vont bien puis ajoute le composant à l'**AppModule**



- Utilisons donc la méthode la plus simple et créons un composant **custom-button**

**ng generate component custom-button**

- Et allons voir un peu à quoi ça ressemble...

custom-button.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

**<p>custom-button works!</p>**

custom-button.component.html

```
import { AppComponent } from './app.component';
import { CustomButtonComponent } from './custom-button.component';

@NgModule({
  declarations: [
    AppComponent,
    CustomButtonComponent
  ],
  imports: [
```

app-module.ts

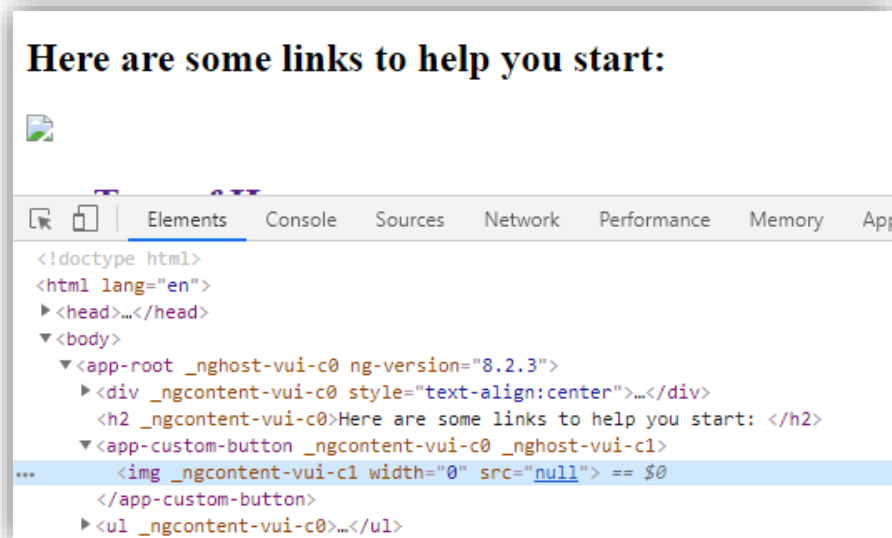


- On peut dès à présent utiliser notre nouveau composant dans un autre composant, en utilisant le selector défini dans le décorateur (ici « `app-custom-button` »)
- Ouvrir `app.component.html`
- Ajouter quelque part :  
`<app-custom-button></app-custom-button>`
- Tester





- Modifier le composant **custom-button** pour lui ajouter :
  - Un attribut « **width** » public de type nombre
  - Un attribut « **source** » public de type chaîne de caractères
- Modifier le template de **custom-button** pour y mettre une image dont la largeur sera **width** et la source **source**
- Tester





- Initialiser `width` et `source` avec les valeurs de votre choix
- Constater le changement
- Corsons un peu la chose :
  - Créer un attribut public `links` qui sera une liste d'objets avec pour propriétés « `title` » et « `target` »
  - Créer une méthode publique `generateLinks` qui remplit cette liste avec des liens quelconques
  - Dans le template (HTML), ajouter une liste qui ne s'affiche que lorsque `links` existe et possède des éléments (`*ngIf`), et qui contient autant d'items avec des liens qu'il y a d'éléments dans `links` (`*ngFor`)
  - Ajouter à l'image `(click)='generateLinks()'`
  - Tester !

# Angular : composants

Corrigé

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

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

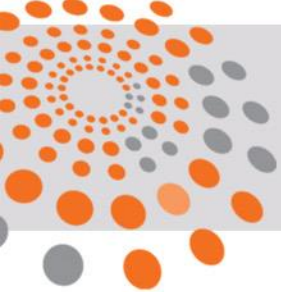
  public width = 100;
  public source = 'https://interactive-examples.mdn.mozilla.net';
  public links: {title: string, target: string}[];

  constructor() { }

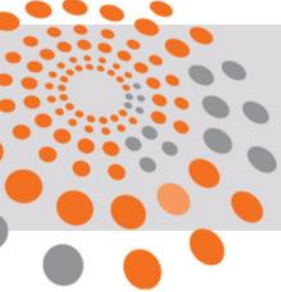
  ngOnInit() {
  }

  public generateLinks() {
    this.links = [
      { title: 'Google', target: 'http://www.google.fr'},
      { title: 'Facebook', target: 'http://www.facebook.fr'},
      { title: 'Twitter', target: 'http://www.twitter.fr'},
    ];
  }
}
```

```
<img [width]="width" [src]="source" (click)="generateLinks()" />
<ul *ngIf="links">
  <li *ngFor="let link of links"><a [href]="link.target">{{link.title}}</a></li>
</ul>
```



- Le composant qui appelle `CustomButton` peut lui passer des paramètres grâce au décorateur `@Input`
- Ajoutez à la classe `CustomButtonComponent` un attribut public `imgTitle` de type chaîne de caractères
- Préfixez-le du décorateur `@Input()`
- Dans le template, utilisez `imgTitle` comme valeur de l'attribut `title` de l'image
- Modifiez `app.component.html` pour donner une valeur à `imgTitle`
- Testez !



# Angular : composants

@Input()

custom-button.component.ts

```
@Input() public imgTitle: string;
```

custom-button.component.html

```
<img [width]="width" [src]="source" [title]="imgTitle" (click)="generateLinks()" />
```

app.component.html

```
<app-custom-button imgTitle="Titre de l'image au survol"></app-custom-button>
```



# Angular : composants

Bonus : un peu de style

- Sur une image, on peut utiliser l'attribut `width`... mais on souhaite modifier tout le CSS, sur n'importe quoi
- Méthode naïve :

```
<div [style]='width: ' + (2 * nbColumns) + 'px'></div>
```

- Méthode Angular :

```
<div [style.width.px]='2 * nbColumns'></div>
```

- Cela génère un attribut `style='width: 240px'`  
On peut en mettre autant qu'on veut,  
et avec toutes les propriétés CSS qu'on veut



# Angular : composants

## Binding (2/2) : la banane dans la boîte

- Reprenons cette ligne :

```
<img [width]="width" [src]="source" (click)="generateLinks()" />
```

- On voit des [crochets] qui vont chercher une information dans la partie TypeScript du composant
- On voit des (parenthèses) qui envoient de l'information vers la partie TypeScript du composant
- Ce n'est pas plus compliqué que ça :
  - Une donnée entrant dans le HTML est entre [crochets]
  - Une donnée sortante est entre (parenthèses)
- Vous l'avez compris : tous les événements (click, keyup, mousedown, etc.) seront entre (parenthèses)



# Angular : composants

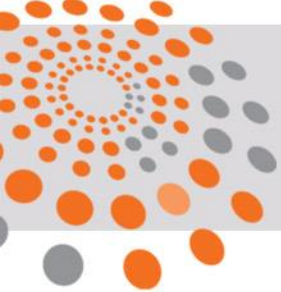
## Binding (2/2) : la banane dans la boîte

- Parfois, on veut que la donnée circule librement entre le HTML et le TypeScript
- Par exemple :
  - On souhaite **binder les champs** d'un formulaire de façon à ce qu'une saisie change la donnée dans le TS, mais qu'une opération dans le TS change le contenu du champ
- On utilise alors **[(ngModel)]='myAttribute'**
  - C'est-à-dire qu'on a à la fois les crochets et les parenthèses

```
<input type="text" name="lastname" [(ngModel)]="lastname" />
```

- Moyen mnémotechnique : la banane dans la boîte !





- Une autre méthode permet d'impacter le HTML avec le TS

```

```

```
@ViewChild('myImage', {static: true}) image: ElementRef;
```

```
@ViewChild(CustomButtonComponent, {static: true}) myButton: CustomButtonComponent;
```

- Mais cette méthode n'est que rarement utilisée, car elle bypass la notion de binding (VanillaJS)



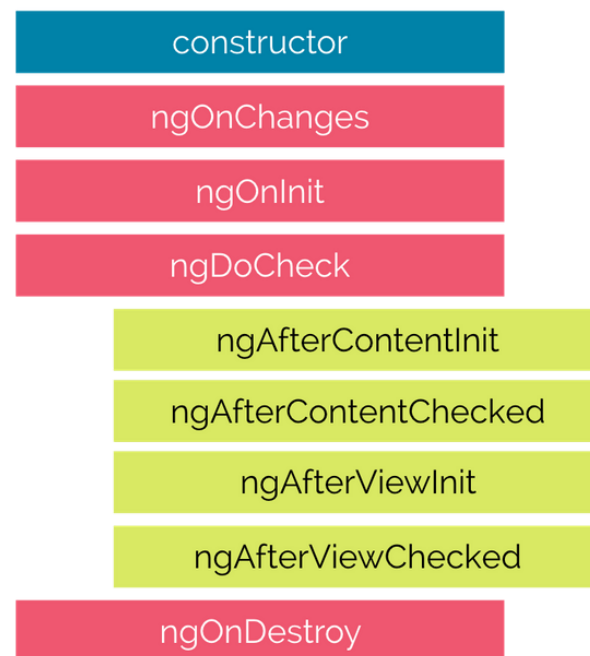
- Tous les composants (et les autres modules) ont un cycle de vie

Lorsqu'on appelle un composant, son constructeur est appelé

S'en suit un tas d'étapes, qui peuvent se répéter en cas de changement

Lorsqu'on change de page, que le composant n'est plus utilisé, il est détruit

- Les méthodes ici présentées sont appelées « *lifecycle hooks* » car elles sont appelées aux différentes étapes de la vie du composant



Source : <https://codecraft.tv/courses/angular/components/lifecycle-hooks/>

- Si on reprend notre composant généré à l'aide d'angular-cli

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

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

  public width = 100;
  public source = 'https://interactive-examples.mdn.mozilla.net/';
  public links: {title: string, target: string}[];

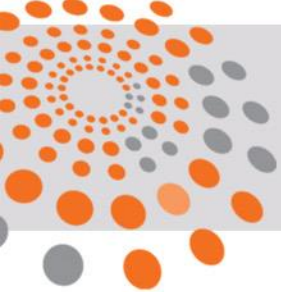
  constructor() { }

  ngOnInit() {
  }
```

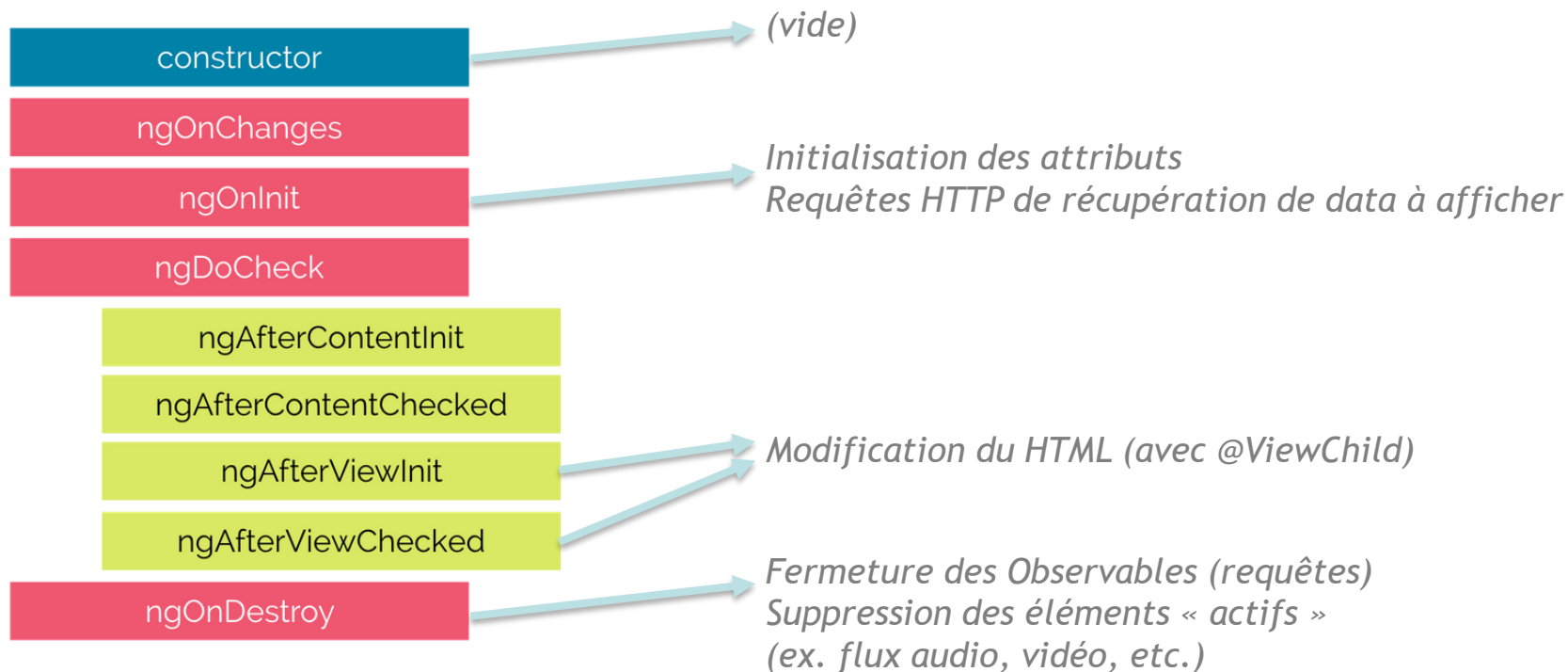
Le corps de cette fonction sera exécuté à l'« initialisation » du composant (voir schéma)

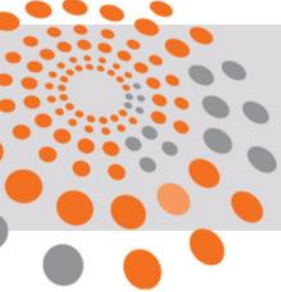
Pour que ngOnInit soit appelée, on doit implémenter l'interface OnInit

De manière générale, pour utiliser un hook, il faut implémenter l'interface associée... et on peut le faire avec autant de hook que souhaité !



- Exemple d'utilisation :





- Les composants sont des bouts de HTML autonomes
- `AppModule` est le « hub » qui déclare tout
- Les composants sont constitués d'1 HTML, 1 CSS, 1 TS
- `{{myAttr}}` pour aller chercher un attribut
- `[src]='myUrl'` pour binder l'attribut `myUrl` (*one-way*)
- `(click)='action()'` pour binder la méthode sur l'événement
- `[(ngModel)]='myAttr'` pour faire du binding *two-way*
- Les lifecycle hooks permettent de déclencher des actions à des moments de la vie d'un composant



# ANGULAR : ROUTING



- Ouvrir `app.component.html`

```
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
<router-outlet></router-outlet>
```

A la navigation, le composant cible vient s'insérer à l'endroit du router-outlet



- Commençons par supprimer la balise `app-custom-button` du composant `App`
- Nous allons l'intégrer à l'aide de la navigation
- Ouvrir `app-routing.module.ts`

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

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



- Pour une fois, Angular ne nous fournit pas d'exemple...  
voici donc comment ça fonctionne :

```
import { NgModule }           from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { DashboardComponent }  from '../dashboard/dashboard.component';
import { HeroesComponent }     from '../heroes/heroes.component';
import { HeroDetailComponent } from '../hero-detail/hero-detail.component';

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];

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

**path:** le chemin dans l'URL

**component:** le composant à charger

Ainsi, si je vais à /heroes,  
c'est le composant HeroesComponent  
qui sera inséré dans le router-outlet

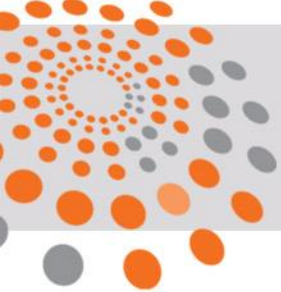


- Adaptons donc `app-routing.module.ts` pour que l'adresse <http://localhost:4200/button> conduise au composant `CustomButton`

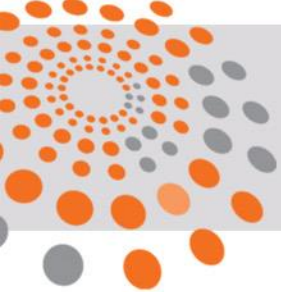
```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CustomButtonComponent } from '../custom-button/custom-button.component';

const routes: Routes = [
  { path: 'button', component: CustomButtonComponent }
];

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



- On teste en allant sur <http://localhost:4200/button>
- En général, on a au moins un **path**: `''` qui conduit au composant d'accueil
- **path**:  `'**'`  permet de spécifier un comportement par défaut si aucune route ne matche avec la demande de l'utilisateur
  - *→ Page 404*



- On souhaite un lien qui nous permettrait d'accéder à cette page sans saisir l'URL (c'est mieux !)
- Ouvrir `app.component.html`
- Ajouter à la liste des liens un lien vers notre « page »
  - Pour cibler une « page » de l'application, on n'utilise pas `href`, mais `routerLink`
- On voit alors que la page ne s'actualise pas : le principe des *Single Page Application* !

# Angular : routing

## Go routing avec des paramètres

```
import { NgModule }           from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { DashboardComponent } from '../dashboard/dashboard.component';
import { HeroesComponent }    from '../heroes/heroes.component';
import { HeroDetailComponent } from '../hero-detail/hero-detail.component';

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];

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

export class AppRoutingModule {}
```

Sur cet exemple du tuto officiel, l'URL `detail` accepte un paramètre tel que `.../detail/452` pour afficher les infos du héros d'ID = 452

Pour passer un tel paramètre via le `routerLink`, on adapte l'URL

```
<a routerLink="/detail/{{hero.id}}">
```

```
<a [routerLink]="'/detail/' + hero.id"></a>
```



# Angular : routing

## Nested routes

- Il est possible de définir des « sous-routes »

```
const routes: Routes = [  
  {path: '', redirectTo: 'home', pathMatch: 'full'},  
  {path: 'find', redirectTo: 'search'},  
  {path: 'home', component: HomeComponent},  
  {path: 'search', component: SearchComponent},  
  {  
    path: 'artist/:artistId',  
    component: ArtistComponent,  
    children: [  
      {path: '', redirectTo: 'tracks'}, (1)  
      {path: 'tracks', component: ArtistTrackListComponent}, (2)  
      {path: 'albums', component: ArtistAlbumListComponent}, (3)  
    ]  
  },  
  {path: '**', component: HomeComponent}  
];
```

Source : <https://codecraft.tv/courses/angular/routing/nested-routes/>



- Le routing dans Angular est très simple, configuré dans un unique fichier (`app-routing.module.ts`), et facile à maintenir
- La navigation se fait à l'aide de `routerLink`
- Pour des questions d'accessibilité, on positionnera toujours les `routerLink` sur des liens `<a>`
- Vous manipulerez beaucoup ce genre de choses dans **Tour of Heroes**

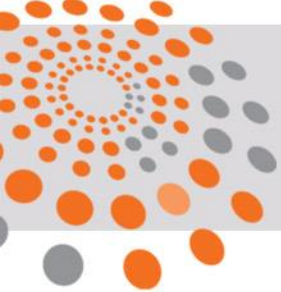


# TOUR OF HEROES

## (1/2)







- **Tour of Heroes** est un tutoriel très complet sur les différentes fonctionnalités et bonnes pratiques d'Angular
- Il a été créé et est maintenu par Google
  - Donc a priori, ça tient la route... et c'est à jour
- C'est un passage obligé pour tous les développeurs Angular



<https://angular.io/tutorial>

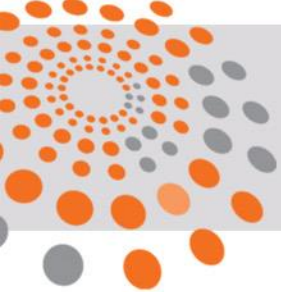
- Dans **Tour of Heroes**, vous allez créer une petite application de gestion de super-héros
- Vous allez utiliser les directives courantes, faire du binding, du routing, etc.
- Vous ferez également un formulaire simple
- Enfin, vous créerez des **services**, avant que je vous en parle en français
- Arrêtez-vous avant le « **6. HTTP** », nous reprendrons plus tard



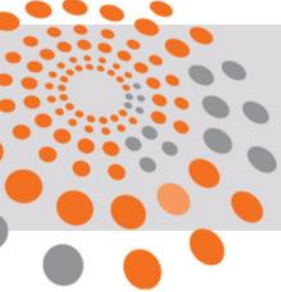
# OUTILS AVANCÉS

SERVICES, PIPES, DIRECTIVES, EVENTEMITTER





- On a vu comment créer des composants
- Dans Tour of Heroes, vous avez également créé un **service** (**HeroService**), qui récupérerait les données d'une fausse base
- Nous allons voir comment créer et à quoi servent :
  - Les services
  - Les pipes
  - Les directives



- Les services contiennent :
  - Le code métier
  - Les appels externes (AJAX, etc.)
- De manière générale, on part du principe que :
  - Les **composants** sont la vue, ils gèrent l'affichage
  - Toute la mécanique autre doit être dans un **service**
- L'agencement des services (nombre, découpage, etc.) est libre et doit être rationalisé



- Dans **HeroService**, il n'y a qu'une méthode :
  - `getHeroes(): Observable<Hero[]>`
- Cette méthode récupère « en base » les héros, et les retourne sous la forme d'un Observable (cf. plus tard)
- Le composant va appeler cette méthode pour pouvoir disposer des héros et en faire ce qu'il veut



- Vous l'avez vu : lorsqu'un module quelconque (composant, service, etc.) souhaite utiliser un service, il le déclare en **argument de son constructeur**

```
constructor(private heroService: HeroService) { }
```

- On peut en déclarer autant que nécessaire

```
constructor(  
  private routeService: Route,  
  private heroService: HeroService,  
  private machinService: MachinService  
) { }
```



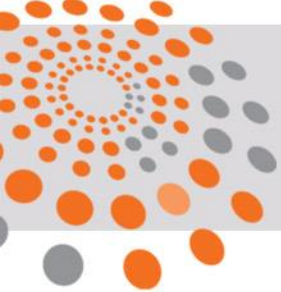
- Les pipes sont des éléments Angular qui permettent de raffiner l’affichage d’une donnée
- Concrètement, c’est ce qui va nous permettre, par exemple, de transformer :

**2019-11-27T09:28:45**

en :

**Mercredi 27 septembre, 09h28**





- Il existe plusieurs pipes natifs Angular, parmi lesquels :
  - **Date** : pour afficher une date selon un format paramètre
  - **Uppercase** : pour afficher la chaîne en capitales
  - **Percent** : pour afficher un pourcentage
  - **Currency** : pour afficher un montant
  - etc.

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

- Un pipe s'utilise... grâce au symbole pipe (|)
- Ses paramètres éventuels sont ajoutés à la suite précédés de « : »



```
{{ dateObj | date }}           // output is 'Jun 15, 2015'  
{{ dateObj | date:'medium' }}  // output is 'Jun 15, 2015, 9:43:11 PM'  
{{ dateObj | date:'shortTime' }} // output is '9:43 PM'  
{{ dateObj | date:'mm:ss' }}    // output is '43:11'
```

```
<!--output '26%'-->  
<p>A: {{a | percent}}</p>  
  
<!--output '0,134.950%'-->  
<p>B: {{b | percent:'4.3-5'}}</p>  
  
<!--output '0 134,950 %'-->  
<p>B: {{b | percent:'4.3-5':'fr'}}</p>
```

```
<!--output '$0.26'-->  
<p>A: {{a | currency}}</p>  
  
<!--output 'CA$0.26'-->  
<p>A: {{a | currency:'CAD'}}</p>  
  
<!--output 'CAD0.26'-->  
<p>A: {{a | currency:'CAD':'code'}}</p>  
  
<!--output 'CA$0,001.35'-->  
<p>B: {{b | currency:'CAD':'symbol':'4.2-2'}}</p>
```

- On peut bien sûr créer des pipes personnalisés

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

/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
 */
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent?: number): number {
    return Math.pow(value, isNaN(exponent) ? 1 : exponent);
  }
}
```

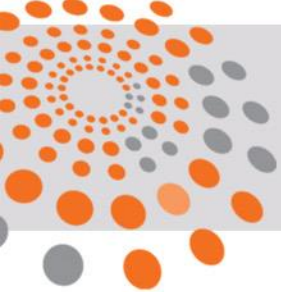
On met le décorateur @Pipe

On implémente PipeTransform

On écrit la méthode **transform** qui prend en entrée :

- La valeur « pipée »
  - Les éventuels paramètres
- et qui retourne la **valeur transformée**

Le type de retour est **personnalisable**

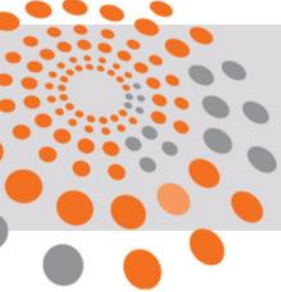


- On peut, dès lors (et une fois déclaré dans l'**AppModule**), l'utiliser dans nos templates :

```
<h2>Power Booster</h2>  
<p>Super power boost: {{2 | exponentialStrength: 10}}</p>
```

### **Power Booster**

Super power boost: 1024



- Les **directives** sont des mots-clés que l'on ajoute aux balises HTML pour spécifier des comportements du DOM
- **\*ngIf**, **\*ngFor**, **\*ngSwitch** sont des **directives structurelles** (« *structural directives* »)
- On peut créer des **directives attributs** (« *attribute directives* »)

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

On utilise le décorateur **@Directive**

Le code exécuté sera cette fois  
situé dans le constructeur

### highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

### my-comp.component.html

```
<p appHighlight>Highlight me!</p>
```

L'élément HTML sur lequel  
la directive est placée

Highlight me!

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @Input('appHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'red');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

```
<h1>My First Attribute Directive</h1>
```

```
<h4>Pick a highlight color</h4>
```

```
<div>
```

```
  <input type="radio" name="colors" (click)="color='lightgreen'">Green
```

```
  <input type="radio" name="colors" (click)="color='yellow'">Yellow
```

```
  <input type="radio" name="colors" (click)="color='cyan'">Cyan
```

```
</div>
```

```
<p [appHighlight]="color">Highlight me!</p>
```

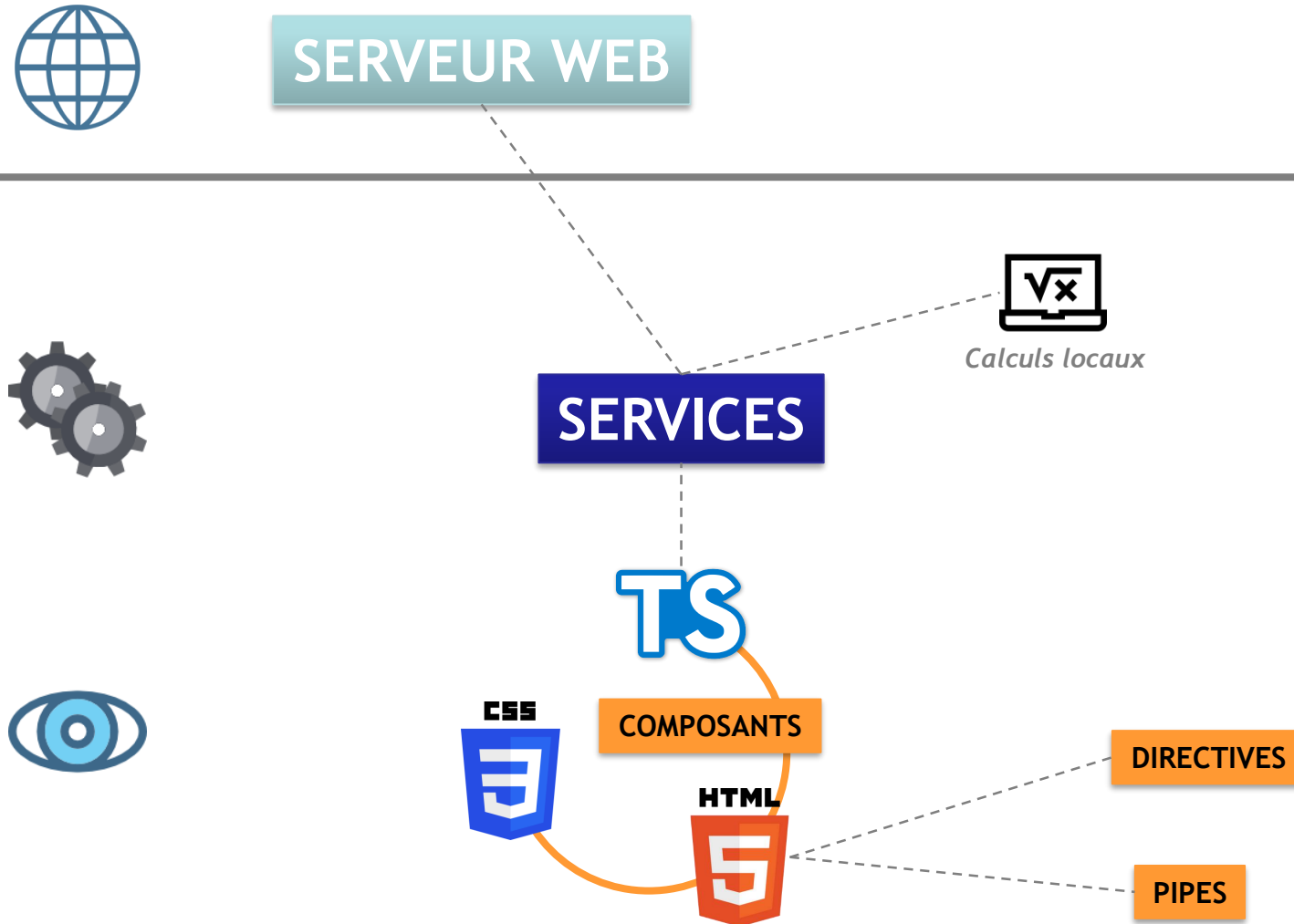
## My First Attribute Directive

Pick a highlight color

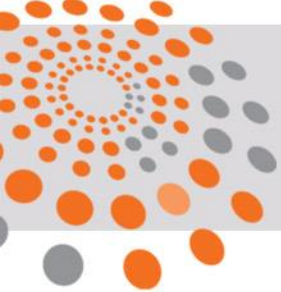
☐ Green ☐ Yellow ☐ Cyan

Highlight me!      no default-color binding

Highlight me too!      with 'violet' default-color binding







- Les composants ont parfois besoin d'interagir entre eux sans forcément passer par un service
- On utilise alors les **EventEmitter**, pour « émettre » une valeur que les autres composants peuvent saisir
- Rappel :

- Une donnée entrant dans le HTML est entre **[crochets]**
- Une donnée sortante est entre **(parenthèses)**

- L'objectif est donc :

```
<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>
```

```
<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>
```

- On utilise le décorateur **@Output** dans le composant qui a un message à émettre

```
export class Zippy {  
  visible: boolean = true;  
  @Output() open: EventEmitter<any> = new EventEmitter();  
  @Output() close: EventEmitter<any> = new EventEmitter();  
  
  toggle() {  
    this.visible = !this.visible;  
    if (this.visible) {  
      this.open.emit(null);  
    } else {  
      this.close.emit(null);  
    }  
  }  
}
```

Dans le composant parent (qui utilise **zippy**), on définit **onOpen** et **onClose**, qui doivent être déclenchés quand **zippy** émet un message (par **.emit(...)**)

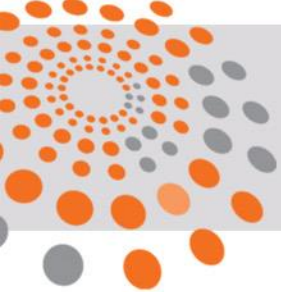
**\$event** contient la valeur émise

C'est le même principe que **(click)='...'**



# HTTP & RXJS



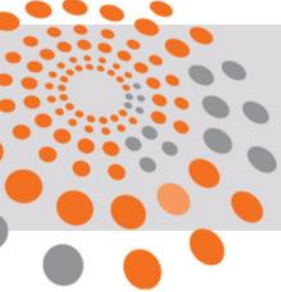


- Angular facilite de façon incroyable les **requêtes serveurs**
- Il met à disposition un service **HttpClient** (anciennement **Http**) qui dispose des méthodes HTTP usuelles :
  - `get()`
  - `post()`
  - `put()`
  - `etc.`
- En paramètre de la fonction utilisée, on précise ce que l'on veut récupérer :

```
this.http.get<number>('http://site/data/count');
```

*Angular parse tout seul  
les données récupérées !*

```
this.http.get<Person>('http://site/people/265');
```



- **HttpClient** est un service, qu'on récupère donc dans le constructeur du service qui souhaite s'en servir

```
import { HttpClient } from '@angular/common/http';
```

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



- Les appels à `get`, `post`, etc. retournent des `Observable`, qui sont une part importante de la bibliothèque `RxJS`
- Tant qu'on n'appelle pas la méthode `subscribe()` d'un `Observable`, l'appel n'est pas effectué
- Exemple :

```
const person = this.http.get<Person>('http://site/people/265');  
// person contient un Observable  
// Cette initialisation ne fait aucun appel au serveur  
  
person.subscribe();  
// Cette fois, l'appel est effectué
```



```
const person = this.http.get<Person>('http://site/people/265');  
// person contient un Observable  
// Cette initialisation ne fait aucun appel au serveur  
  
person.subscribe();  
// Cette fois, l'appel est effectué
```

- On peut (et on souhaite, a priori...) manipuler la donnée récupérée dans le **subscribe**

```
const personObservable = this.http.get<Person>('http://site/people/265');  
personObservable.subscribe(personFromServer => {  
  ...  
  this.person = personFromServer;  
});
```

- **personFromServer** est directement de type **Person**, car **personObservable** est un **Observable<Person>**

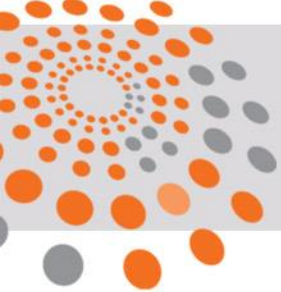


```
const personObservable = this.http.get<Person>('http://site/people/265');
personObservable.subscribe(personFromServer => {
  this.person = personFromServer;
});
```

- Ou plus simplement :

```
this.http.get<Person>('http://site/people/265')
  .subscribe(person => {
    this.person = person;
  });
```





- Une fois qu'un **Observable** a été **subscribe**, il n'est plus utilisable
  - On ne peut pas **subscribe** deux fois un même **Observable**
- Il est possible d'appliquer des pipes aux Observables de la même façon que les pipes du template :

```
this.http.get<Person>('http://site/people/265')  
  .pipe(  
    map(person => {  
      // Do something with person  
    })  
  )  
  .subscribe();
```

- Ainsi, on peut spécifier le comportement de retour avant même d'exécuter la commande (pour, par exemple, passer l'Observable prêt-à-exécuter à un service/composant)



- Pour bien comprendre, imaginons que vous allez faire les courses, et qu'on vous explique ce qu'il faudra faire en revenant



```
this.http.get<Person>('http://site/people/265')
  .subscribe(person => {
    this.person = person;
  });
```

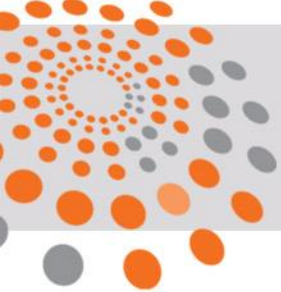
*Traitement dans le subscribe*

Après votre retour du supermarché, on vous explique que tel commission va dans le placard, telle commission va au frigo, et une fois fini, il faut nourrir le chat

```
this.http.get<Person>('http://site/people/265')
  .pipe(
    map(person => {
      // Do something with person
    })
  )
  .subscribe();
```

*Traitement dans un pipe*

Quand vous reviendrez, quelle que soit l'heure de votre départ, il faudra séparer ce qui va au frais et ce qui n'y va pas, et quand ce sera terminé, il faudra nourrir le chat



- Classiquement, on suit l'architecture suivante :

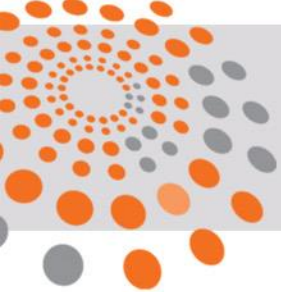
*person.service.ts*

```
function getPersonById(id: number): Observable<Person> {  
  return this.http.get<Person>('http://site/people/265')  
    .pipe(  
      map(person => {  
        // Do some stuff with the person  
        return person;  
      })  
    );  
}
```

*my-custom-comp.component.ts*

```
function myFunctionInView(idPerson: number) {  
  this.personService.getPersonById(idPerson)  
    .subscribe(person => this.viewedPerson = person);  
}
```

- On prépare toute la mécanique dans le service (si nécessaire), et on appelle **subscribe** dans le composant
- Au moment du **subscribe**, le pipe est exécuté, et son résultat est inséré dans **this.viewedPerson**



- Il y a une autre façon d'appeler **subscribe** dans un composant : le pipe **Async**

*person.service.ts*

```
function getPersonById(id: number): Observable<Person> {  
  this.http.get<Person>('http://site/people/265')  
}
```

*my-custom-comp.component.ts*

```
public viewedPerson$: Observable<Person> = this.personService.getPersonById(12);
```

Le pipe **Async** appelle la méthode **subscribe** sur l'Observable

Ici, le **\*ngIf** remplit **viewedPerson** (qui n'existe pas dans le composant) avec le retour de **viewedPerson\$**

*my-custom-comp.component.html*

```
<div *ngIf="(viewedPerson$ | async) as viewedPerson">  
  <span>{{ viewedPerson.name }}</span>  
  <span>{{ viewedPerson.firstname }}</span>  
</div>
```



# TOUR OF HEROES

## (2/2)





- Vous allez maintenant reprendre **Tour of Heroes**, pour faire la partie « 6. HTTP » (et les précédentes si ce n'est pas fait)
- Dans cette partie, vous allez simuler un serveur avec une base de données de super-héros
- Vous allez requêter cette liste de héros, et implémenter les fonctions basiques de CRUD

# Tour of Heroes

The end



You're at the end of your journey, and you've accomplished a lot.

- You added the necessary dependencies to use HTTP in the app.
- You refactored HeroService to load heroes from a web API.
- You extended HeroService to support `post()`, `put()`, and `delete()` methods.
- You updated the components to allow adding, editing, and deleting of heroes.
- You configured an in-memory web API.
- You learned how to use observables.



- Vous avez réussi **Tour of Heroes**, ok, vous êtes très forts
- Mais **Angular** et **RxJS** ont encore des secrets pour vous et nous allons en dévoiler quelques-uns :
  - Les **Subjects** : des **Observables** un peu particuliers
  - Les **BehaviorSubjects** : des **Subjects** un peu particuliers
  - Les opérateurs de combinaison : pour combiner les **Observables**
  - La compilation **Ahead-of-Time** : pour aller plus vite





# DERNIER ROUND



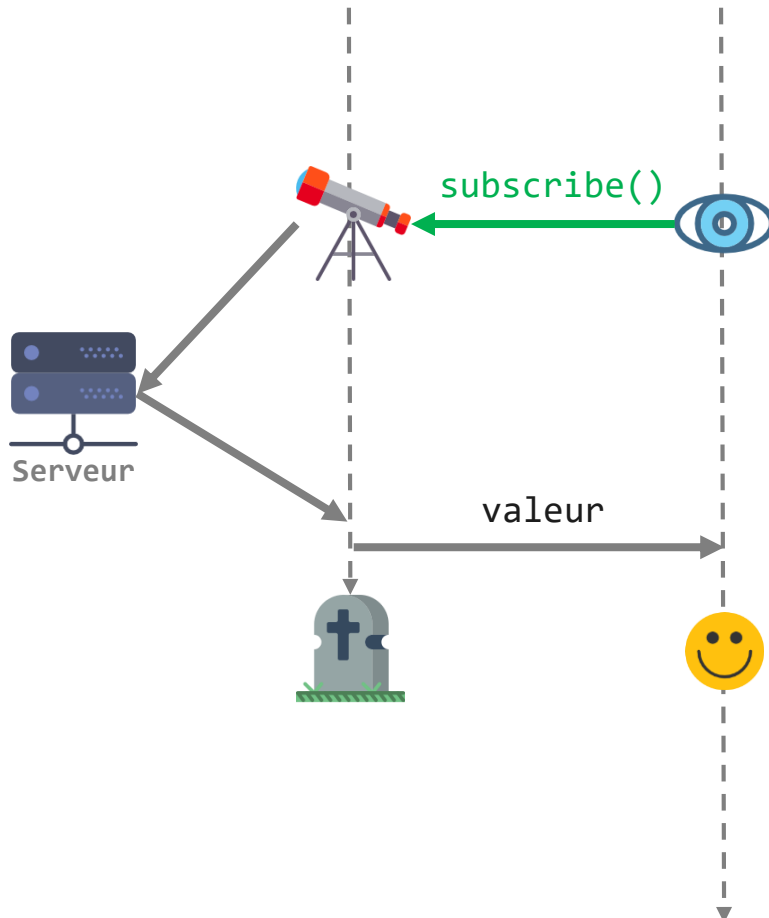


- Vous avez manipulé des **Observables**, qu'on a écoutés une fois pour toutes avec **subscribe**
- Il existe des **Observables** qu'on écoute en continu, et qui peuvent envoyer différentes valeurs à différents moments de la vie de l'application
- Ce sont les instances de la classe **Subject**

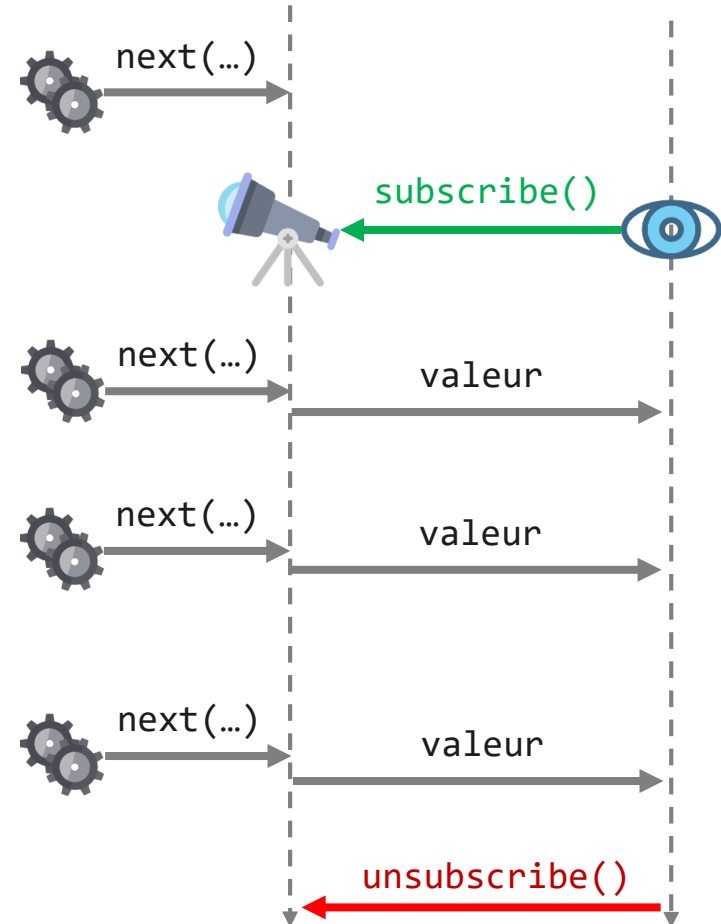
# Dernier round

## Subjects

### Observable



### Subject



- Les **Subjects** permettent de déclencher une action à chaque fois qu'ils reçoivent une nouvelle valeur
- Exemple :

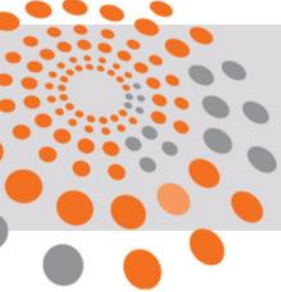


Lecteur audio basique

On subscribe un **Subject isPlaying\$** pour afficher le bouton **pause** ou **play**

On subscribe un **Subject currentTime\$** pour mettre à jour la barre de progression

- Problème : on « attrape » une nouvelle valeur, mais on ne peut pas récupérer la dernière valeur « popée »



- Solution : `BehaviorSubject`
- Le `BehaviorSubject` est un `Subject` dont le constructeur prend une valeur initiale et qui a un attribut `value` qui contient la dernière valeur
- On peut ainsi `subscribe` un `Subject` et récupérer la dernière valeur qu'il a émise, pour « *attraper le train en marche* »

- Exemple :

```
export class RandomView {  
  constructor(  
    ... private counterService: CounterService  
  ) { }  
  
  ngOnInit() {  
    ... this.counterService.counter  
    ... .subscribe(value => document.write(value));  
  }  
}
```

Ce composant écrira 1, 2, 3...

```
@Injectable()  
export class CounterService implements OnInit {  
  public counter = new BehaviorSubject<number>(0);  
  
  constructor() { }  
  
  public increment() {  
    ... this.counter.next(this.counter.value + 1);  
  }  
}
```

```
export class RandomView2 {  
  constructor(  
    ... private counterService: CounterService  
  ) { }  
  
  ngOnInit() {  
    document.write(this.counterService.counter.value);  
  
    ... this.counterService.counter  
    ... .subscribe(value => document.write(value));  
  }  
}
```

Ce composant écrira 0, 1, 2, 3...



- Il existe de nombreux opérateurs de combinaison des Observables

<https://www.learnrxjs.io/operators/combinations/>

### Combination

combineAll

combineLatest

concat

concatAll

endWith

forkJoin

merge

mergeAll

pairwise

race

startWith

withLatestFrom

zip

Retourne la dernière valeur de chaque Observable dès que l'un d'eux émet une valeur

Retourne la dernière valeur de chaque Observable dès que le dernier Observable émet une valeur

```
/*
  when all observables complete, give the last
  emitted value from each as an array
*/
const example = forkJoin(
  //emit 'Hello' immediately
  of('Hello'),
  //emit 'World' after 1 second
  of('World').pipe(delay(1000)),
  //emit 0 after 1 second
  interval(1000).pipe(take(1)),
  //emit 0...1 in 1 second interval
  interval(1000).pipe(take(2)),
  //promise that resolves to 'Promise Resolved' after 5 seconds
  myPromise('RESULT')
);
//output: ["Hello", "World", 0, 1, "Promise Resolved: RESULT"]
const subscribe = example.subscribe(val => console.log(val));
```



# Dernier round

JiT vs AoT

- On l'a vu, pour déployer une application Angular, il faut la compiler (**ng build**)
- Lors de la compilation, Angular prend l'ensemble de vos templates et les intègre dans du JavaScript (**main.bundle.js** et **vendor.bundle.js**) (ou **-es2015**)

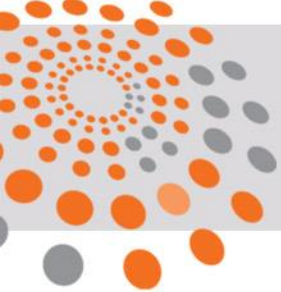
```
/***/ })),  
  
/***/ ".../.../.../.../src/app/components/home/home.component.html":  
/***/ (function(module, exports) {  
  
  module.exports = "{user|json}}\n\n<button (click)='logout()'>logout</button>";  
  
/***/ })),
```

Source : <https://sebastienollivier.fr/blog/angular/angular-jit-vs-aot>

- Au chargement d'un composant, Angular va lire cette chaîne, l'interpréter et exécuter les bindings nécessaires
- On appelle cela la compilation Just in Time (JiT)







- La compilation génère un fichier **main** et un fichier **vendor**, qui contiennent le JavaScript de votre application
- Avec une application extrêmement basique, on obtient déjà :

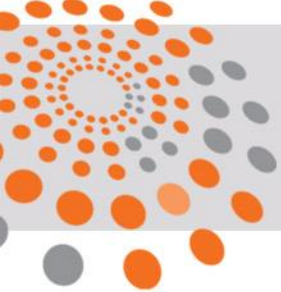


**/dist**  
**17,5 Mo**

- C'est déjà assez conséquent... et donc pénible à charger
- En plus, les performances sont impactées par le fait que chaque composant est compilé « à la volée » (JiT)



- Pour améliorer les performances, Angular met à disposition la compilation **Ahead-of-Time (AoT)**
- Cette compilation va transformer chaque template HTML en code JavaScript (l'opération effectuée à la volée en JiT)
- On gagne donc en performance !
- On utilise `ng build --aot`



- Par ailleurs, par défaut, le projet est compilé en mode « développement », ce qui nous permet de le débbugger simplement (TypeScript dans Chrome, sources, etc.)
- Lorsqu'on veut mettre en production, on souhaite retirer toutes ces informations pour ne garder que l'indispensable (d'ailleurs, Chrome le dit, en dev !)
- On gagne **encore plus** en performance (et prod inclut l'AoT) !
- On utilise `ng build --prod`



/dist  
664 Ko

# Dernier round

## JiT vs AoT

JiT

`ng build`



`/dist`



Idéal pour le **développement**  
car la compilation est rapide

AoT

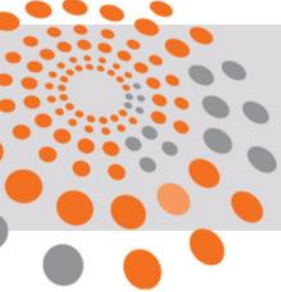
`ng build --aot`    `ng build --prod`



`/dist`



Idéal pour la **production**  
car c'est performant !



- L'IndexedDB est une base de données noSQL propre au navigateur
- Elle sert à stocker des données au même titre que les cookies, ou le local storage
- Elle fonctionne sur le principe de clés/valeurs, et stocke des objets JavaScript
- Il existe un package pour Angular :

<https://www.npmjs.com/package/ngx-indexed-db>

- Et un tuto plutôt bien construit : 

<https://lesdieuxducode.com/blog/2018/3/stockage-cot-client-avec-indexeddb>



- Les **service workers** gèrent la mise en cache des applications web, de telle façon qu'elles soient accessibles **hors-ligne** (si, si !)
- Ils sont stockés côté navigateur et retournent des informations, **à jour** si on est connecté... **moins à jour** si on ne l'est pas
- Ces applications, accessibles hors-ligne, sont appelées **Progressive Web App (PWA)**

<https://angular.io/guide/service-worker-intro>



- **Socket.IO** est une bibliothèque qui permet de créer des sockets au sein d'une application web
- Ces sockets offrent la possibilité de **communiquer** avec une autre application via un serveur, en évitant de faire des appels en boucle
- Concrètement, un(e) socket se branche sur une URL (un *endpoint*) et écoute ce qui se passe dessus (comme un appel qui ne se finirait jamais)
- Une application évidente : un **chat en ligne**
- Il existe une implémentation pour Angular :  
<https://www.npmjs.com/package/ngx-socket-io>



# CONCLUSION





- **Angular** est un framework innovant, en constante évolution
- Il est porté par **Google** et soutenu par une énorme communauté
- L'une des difficultés est la **documentation**, qui doit sans cesse suivre les mises à jour
  - **Attention à StackOverflow** : toujours vérifier la version d'Angular ou la date de publication...
- Angular sert également à faire des **applications mobiles**, avec des surcouches telles qu'**ionic**



- **Angular** est un framework innovant, en constante évolution
- Il est porté par **Google** et soutenu par une énorme communauté
- L'une des difficultés est la **documentation**, qui doit sans cesse suivre les mises à jour
  - **Attention à StackOverflow** : toujours vérifier la version d'Angular ou la date de publication...
- Angular sert également à faire des **applications mobiles**, avec des surcouches telles qu'**ionic**





**MERCI DE VOTRE ATTENTION**

