

# Angular 2

Adrien Vossough



# SOMMAIRE

Les outils

HTML5 et DOM

Web component

Premier projet

IoC/Injection de dépendances

Premier projet

Angular CLI

Templates + composants

Les pipes

Les services

Programmation réactive

Les Directives

Composant : Style et template

Formulaires et Binding Two Way

Bibliographie



# Les outils



# Les outils



## Compiler :

- La compilation est le fait de traduire du code dans un langage A vers un langage B plus bas niveau
  - Exemple : Transformer du code en C++ vers du code machine

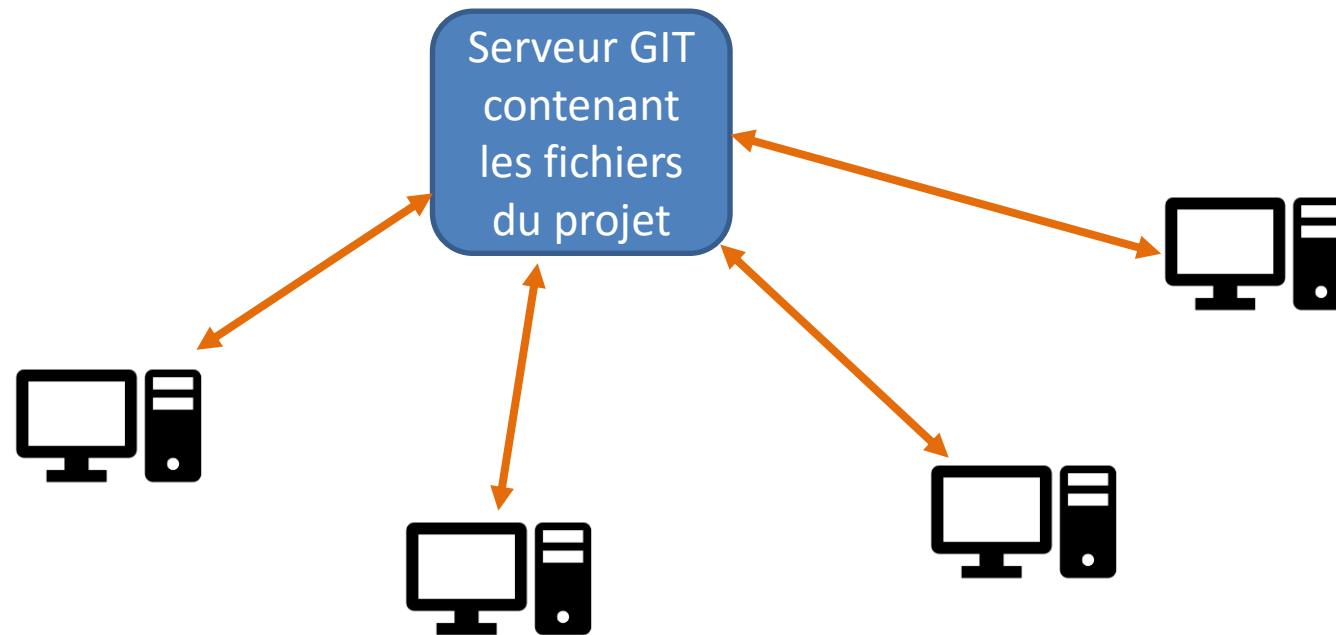
## Transpiler :

- Transpiler transforme du code dans un langage vers un autre de même niveau.
  - Exemple transformer la version de JavaScript ES6 vers la version ES5

# Les outils : Git

## Git :

- Gestionnaire de versions
- Permet de garder un historique des modifications
- Permet de mutualiser le développement d'un projet entre plusieurs développeurs



Chaque développeur peut modifier les fichiers du projet et ensuite le partager avec son équipe en l'envoyant sur un serveur Git.

Chaque développeur doit avoir un "client Git" pour pouvoir se connecter au serveur.

# Les outils : Node.js

## Node.js :

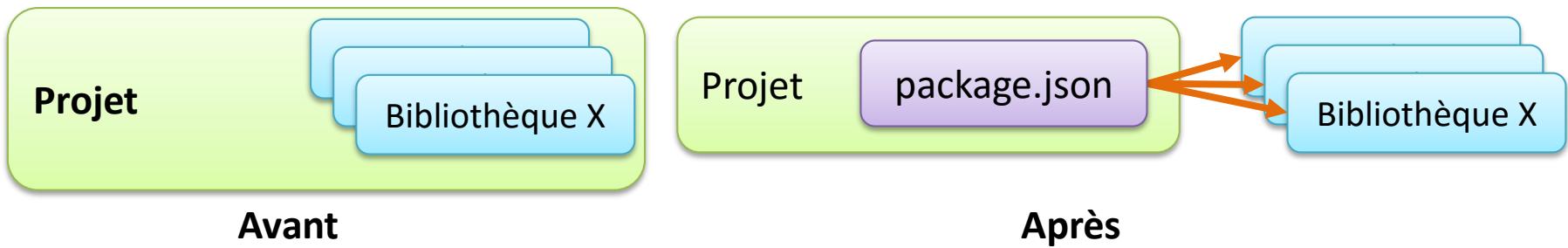
- Est une plateforme applicative pour exécuter des applications en JavaScript
- Très utilisé dans le web pour faire des serveurs ou des outils pour aider les développeurs.
- <https://nodejs.org/en/>

# Les outils : npm

npm :

- Gestionnaire de paquet de Node.js
- Il permet de télécharger facilement des paquets (bibliothèques ou programmes)

La commande **npm init** dans le répertoire du projet va créer un fichier contenant les dépendances du projet.



Dans le répertoire du projet, télécharge et ajoute cette dépendance au fichier du projet

```
npm install --save @angular/core
```

Ajouter une bibliothèque accessible à tous les projets :

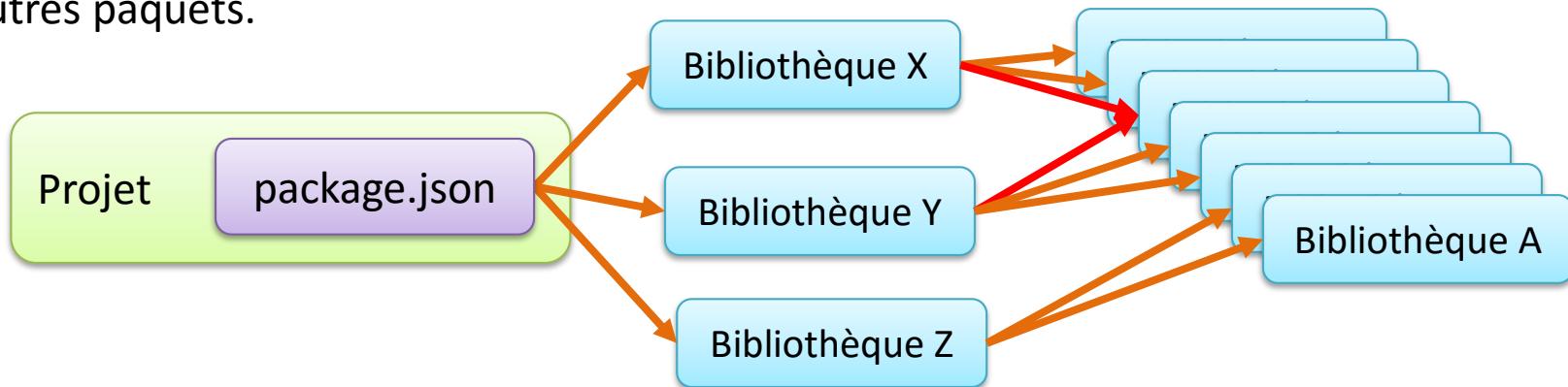
```
npm install -g angular-cli
```

# Les outils : Bower

bower :

- Gestionnaire de paquet pour Node.js
- Il permet de télécharger facilement des paquets (bibliothèques ou programmes)

Les gestionnaires de paquets gère les dépendances qui elles-mêmes peuvent dépendre d'autres paquets.



**npm** **contrairement** à **bower** ne reconnaît pas les interdépendances, ce qui veut dire que si la bibliothèques X et Y ont les mêmes dépendances, nous les aurons en doublon.



Côté développement front-end, **npm** est utilisé pour les outils tandis que **bower** pour les dépendances du projet car plus rapide pour installer les dépendances

# Les outils : Gulp

gulp :

- Automatiseur de tâches
- Permet d'exécuter une série de commandes (tâches) selon des cas définis

**Exemple de tâches pouvant être géré automatiquement par gulp :**

- On démarre le projet
- Il faut vérifier que toutes les bibliothèques sont présentes
- Il faut transformer les fichiers TypeScript en fichier JavaScript
- Il faut exécuter le serveur qui lancera le projet

**Installer Gulp : <http://gulpjs.com/>**

# Les outils : Yeoman



Yeoman :

- Générateur de projets
- Permet de créer la structure et mettre en place les outils pour l'ensemble d'un projet web (back et front) et de le configurer pour qu'il soit exécutable immédiatement

Installer Yeoman :

```
npm install -g yo
```

L'outil est installé, il a besoin d'une fichier de configuration appelé générateur de projet :  
<http://yeoman.io/generators/>

L'installation d'un générateur est généralement indiquée sur une page web

Exemple : <http://fountainjs.io/>

# Les outils :



## TSLint, ESLint, JSLint, JSHint :

- Analyseurs de code (TypeScript, ES6, ES5)
- Donnent des indications sur le code source pour le rendre plus propre et maintenable

## Babel :

- Transpileur ES6 vers ES5

## Browsersync :

- Relance automatiquement le navigateur à chaque modification de l'application pour afficher les changements.

## Protractor :

- Permet de tester l'application Angular dans un navigateur en simulant un utilisateur

## Jasmine :

- TDD : Permet de mettre en place des tests unitaires

# HTML5 et DOM





Le **DOM** (Document Object Model) est une représentation d'un document sous forme structuré, généralement sous forme d'arbre.

- Pour le navigateur, chaque balise est un objet avec des attributs et méthodes
- Les balises et valeurs sont appelés nœuds dans le DOM.
- Un nœud est appelé parent s'il a des nœuds descendants, ces derniers sont appelés "nœuds enfants".
- Les nœuds du même parent sont frères.

*(vocabulaire utilisé par les navigateurs et non de la théorie des graphes)*

# HTML5 : Le DOM

Exemple de page HTML :

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>Titre de ma page</title>
  </head>

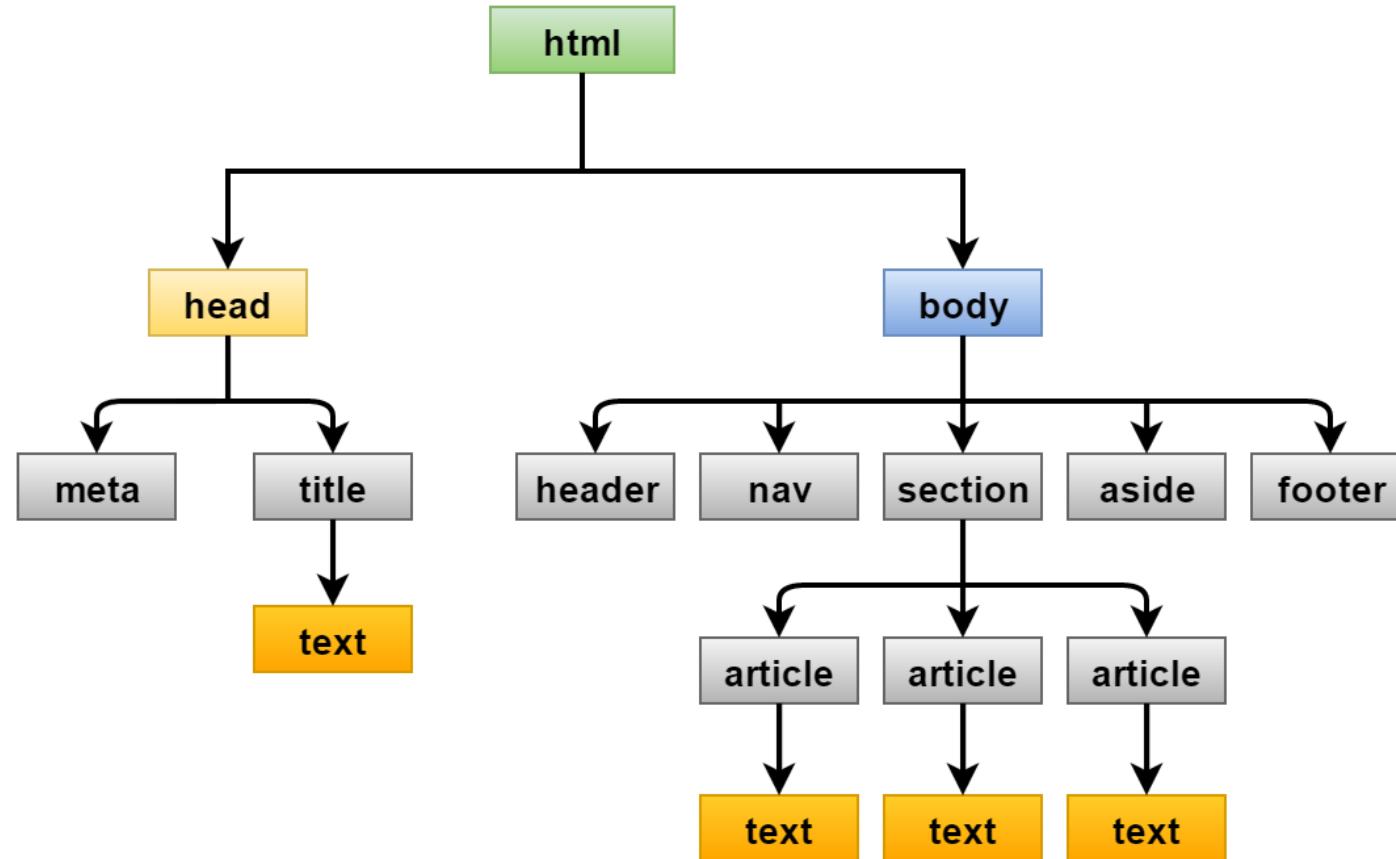
  <body>
    <header></header>
    <nav></nav>

    <section>
      <article>Texte de l'article 1</article>
      <article>Texte de l'article 2</article>
      <article>Texte de l'article 3</article>
    </section>

    <aside></aside>
    <footer></footer>
  </body>
</html>
```

# HTML5 : Le DOM

Représentation du DOM de la page HTML :



*DOM simplifié (seul les éléments HTML sont représentés)*

# HTML5 : Liste de balises

<... /> balises "auto-fermantes"    <address>: balise de type "block"

Obsolète	HTML5 mal supporté	HTML5 bien supporté	A éviter
<!-- ... -->	<canvas>	<figcaption>	<progress>
<!DOCTYPE>	<caption>	<figure>	<td>
<a>	<center>	<font>	<textarea>
<abbr>	<cite>	<footer>	<tfoot>
<acronym>	<code>	<form>	<th>
<address>	<col />	<frame>	<thead>
<applet>	<colgroup>	<frameset>	<time>
<area />	<command />	<h1> à <h6>	<title>
<article>	<datalist>	<head>	<tr>
<aside>	<dd>	<header>	<track />
<audio>	<del>	<hgroup>	<tt>
<b>	<details>	<hr />	<u>
<base />	<dfn>	<html>	<ul>
<basefont>	<dialog>	<i>	<var>
<bdi>	<dir>	<iframe>	<video> 5
<bdo>	<div>	<img />	<wbr />
<big>	<dl>	<input />	
<blockquote>	<dt>	<ins>	
<body>	<em>	<kbd>	
 	<embed />	<keygen />	
<button>	<fieldset>	<label>	
		<legend>	
		<link />	
		<map>	
		<mark>	
		<menu>	
		<menuitem>	
		<meta />	
		<nav>	
		<noframes>	
		<noscript>	
		<object>	
		<ol>	
		<optgroup>	
		<option>	
		<output>	
		<p>	
		<param />	
		<pre>	
		<table>	
		<tbody>	

# Web Component



# Web Component



## Web Component :

- Brique web facilement réutilisable
- Généralement, un composant indépendant

## Les Web Components se bases sur 4 éléments :

- Custom elements ("éléments personnalisés")
- Shadow DOM ("DOM de l'ombre")
- Template
- HTML imports

Ces 4 éléments ne fonctionnent pas sur tous les navigateurs, il faut utiliser un **polyfill**.

## Polyfill :

- Bibliothèque (**js** généralement) pour mettre à niveau la technologie d'un navigateur.
- Polymer et X-tag permettent l'intégration des Web Components



## Custom elements :

- Balises HTML5 personnalisées
- Doit contenir un tiret : <**ma-balise**></**ma-balise**>

### Création d'un Custom element

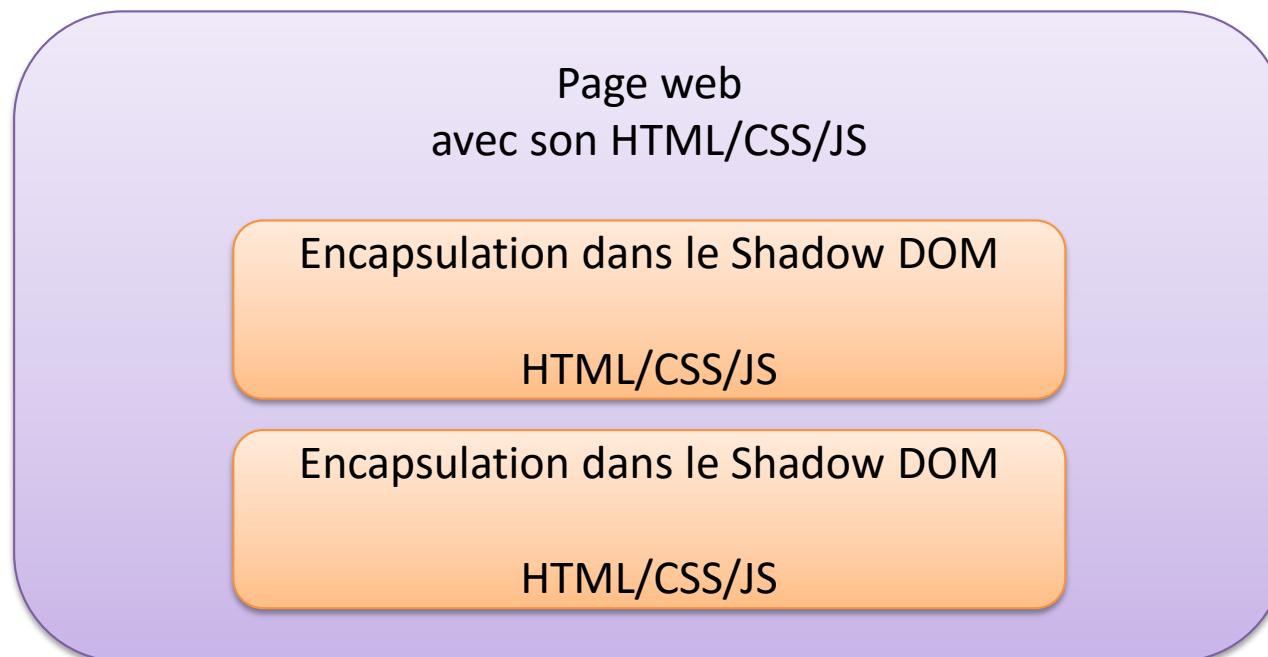
```
let BalisePersonnalisee = document.registerElement('ma-balise');

document.body.appendChild(new BalisePersonnalisee());
```

# Web Component : Shadow DOM

## Shadow DOM :

- DOM encapsulé
- Contient son HTML, CSS et JS qui lui est propre
- Le CSS et le JS ne s'applique qu'à son HTML
- Le CSS/JS de la page ne s'applique pas sur lui ni celui des autres encapsulations





## Template :

- Balise HTML <template></template>
- La balise et son contenu ne sont pas affichés
- Les balises <script></script> et <img> à l'intérieur ne sont pas exécutées
- Sert de patron comme une classe
- Il doit être clone (sorte d'instanciation)

## HTML imports :

- Possibilité d'importer du HTML dans du HTML comme un include en PHP, import en JAVA, etc.
- <link rel="import" href="mon-fichier.html">.

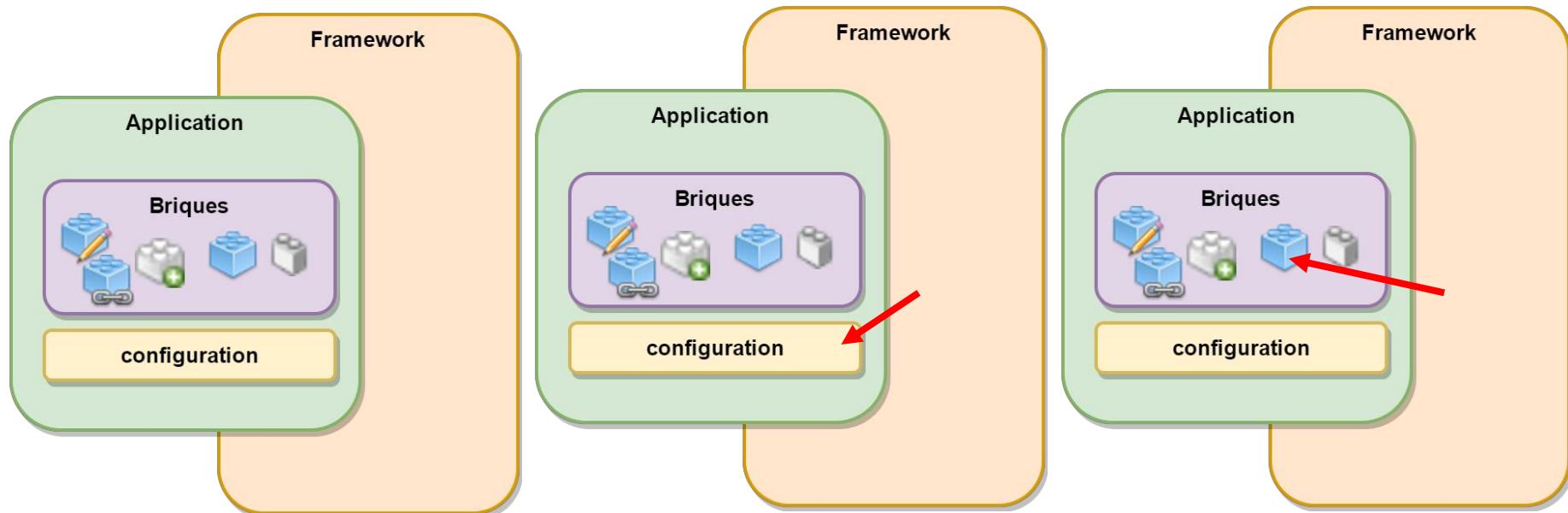
# Inversion de contrôle et injection de dépendances



# Inversion de contrôle

## Inversion de contrôle :

- Ce n'est pas l'application qui contrôle l'ordre d'exécution de ses briques mais une entité extérieure.
- Cela peut être un framework, un programme ou un utilisateur.



Mise en place de l'application avec le framework

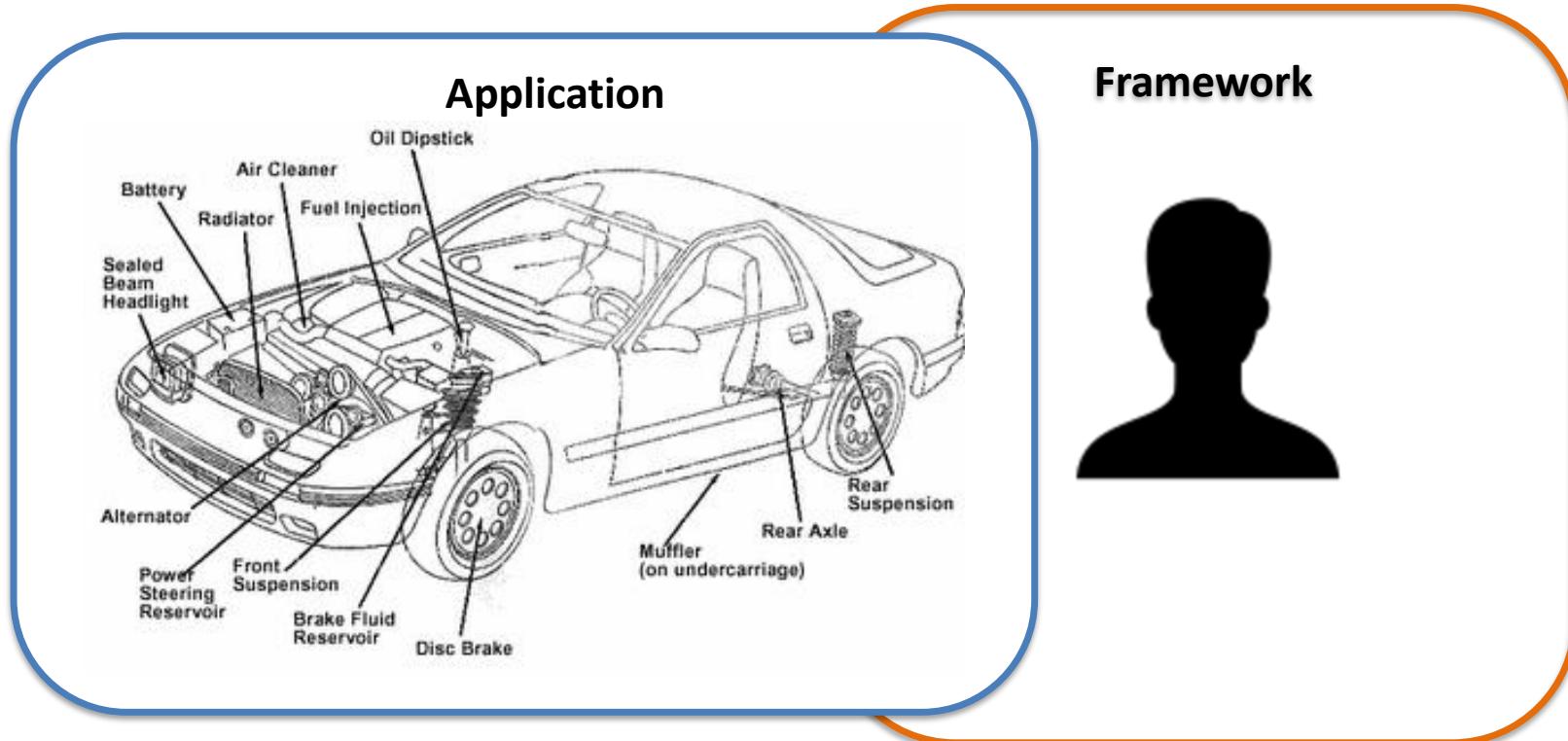
Exécution de l'application.  
Le framework lit la configuration

Le framework utilise les briques applicatives lorsqu'il en a besoin.

# Inversion de contrôle

## Inversion de contrôle :

- Nous pouvons comparer cela au conducteur avec sa voiture



1 - La configuration et ce qui se trouve dans l'habitacle, position du volant, levier de vitesses, freins, etc.

2 - Lorsque le conducteur à besoin d'accélérer il activera le levier de vitesse, et s'il doit freiner appuiera sur la pédale de frein.

**Le conducteur contrôle l'ordre d'exécution des commandes.**



## Injection de dépendances :

- Type d'inversion de contrôle.
- Lorsqu'un élément N a besoin d'un élément B pour fonctionner, on le lui fournit.

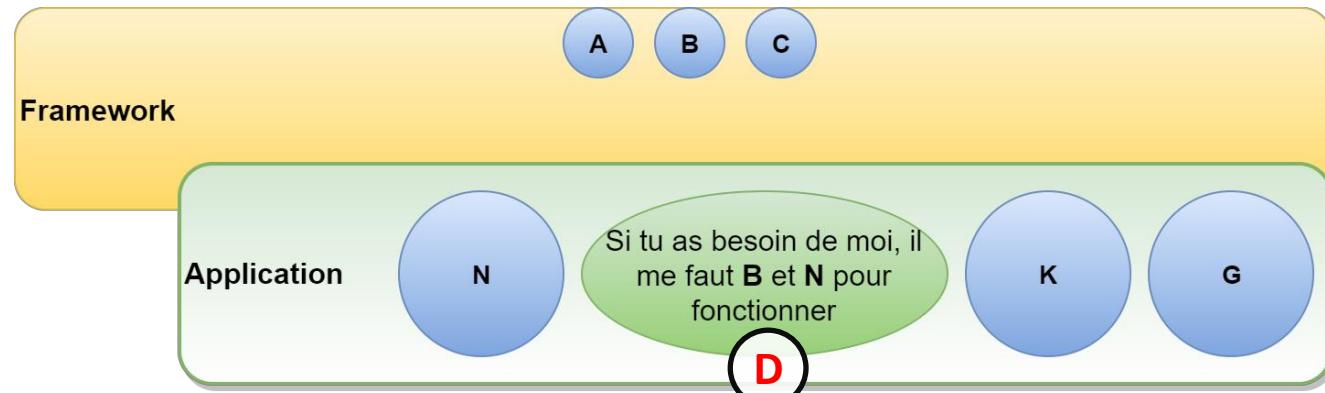
## Dans le cas de notre voiture :

- Notre moteur a besoin d'essence pour fonctionner. On lui injecte donc de l'essence.
- Nos freins ont besoin de liquide, on le lui ajoute.

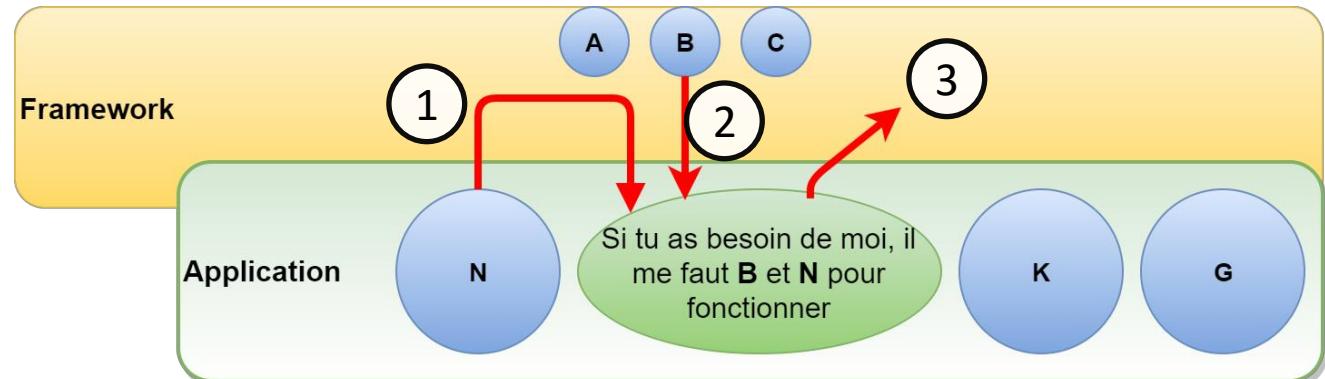
Ce n'est donc pas la voiture qui gère les dépendances mais le conducteur.

# Inversion de contrôle : Injection de dépendances

## Injection de dépendances dans un framework :



L'élément D a besoin d'autres éléments pour être utilisé.



Au moment où le framework aura besoin de D, il va récupérer les éléments N et B, les fournir à D puis le récupérer pour l'utiliser.

Les éléments peuvent provenir de l'application elle-même ou du framework.

# Premier projet



# Installation

## Installer Node.JS :

- <https://nodejs.org/en/>
- Prendre la version LTS (version stable)
- Une fois l'installation faite, dans la console windows taper : `node -v`
- Si rien ne se passe : vérifier les variables d'environnement (C:\Program Files\nodejs\)

## Vérifier npm:

- Ouvrir une console et taper `npm -v`

**Créer le répertoire du projet et se placer avec la console dans ce répertoire.**

## Ajouter TypeScript :

- Installation : `npm install -g typescript`
- Initialiser le fichier de configuration du compilateur TypeScript :

```
tsc --init --target es5 --sourceMap --experimentalDecorators --emitDecoratorMetadata
```



Les commandes `npm install -g` n'ont plus besoin d'être retapés pour les futurs projets.  
`-g` indique une installation globale, donc pour tous les projets.

# Installation

tsc a créé un fichier de configuration

- <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>
- <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

## Fichier tsconfig.json :

```
{  
  "compilerOptions": {  
    "target": "es5", // version cible de JavaScript (ES3 par défaut)  
    "experimentalDecorators": true, // accepte les décorateurs non standard  
    "emitDecoratorMetadata": true, // obligatoire pour l'option précédente  
    "sourceMap": true, // crée un fichier Map pour chaque fichier transpilé  
    "module": "commonjs", // permet l'utilisation de modules (require, export)  
    "noImplicitAny": false // les variables non typées ne seront plus des "any"  
  },  
  "exclude": [  
    "node_modules" // ne doit pas compiler ce qui se trouve dans ce répertoire  
  ]  
}
```

# Installation

Initialiser le répertoire avec le fichier de configuration npm : `npm init`

- Un fichier **package.json** contenant les dépendances de notre projet est créé.
- <https://docs.npmjs.com/>

Installer les packages pour notre projet :

**npm install --save**

<b>@angular/core</b>	// framework Angular
<b>@angular/compiler</b>	// Fait le lien entre le template et les fonctions
<b>@angular/common</b>	// package de directives, pipes et services
<b>@angular/platform-browser</b>	// package pour l'exécution sur navigateur
<b>@angular/platform-browser-dynamic</b>	// gestion/affichage des templates
<b>rxjs</b>	// bibliothèque pour faire du code basé sur les événements
<b>reflect-metadata</b>	// permet les annotations avec options
<b>zone.js</b>	// obligatoire pour le binding (voir bibliographie)
<b>systemjs</b>	// charge les modules ES et gère dépendances

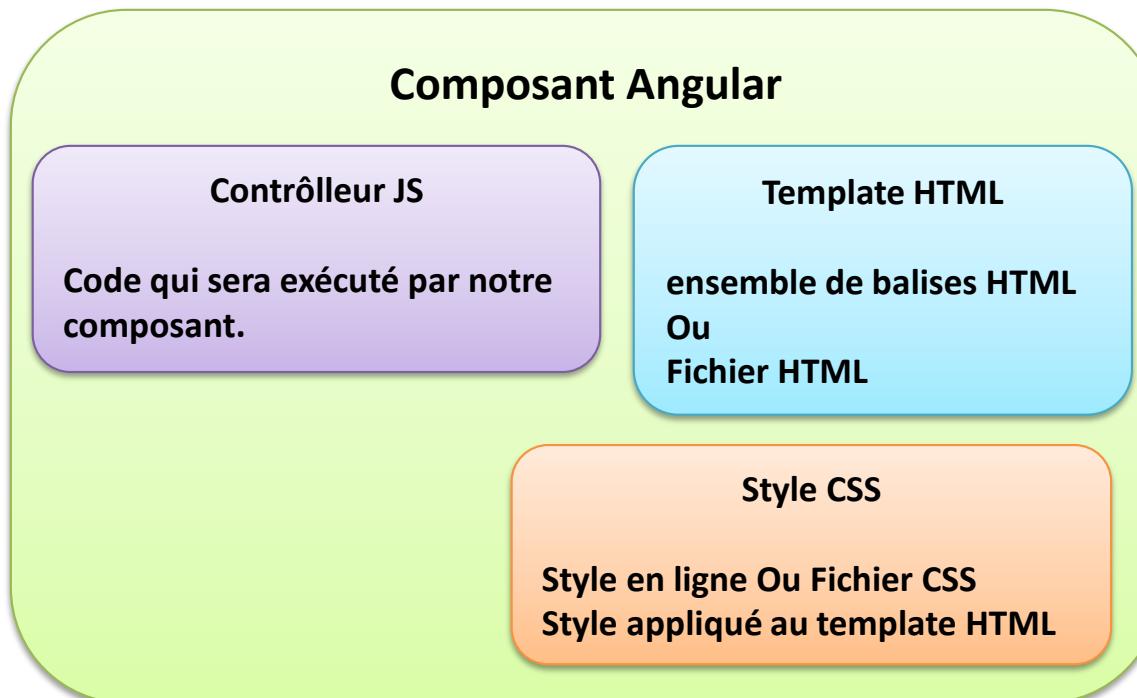
Ajouter les fichiers de déclarations de types (voir bibliographie)

`npm install --save-dev @types/core-js`

# Mini projet : premier composant

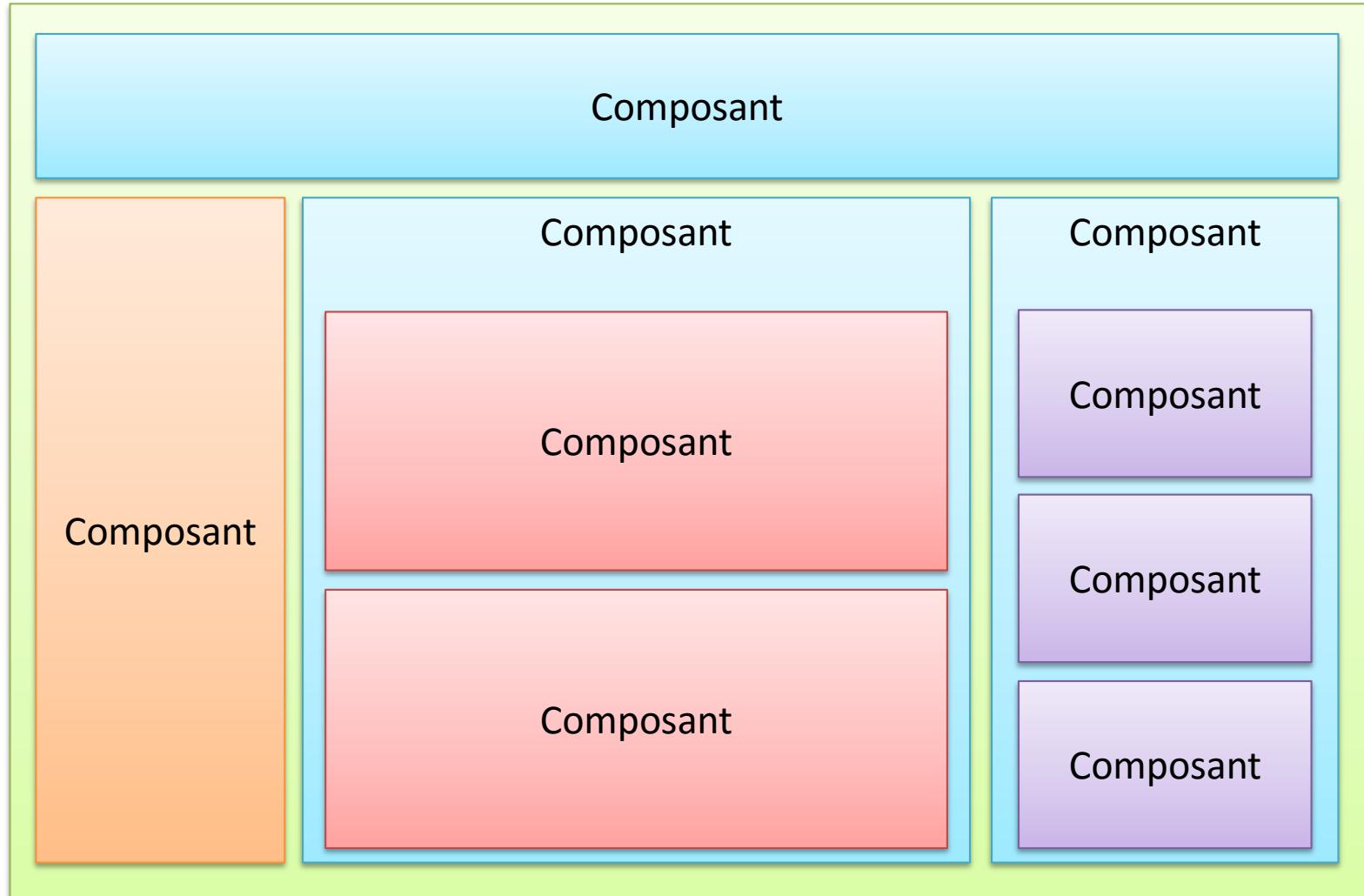
## Composant Angular :

- Brique Angular contenant du JS, HTML et CSS.
- Chacun des éléments de la brique sont "indépendants" du reste de l'application



# Mini projet : premier composant

Une page HTML peut ressembler à cela :



# Mini projet : premier composant

Premier composant :

premier.component.ts

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

@Component({
  selector: 'premier-composant',
  template: '<h1>Mon premier composant</h1>'
})

export class PremierComponent {
```

Le décorateur **@Component** indique à Angular que PremierComponent est un composant.

**selector** : Identifiant de notre composant

**template** : template HTML du composant

**La classe** : Est notre composant

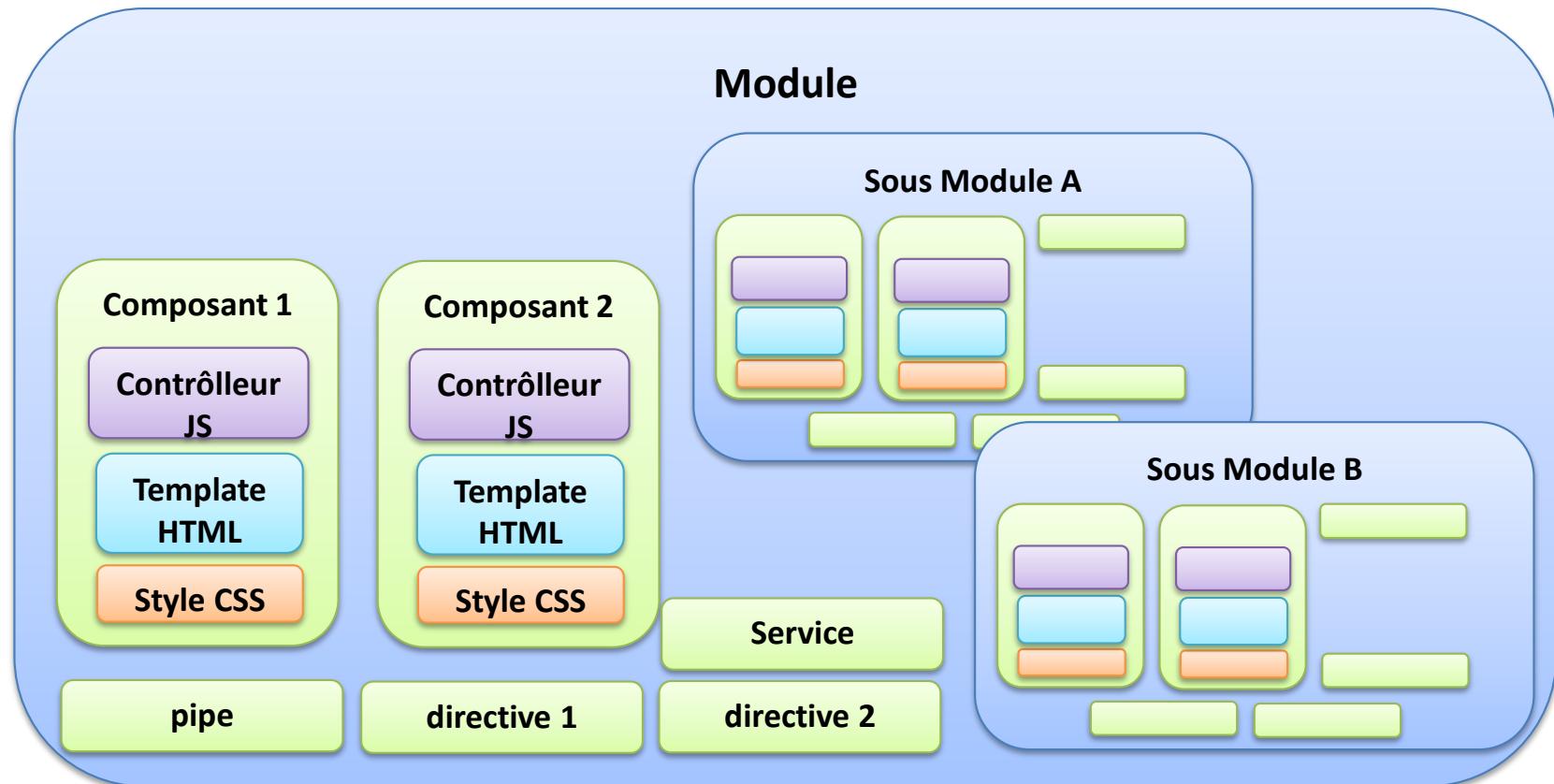


La commande **tsc** dans la console permet de transpiler le code TS en JS  
**tsc --watch** transpile lors de chaque sauvegarde d'un fichier TS.

# Mini projet : premier module

## Module Angular :

- Ne pas confondre avec les modules ES6 et TS
- Conteneur comprenant différentes briques Angular (composants, pipes, directives, services ou autres modules)
- Les modules permettent d'organiser le code d'une application sous forme de blocs de fonctionnalités communes.



# Mini projet : premier module

Création du module racine :

app.module.ts

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

@NgModule({
  imports: [BrowserModule],
  declarations: [PremierComponent],
  bootstrap: [PremierComponent]
})

export class AppModule { }
```



**BrowserModule** ne peut être inclus que dans le module principal sinon il faut utiliser **CommonModule**

- **NgModule** : Indique à Angular que **AppModule** est un module.
- **BrowserModule** : Importe les éléments habituels d'une application (directives, services...)
- **declarations** : Liste des **composants** à inclure
- **bootstrap** : Composant à charger en premier sur la page index.html

# Mini projet : Script d'exécution



## Script d'exécution de l'application:

main.ts

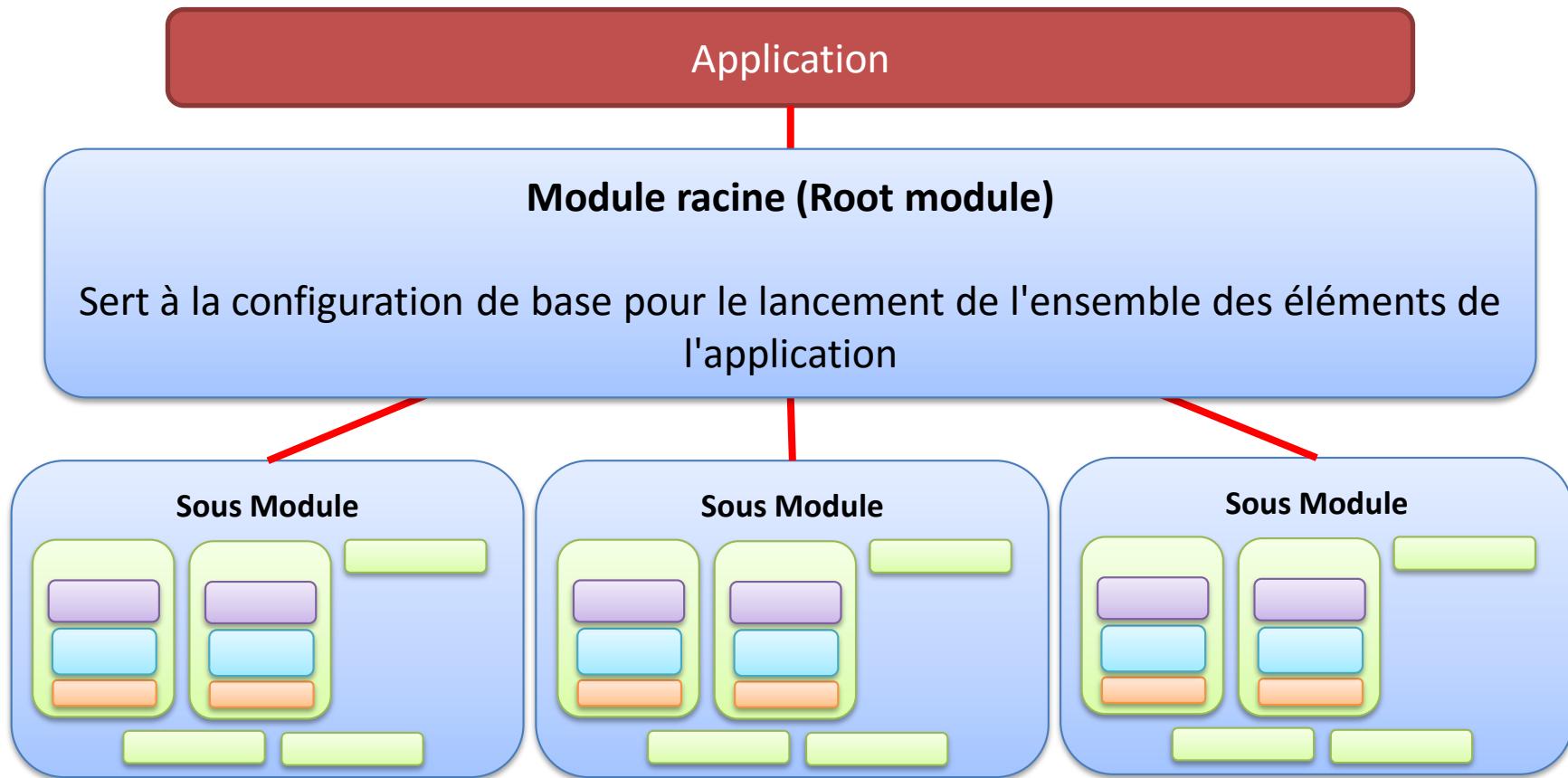
```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

**platformBrowserDynamic** : Permet d'executer une application dans un navigateur  
**bootstrapModule** : Indique le module racine de notre application

# Mini projet : premier module

Schéma simplifié d'une application qui sera lancé par Angular :



# Mini projet : Configuration SystemJS

## Configuration de SystemJS:

index.js

```
// configuration de SystemJS
System.config({
    // mapping des dépendances
    map: {
        '@angular/core': 'node_modules/@angular/core/bundles/core.umd.js',
        '@angular/common': 'node_modules/@angular/common/bundles/common.umd.js',
        '@angular/compiler': 'node_modules/@angular/compiler/bundles/compiler.umd.js',
        '@angular/platform-browser': 'node_modules/@angular/platform-
browser/bundles/platform-browser.umd.js',
        '@angular/platform-browser-dynamic': 'node_modules/@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
        'rxjs': 'node_modules/rxjs'
    },
    packages: {
        // '.' tous les packages -> les modules serons chargés sans suffixe ".js"
        // SystemJS ajoutera l'extension automatiquement
        '.': {}
    }
});
// démarrage de notre application
System.import('main');
```

# Mini projet : Création template

Fichier index.html :

Index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Titre de mon application</title>
  </head>

  <body>
    <premier-composant> Chargement d'Angular... </premier-composant>
  </body>

</html>
```

premier.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'premier-composant',
  template: '<h1>Mon premier composant</h1>'
})
export class PremierComponent {
```



- La balise `<premier-composant> </premier-composant>` active le composant correspondant au sélecteur "premier-composant"
- Le contenu de la balise **premier-composant** sera retiré après le chargement du composant.

# Mini projet : Configuration template



Ajout des scripts à lancer par le template de démarrage :

- Nous ajoutons les scripts pour le besoin de notre application

## index.html en-tête

```
<head>
  <meta charset="UTF-8">
  <title>Titre de mon application</title>

  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/systemjs/dist/system.js"></script>
  <script src="index.js"></script>
</head>
```

Installation d'un serveur pour exécuter l'application :

```
npm install -g http-server
http-server // Vérifier que tous les fichiers .ts sont transpilés en .js avant de lancer le serveur
```

Dans un navigateur aller sur : <http://localhost:8080/>

# Angular CLI



# Angular CLI

## Angular CLI (voir bibliographie) :

- Interface en ligne de commande.
- Permet de générer un squelette de projet Angular.
- Permet de générer le squelette d'un élément Angular (composant, pipe, etc.).

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

Générer un projet : **ng new monprojet**

Pour exécuter un projet : **ng serve**

Élément	commande
Component	ng g component my-new-component
Directive	ng g directive my-new-directive
Pipe	ng g pipe my-new-pipe
Service	ng g service my-new-service
Class	ng g class my-new-class
Interface	ng g interface my-new-interface
Enum	ng g enum my-new-enum
Module	ng g module my-module



## Fichiers et répertoires générés :

**package.json** : dépendances gérées par npm

**.editorconfig** : permet la configuration de l'IDE

**README.md** : présentation du projet (équivalent à lisez-moi.txt)

**.gitignore** : liste de fichiers ou répertoire ne devant pas être catalogués par GIT

**karma.conf.js** : fichier de configuration de Karma.

**protractor.conf.js** : fichier de configuration de protractor

**tslint.json** : règles d'analyse du code

**angular-cli.json** : fichier de configuration de l'application

**src** : fichiers de démarrages de notre application

**src/app** : code source de l'application

**src/assets** : images, polices d'écriture et autres éléments divers de notre application

**src/environments** : configuration de l'environnement de travail

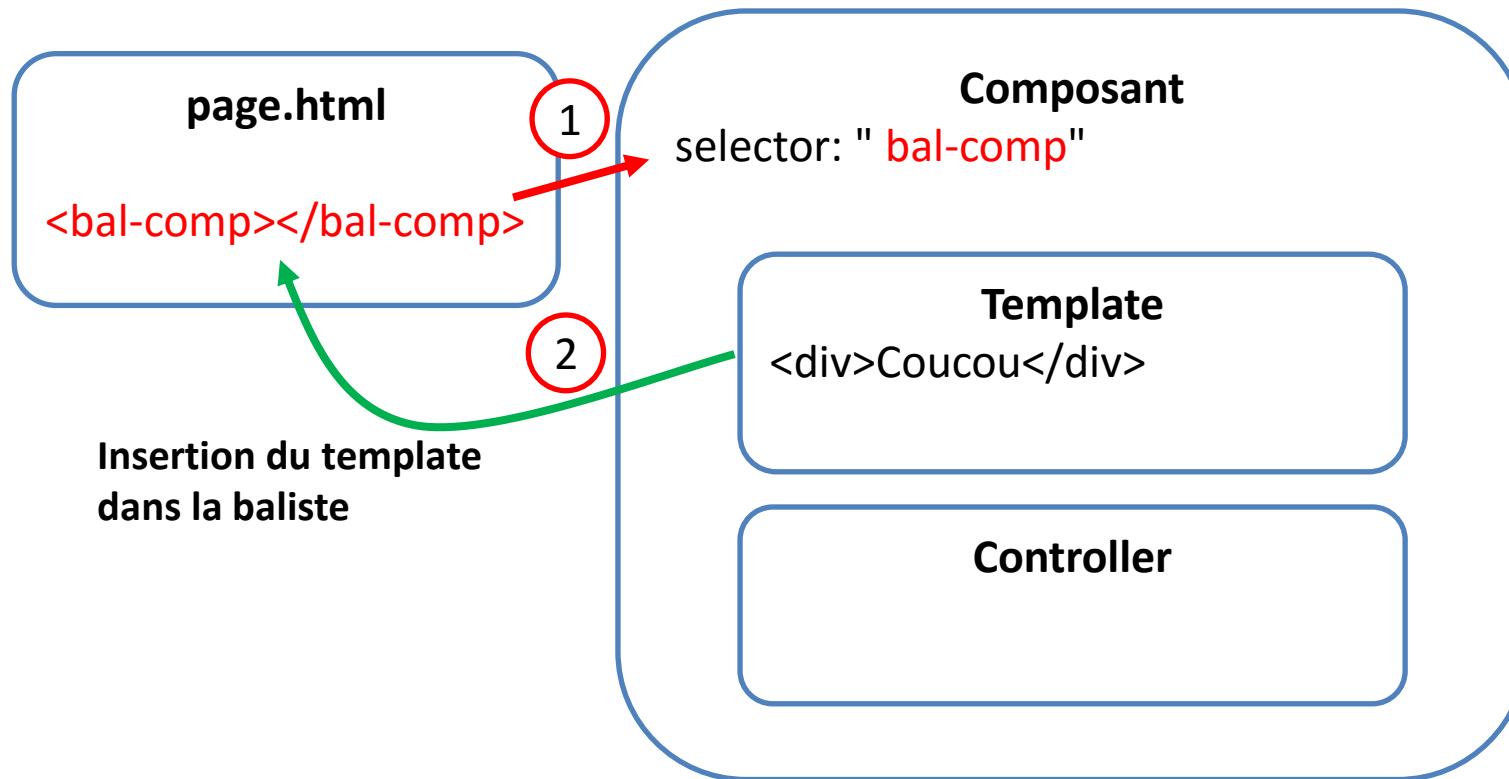
# Templates



# Templates

## Rappel :

- Pour appeler un composant dans une page html, il suffit d'utiliser le selecteur comme balise
- Le composant va insérer son template dans la balise correspondant au sélecteur.



# Templates : fichier séparé

Composant avec template séparé :

app.component.ts

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

```
@Component({  
  selector: app-comp',  
  templateUrl: './app.component.html'  
})
```

```
export class PremierComponent {  
}
```

app.component.html

```
<h1>Bonjour</h1>
```

# Templates : Afficher des valeurs

Afficher des valeurs depuis le contrôleur :

app.component.ts

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

@Component({
  selector: app-comp',
  templateUrl: './app.component.html'
})

export class PremierComponent {
  etudiants: number = 322;
  dir:any = { prenom: 'Adrien' };
}
```

app.component.html

```
<h1>Bonjour</h1>

<p> il y a {{etudiants}} élèves</p>
<p> le directeur se nomme : {{dir.prenom}}</p>
```

Résultat de app.component.html

```
<h1>Bonjour</h1>

<p> il y a 322 élèves</p>
<p> le directeur se nomme : Adrien </p>
```

# Templates : Afficher des valeurs

## Binding :

- Que l'on peut traduire par "lien"
- Le fait de lier une donnée entre le contrôleur et le template
- Il existe 4 types de "binding"



Template

`{{expression}}`

interpolation

`[propriété]="expression"`

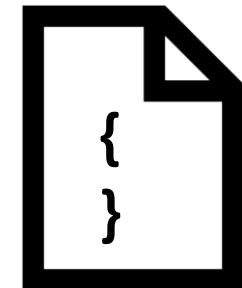
One Way Binding

`(événement)="instruction"`

Event Binding

`[(ngModel)]="propriété"`

Two Way Binding



Composant

# Templates : Binding One Way



**Interpolation et One Way Binding** sont identiques dans leur fonctionnement.

- Le contrôleur fournit une donnée
- Si le contrôleur modifie la donnée, la nouvelle valeur est affichée par le template
- Le template ne peut pas modifier la donnée

L'interpolation est un raccourcis d'écriture pour ce Binding One Way de l'attribut `textContent`

app.component.html avec **interpolation**

```
<h1>Bonjour<h1>  
  
<p> {{etudiants}} </p>  
<p> {{dir.prenom}}</p>
```

app.component.html avec **Binding One Way**

```
<h1>Bonjour<h1>  
  
<p [textContent] = "etudiants" > </p>  
<p [textContent] = "dir. prenom" ></p>
```



Si le template essaye d'afficher une **valeur inexiste**nte tel que :  
{{mauvaisNom}}, Angular la **remplace** par une **chaîne vide**.  
De même si la valeur vaut : **undefined et null**



Dans le cas **d'un attribut d'objet** définit dans le composant :  
**dir:any = { prenom: 'Adrien' };**  
Si nous tapons dans le template : {{dir.prenom}}  
Nous aurons une **erreur**

## Safe Navigation Operator" :



Dans le cas d'une **valeur** reçue de façon **asynchrone** (que nous recevront peut-être plus tard), il est possible de l'indiquer au template: {{dir?.prenom}}

# Templates : Binding One Way

## Binding One Way :

- Permet d'écrire dans n'importe quel attribut d'un élément du DOM

### Exemple d'attributs

```
<h1 class="titre" >Bonjour</h1>

```

### Avec Binding One Way

```
<h1 [class]="title" >Bonjour</h1>
<img [src]="imageUrl" >
```

### Composant

```
{
  title: string = "titre";
  imageUrl: string = " mon-image.jpg"
}
```



On préférera utiliser l'interpolation pour modifier du texte et le Binding One Way pour tous les autres attributs.

# Templates : Evénements

## Événement :

- Quelque chose qui peut se produire.
- On peut attendre que l'événement se produise (on l'écoute)

En JavaScript les événements sont par exemple : clic souris, souris qui vient de bouger, touche du clavier appuyé etc.

Ces événements peuvent être écoutés, et lorsqu'ils se produisent, exécuter une action grâce à des fonctions.

exemple.html

```
<h1 onclick="alert('coucou');"> Salut </h1>
```

template\_angular.html

```
<h1 (click)="afficheMessage('coucou')">Bonjour</h1>
```

composant.ts

```
...
export class MonComponent{
    afficheMessage(message: string): void {
        alert("mon message: " +message);
    }
}
```

# Templates : Evénements

## Fonctionnement d'un événement en Angular :

- Basé sur le patron de conception Observateur (voir Bibliographie)
- Les événements en Angular sont dit **bouillonnant**
- "bouillonnant" (bubbling up) indique que l'événement se propage vers son parent, qui lui-même le transmet à son parent, etc.

template\_angular.html

```
<div (click)="afficheMessage('parent')">  
    Message parent  
    <span (click)="afficheMessage('enfant')">Message enfant</span>  
</div>
```

affiche

Message parent Message enfant

Dans la span

Dans la div

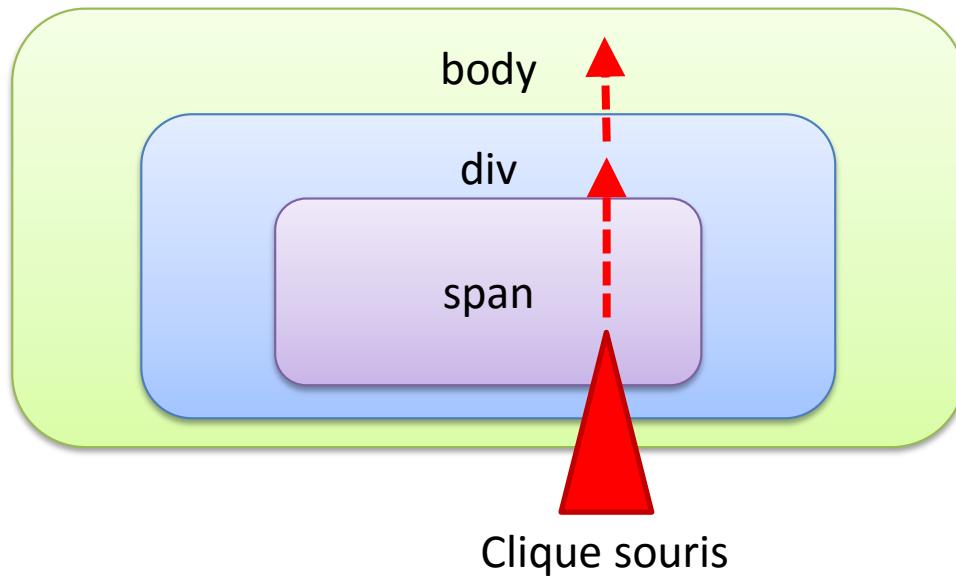
Si nous cliquons sur "Message parent", le message "parent" apparaît.

Si nous cliquons sur "Message enfant", le message "enfant" apparaît puis "parent".

# Templates : Evénements (bubble up)

## Bouillonnement :

- Transmission de l'enfant vers le parent



**Le contraire de bouillonnement, la capture d'événement.  
L'enfant récupère l'événement du parent.**

# Templates : Evénements (bubble up)

## Bouillonnement :

- Possible d'arrêter la transmission

Lors d'un événement, un objet est fabriqué, celui-ci contient l'information sur l'événement.

Pour le récupérer et arrêter sa propagation :

### template\_angular.html

```
<div (click)="afficheMessage('parent', $event)">
    Message parent
    <span (click)="afficheMessage('enfant', $event)">Message enfant</span>
</div>
```

### composant.ts

```
...
export class MonComponent{
    afficheMessage(message: string, event:any): void {
        alert("mon message: " +message);
        event.preventDefault();
        event.stopPropagation();
    }
}
```

Objet event envoyé

Action par défaut et  
bouillonnement  
arrêté



## Expression dans le binding :

- **Autorisés** : age , etudiant.prenom, operation entre plusieurs variables 2+5 et age \* 3 , opérateur ternaire ?:
- **Interdites** : assignation (=, +=, -=, ...) , opérateur new, virgule et pt virgule, ++ et –

## Instruction (statement) dans le binding :

- uneFonction()
- uneFonction(); uneAutre;

# Templates : variables locales



## Variable locale d'un template :

- Variable créée dans le templating
- La variable représente l'attribut dans lequel il est défini

template.html

```
<h1 #monTitre>Bonjour</h1>
{{monTitre.textContent}}
```

Mon titre représente l'objet DOM h1 dans lequel il est défini.

Nous avons donc accès à tous ses attributs et méthodes.

# Templates : balise template

## Rappel HTML5, balise template :

- Block servant à préparer des éléments mais à ne pas les afficher
- Utilisé avec le JavaScript pour récupérer son contenu lorsqu'il y a besoin de l'afficher
- On peut y voir un rapprochement entre Classe et objet en POO

template.html

```
<template>
  <ul>
    <li>message</li>
    <li>message</li>
    <li>message</li>
  </ul>
</template>
```

Ne sera pas  
affiché

# Templates : Directives

## Les directives :

- Sorte de composant sans template
- Permet de modifier le comportement des balises (donc de l'affichage d'un template)

**ngIf** : est une directive qui se comporte comme une condition "if"

template.html

```
<template [ngIf]="8 > 1">
  <ul>
    <li>message</li>
    <li>message</li>
    <li>message</li>
  </ul>
</template>
```

Les balises contenues  
dans le bloc  
s'afficheront que si la  
condition est vraie

composant.ts

```
...
export class MonComponent{
  age: number = 19;
}
```

template.html

```
<template [ngIf]="age > 18">
  <h2>coucou</h2>
</template>
```

On peut utiliser  
une variable

# Templates : Directives



## Écriture raccourcie :

- Ajouter une étoile \* devant la directive et retirer la balise template

template.html

```
<ul *ngIf="8 > 1">
    <li>message</li>
    <li>message</li>
    <li>message</li>
</ul>
```



La plupart des directives de base sont fournies par le module **BrowserModule** ou **CommonModule** et préchargées par Angular

# Templates : Directives

ngFor :

- Boucle de type foreach

composant.ts

```
...
export class MonComponent{
    quantites: number[] = [12, 189, 17, 12];
}
```

template.html

```
<ul>
    <li *ngFor="let valeur of quantites">{{valeur}}</li>
</ul>
```

affiche

12  
189  
17  
12

template.html

Affichage de l'indice

```
<ul>
    <li *ngFor="let valeur of quantites; let indice=index ">{{indice}} {{valeur}}</li>
</ul>
```

affiche

0 12  
1 189  
2 17  
3 12

Index est une variable fournie par ngFor, il en existe d'autres

# Templates : Directives

## ngSwitch :

- Condition switch
- Permet la sélection d'un template selon une valeur reçue

composant.ts

```
...
export class MonComponent{
    position : number= 2;
}
```

template.html

```
<div [ngSwitch]="position">
    <p *ngSwitchCase="1">Premier</p>
    <p *ngSwitchCase="2">Second</p>
    <p *ngSwitchDefault>Pas dans les premiers</p>
</div>
```

affiche

Second

# Templates : Directives

## ngStyle :

- Permet de modifier les propriétés CSS d'un élément

### template.html

```
<div [ngStyle]="{ color: 'red', fontWeight: 'bold' }">Texte en rouge et gras</div>
```

## ngClass :

- Permet de modifier les classes d'un élément

### template.html

```
<div [ngClass]="{'une-classe': true, 'une-autre': false}">  
    div avec des classes dynamiques  
</div>
```

En inspectant l'élément dans le navigateur, celui-ci aura la classe : "une-classe"

# Les pipes



# Les pipes: introduction

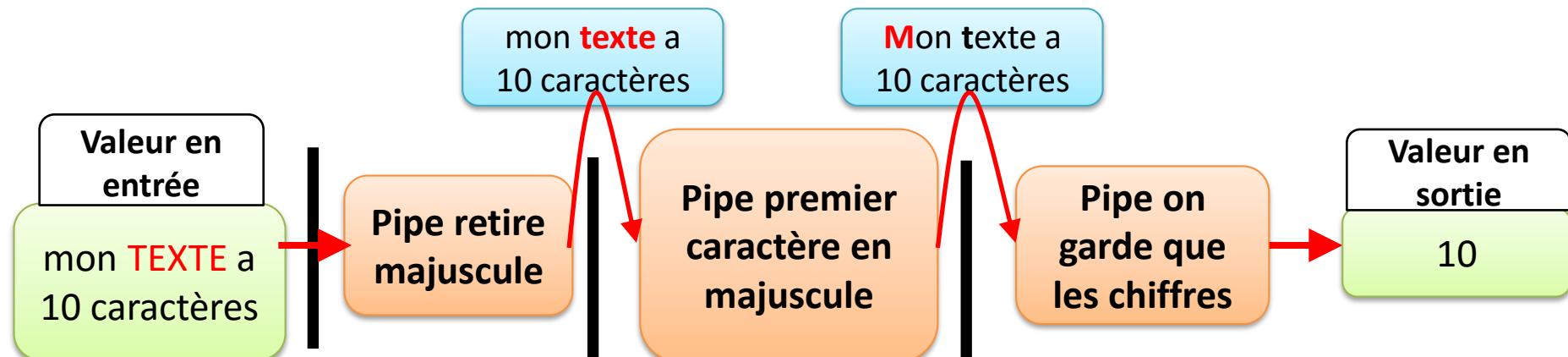


## Pipe :

- Anciennement filtre en AngularJS
- Le nom provient de sa syntaxe utilisant le caractère pipe |



Il est possible de chaîner les pipes



# Les pipes: exemple



## lowercase :

- Retire les majuscules d'un texte

template.html

```
 {{message | lowercase }}
```

composant.ts

```
...
export class MonComponent{
    message: string= "mon TEXTE à modifier";
}
```

## uppercase :

- Tous les caractères sont passés en majuscule

# Les pipes : avec paramètres



## Pipes:

- Il est possible de fournir des arguments

## Slice :

- Permet de découper une collection ou une chaîne de caractères

```
template.html  
{{ [2,9,10] | slice:0:1 }}
```

```
affiche  
2
```

Le premier paramètre '**0**' est le point de départ de ce qui va être gardé et '**1**' celui d'arrivée

# Les pipes : création

Pour créer un pipe:

- Implémenter **PipeTransform**, interface dans **@angular/core**
  - ✓ La méthode à implémenter est : **transform(value: any, ...args: any[]): any;**
- Utiliser le décorateur **@Pipe** en fournissant le nom du pipe à utiliser

getchar.pipe.ts

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

@Pipe({ name: 'getchar' })

export class GetCharPipe implements PipeTransform {
  transform(text: string) {
    return text[0]; // retourne le premier caractère d'une chaîne.
  }
}
```

Intégration du pipe  
dans la liste des  
éléments utilisable  
par le module:

```
import { GetCharPipe } from 'chemin/getchar.pipe.ts';
@NgModule({
  declarations: [GetCharPipe],
  ...
})
```

template

```
{{'mon text' | getchar}}
```

affiche

m

# Les pipes : création

Pour fournir des variables à un pipe:

- Les paramètres se trouvent de la valeur à filtrer

getchar.pipe.ts

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

@Pipe({ name: 'getchar' })

export class GetCharPipe implements PipeTransform {
  transform(text: string, pos?:number) {
    if( pos && text.length>=pos ) { // on teste le paramètre 'pos'
      return text[pos] // retourne le caractère en position choisie
    }
    return text[0]; // retourne le premier caractère d'une chaîne.
  }
}
```

template

```
{{'mon text' | getchar:2}}
```

affiche

n

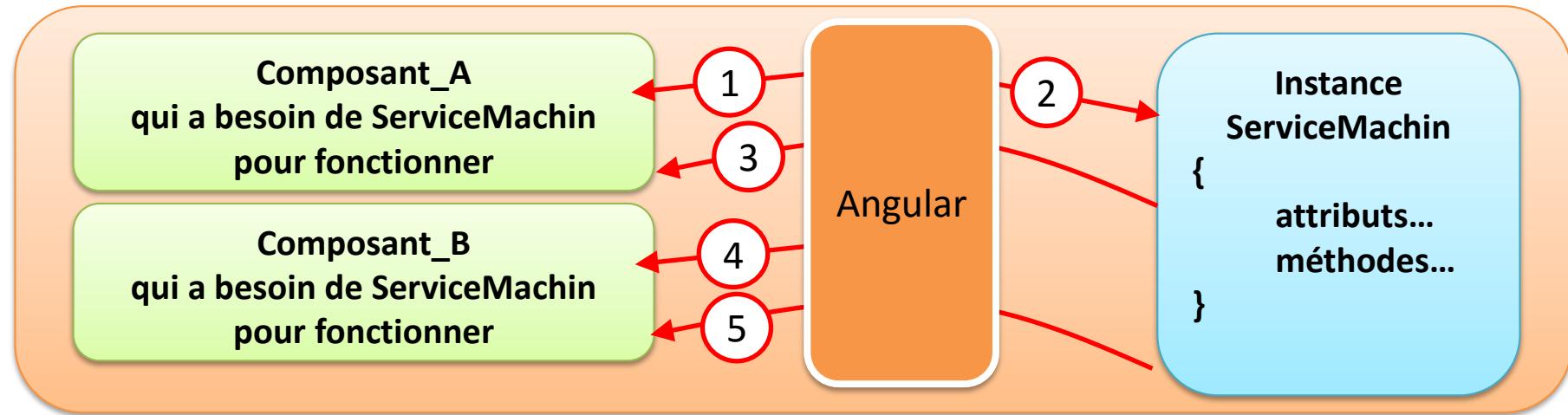
# Les services



# Les services

## Service :

- Objet qui peut être utilisé un peu n'importe où
- Angular les fournit lorsqu'une classe en a besoin (**injection**)
- Ce sont des singletons, ce qui veut dire que l'objet n'existe qu'en un seul exemplaire
- Si nous enregistrons des valeurs dans un singleton, elles seront partagées



- 1 – Le **Composant\_A** demande une instance de **ServiceMachin** pour fonctionner
- 2 – Angular recherche le ServiceMachin et crée une **instance de ServiceMachin**
- 3 – Le service est instancié, Angular le fournit à **Composant\_A** et peut maintenant fonctionner
- 4 – **Composant\_B** a besoin aussi de **ServiceMachin**
- 5 – Angular avait déjà une instance de prête de **ServiceMachin** et le fournit à **Composant\_B**

# Les services : création

Service :

exemple.service.ts

```
export class ExempleService {  
    constructor() {}  
    uneMethode() {  
        console.log("Méthode de mon service");  
    }  
}
```

Enregistré en tant que service

mon.module.ts

```
import { ExempleService }  
from './chemin/exemple.service';  
  
@NgModule({  
    providers: [ ExempleService ]  
})
```

mon.component.ts

```
import { Component } from '@angular/core';  
import { ExempleService } from '/chemin/exemple.service';  
  
@Component({  
    selector: 'mon-comp',  
    templateUrl: './mon.component.html'  
})  
  
export class MonComponent {  
    constructor(exService: ExempleService) {  
        exService.uneMethode();  
    }  
}
```

Demande d'injection par le type

# Les services : création

## Service :

- Un Service\_A peut demander l'injection d'un Service\_B
- Le Service\_B doit être enregistré dans le provider

exemple.service.ts

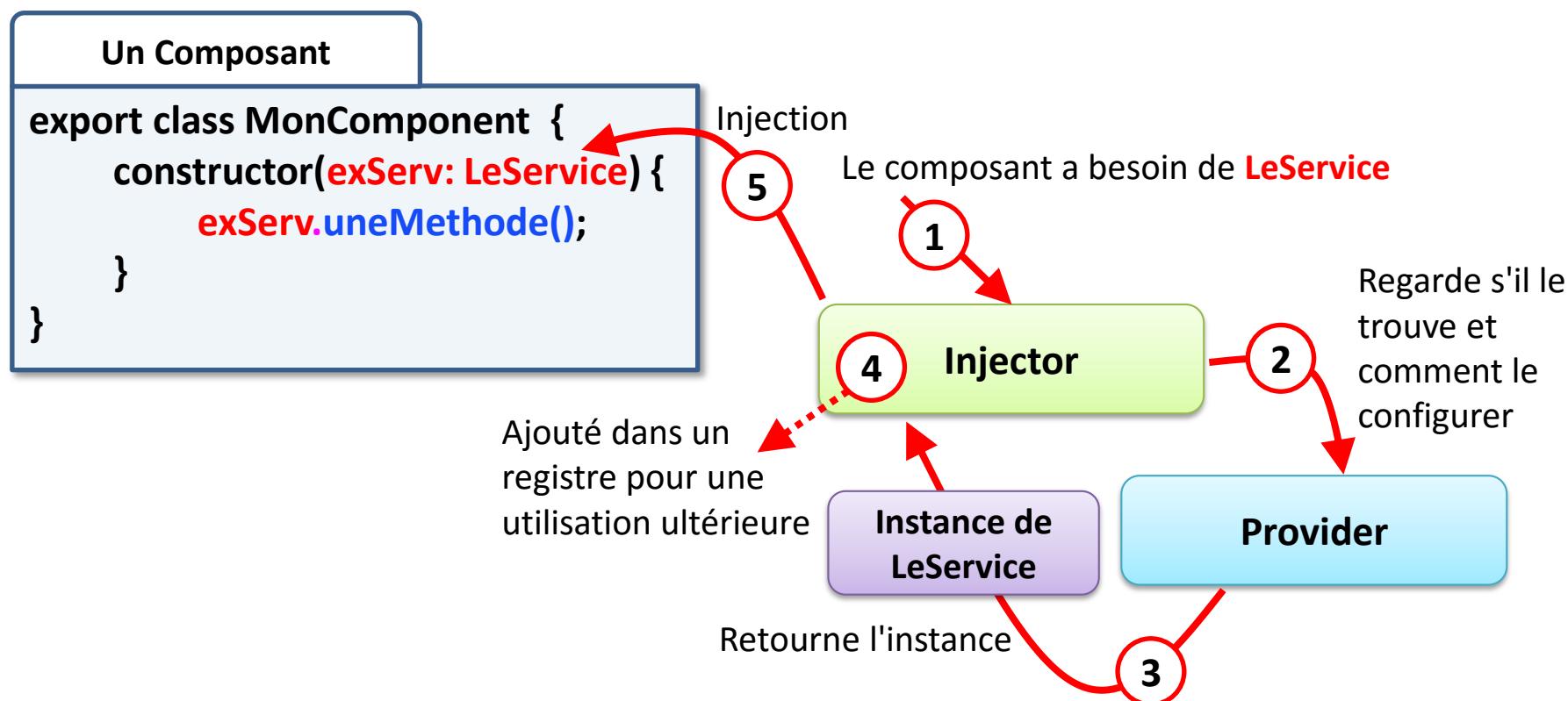
```
import { Injectable } from '@angular/core';
import { UnAutreService } from '/chemin/unautre.service';

@Injectable()
export class ExempleService {
    constructor(autre: UnAutreService) {
        autre.action();
    }
}
```

# Les services : Dans le framework

L'objet **Injector** (qui est un service) reçoit un **token "LeService"** qui correspond à une classe.

L'injecteur va instancier la classe correspondante en suivant les instructions indiquées par le provider



# Les services : Provider

**Provider** : fournisseur (généralement de services)

Renommer le **token** (nom du service) pour qu'il soit différent de celui de la classe :

```
import { ExempleService } from './chemin/exemple.service';

@NgModule({
  providers: [
    { provide: "NouveauNom", useClass: ExempleService }
  ]
})
```

Autre type de Provider :

```
providers: [
  { provide: "UNE_CONSTANTE", useValue: "Salut !"}
]
```

# Les services : Provider

Factory : fabrique et configure le service avant utilisation.

```
providers: [
  {
    provide: ExempleService

    useFactory: function() {
      let texte: string = "coucou";
      let service = new ExempleService();
      service.message = texte;
      return service;
    }
  }
]
```

# Les services : Provider

## Factory avec dépendance :

- Il est possible de récupérer un autre service ou une valeur définie dans les providers

```
providers: [
  { provide: "UNE_CONSTANTE", useValue: "Salut !",

  {
    provide: ExempleService,

    useFactory: function(UNE_CONSTANTE) {
      let texte: string = UNE_CONSTANTE;
      let service = new ExempleService();
      service.message = texte;
      return service;
    },
    deps: ["UNE_CONSTANTE"]
  }
]
```

# Les services : Provider

Il est possible de définir les services dans le composant directement.

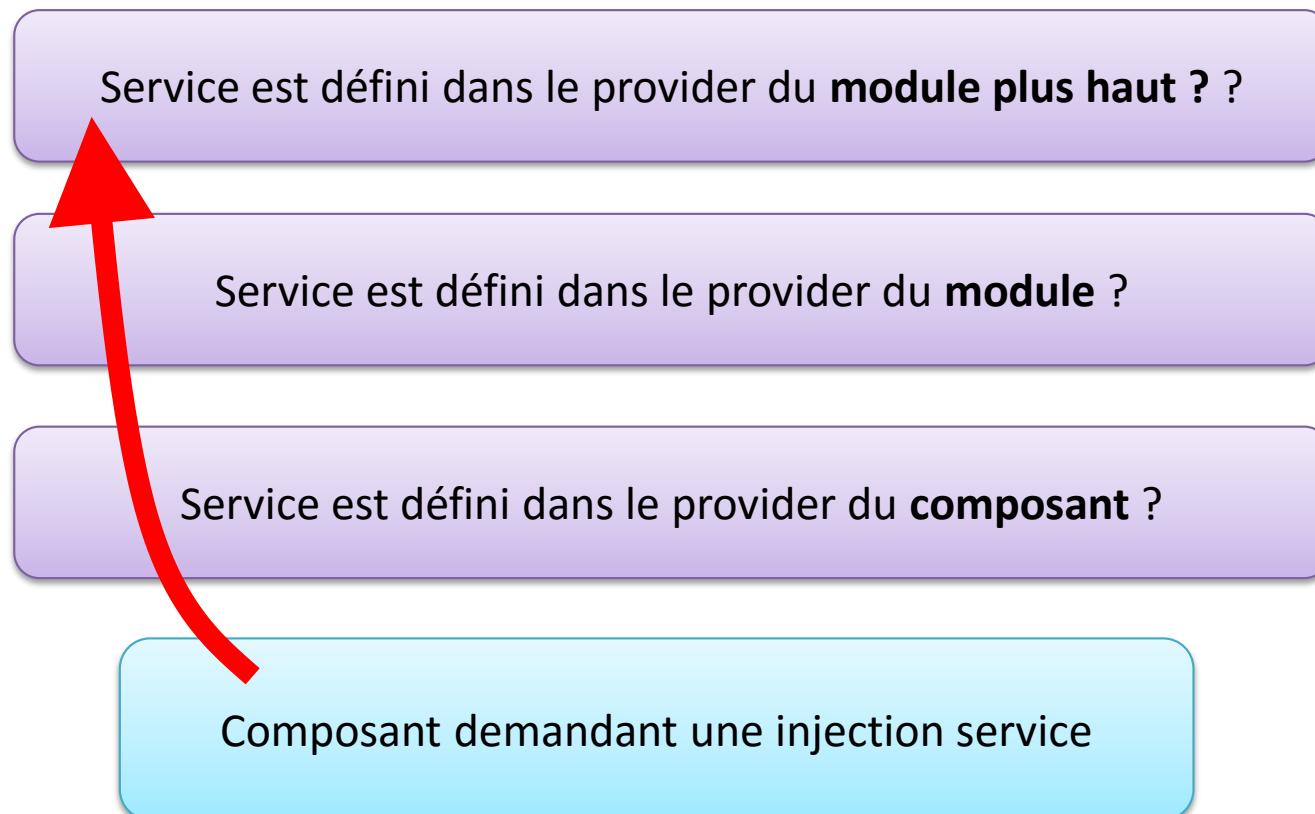
```
import { ExempleService } from './chemin/exemple.service';

@Component({
  selector: 'app-article',
  templateUrl: './article.component.html',
  providers: [
    { provide: ExempleService }
  ]
})
```

# Les services : Provider

Recherche de service dans les différents niveau de provider.

Un provider défini dans un composant est enfant de celui du module, ainsi de suite...



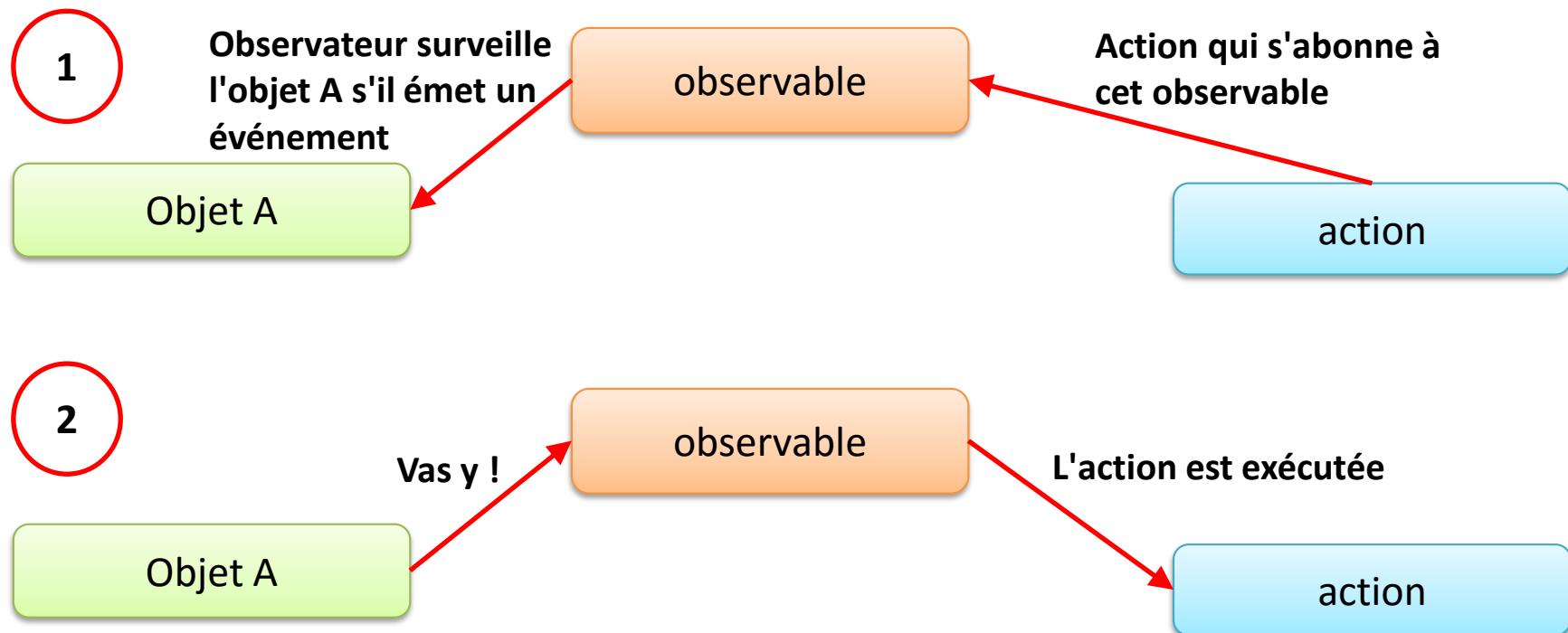
# Programmation réactive



# Programmation réactive

Programmation réactive (*reactive programming*) :

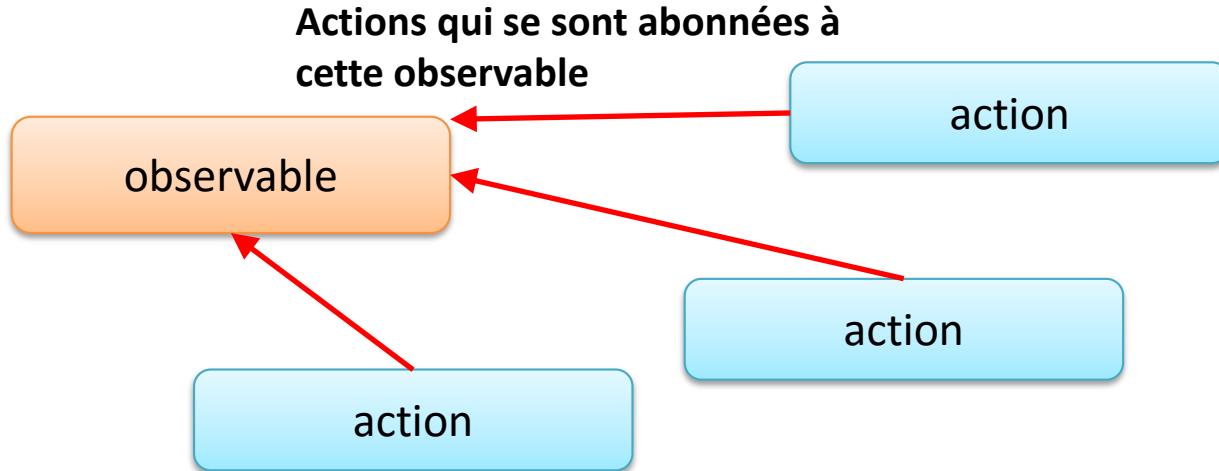
- Basé sur les événements
- Un objet A sera observé lorsque celui-ci émettra un signal (événement), l'observable va lancer des actions qui se sont abonnés



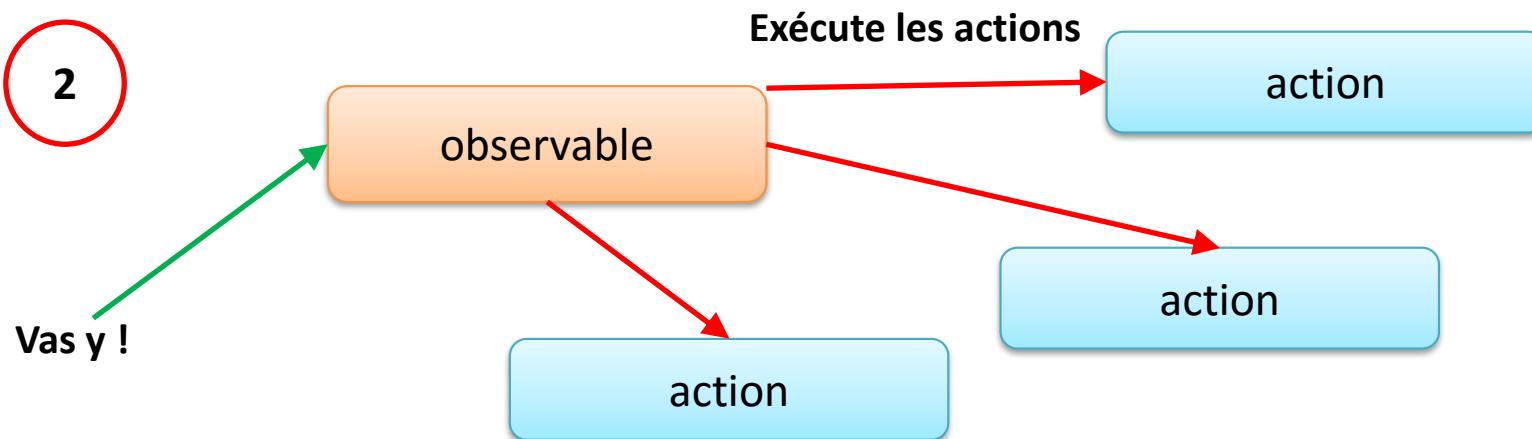
# Programmation réactive

Autre façon :

1



2





## Observable :

- Objet qui va récupérer l'événement et lancer des actions

## Quelques méthodes :

- map(fn) : applique une fonction **fn** à chaque événement et retourne la valeur
- filter(condition) : laisse passer l'événement sur la condition est vraie
- subscribe(fn) : exécute une fonction **fn** à chaque événement

# Programmation réactive

## RxJS

- Bibliothèque pour faire de la programmation réactive
- Documentation : <https://github.com/Reactive-Extensions/RxJS>
- Choisi par Angular
- On retrouve des bibliothèques équivalentes pour la plupart des langages

exemple.component.ts

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

@Component({
  selector: 'mon-comp',
  templateUrl: './exemple.component.html'
})

export class AppComponent {
```

```
  constructor(){
    Observable.from( [2, 10, 5, 8, 9] )
      .map(x => x * 3)
      .filter(x => x > 20)
      .subscribe(x => console.log(x));
  }
}
```

Création d'un observable sur un tableau  
Il va s'exécuter comme un flux de données entrant.

Chacune des valeurs va être lue et lancer un événement avec la valeur

On multiplie chacune des valeurs par 3

Si la valeur fois 3 ne dépasse pas 20, on la filtre.

Nous affichons les valeurs non filtrées

affiche

30 24 27

# Programmation réactive

Angular et RxJS :

- Angular intègre une surcouche à la bibliothèque RxJS
- L'objet Observable se nomme EventEmitter

exemple

```
import { Component, EventEmitter } from '@angular/core';
...
export class AppComponent {
  constructor {
    let obs = new EventEmitter();
    let subscribe = obs.subscribe (
      value => console.log(value),
      error => console.log(error),
      () => console.log('fin')
    );
    obs.emit('valeur1'); // affiche : valeur1
    obs.emit('valeur2'); // affiche : valeur1
    obs.complete(); // affiche : fin
    subscribe.unsubscribe(); // retire toutes les méthodes
    obs.emit('valeur3'); // ne provoque rien
  }
}
```

# Les Directives



# Les directives



## Directive :

- Brique d'Angular qui va interagir avec le template

Une directive utilise un sélecteur CSS pour indiquer son point d'activation

- **div** : pour sélectionner les éléments HTML de type **div**
- **.titre** : pour sélectionner tous les éléments ayant **class="titre"**
- **[src]** : éléments ayant l'attribut **src=""**
- **[attr=valeur]** : éléments ayant un attribut d'une certaine valeur
- **Possibilité d'utiliser une combinaison CSS** : **div.titre** (**div** ayant une classe **titre**)

# Les directives : création

Exemple :

color.directive.ts

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

@Directive({ selector: '[atcolor]'})

export class ColorDirective {
    constructor() {
        console.log("Je m'execute à chaque élément ayant l'attribut atcolor ");
    }
}
```

mon.module.ts : ajout dans les éléments utilisable par le module

```
import { ColorDirective } from './chemin/color.directive';
@NgModule({
    declarations: [ColorDirective],
    ...
})
```

template

```
<h2 atcolor>Ma directive</h2>
<div atcolor></div>
```

Dans la console du navigateur :  
Je m'execute à chaque élément ayant l'attribut atcolor  
Je m'execute à chaque élément ayant l'attribut atcolor

# Les directives : création avec entrées

Exemple avec paramétrage :

color.directive.ts

```
import { Directive } from '@angular/core';
@Directive({
  selector: '[atcolor]',
  inputs: [ 'conf: atcolor ', 'couleurs: colors' ]
})
export class ColorDirective {
  constructor() {
    console.log("start");
  }
  set conf(value) {
    console.log(value);
  }
  set couleurs(value) {
    console.log(value);
  }
}
```

inputs: [ 'conf: atcolor ', 'couleurs: colors' ]

Nous choisissons un ou plusieurs attributs qui va contenir nos paramètres.  
La valeur de atcolor sera transmis au setter conf et celui de colors à couleurs

Template.html

```
<div atcolor="truc" colors="rouge"></div>
<h2 atcolor="une valeur" >Ma directive</h2>
```

Dans la console du navigateur :

```
start
truc
rouge
start
une valeur
```

# Les directives : création avec entrées

color.directive.ts

```
import { Directive } from '@angular/core';
@Directive({
  selector: '[atcolor]',
  inputs: ['colors']
})
export class ColorDirective {
  constructor() {}
  set colors(value) {
    console.log(value);
  }
}
```

Écriture plus concise

Template.html

```
<div colors="rouge" atcolor></div>
```

Il est possible d'utiliser une expression (pipes comprises)

Template.html

```
<div atcolor="{{proprieteComposant}}" colors="{{'Rouge' | lowercase}}></div>
```

# Les directives : création avec entrées

Nouvelle écriture (plus simple):

color.directive.ts

```
import { Directive, Input } from '@angular/core';
@Directive({
  selector: '[atcolor]'
})

export class ColorDirective {
  constructor() {
  }
  @Input()
  set colors(value) {
    console.log(value);
  }
}
```

Template.html

```
<div colors="rouge" atcolor></div>
```

# Les directives : Modification de balise

## Directive et balise :

- La directive peut modifier la balise qui l'appel
- Nous importons  **ElementRef** qui permet de travailler directement avec les Elements DOM

color.directive.ts

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

@Directive({
  selector: '[atcolor]'
})

export class ColorDirective {
  el: ElementRef;
  constructor(el: ElementRef) {
    this.el = el;
  }
  @Input()
  set colors(value) {
    this.el.nativeElement.style.backgroundColor = value;
  }
}
```

Template.html

```
<h2 colors="red" atcolor> mon texte</h2>
```

1

Template.html

2

```
<h2 colors="red" atcolor> mon texte</h2>
```

3

1 – L'attribut atcolor indique qu'il faut exécuter cette directive

4

2 – Lors de la création de la directive, Angular fournit l'élément DOM qui l'a appelé

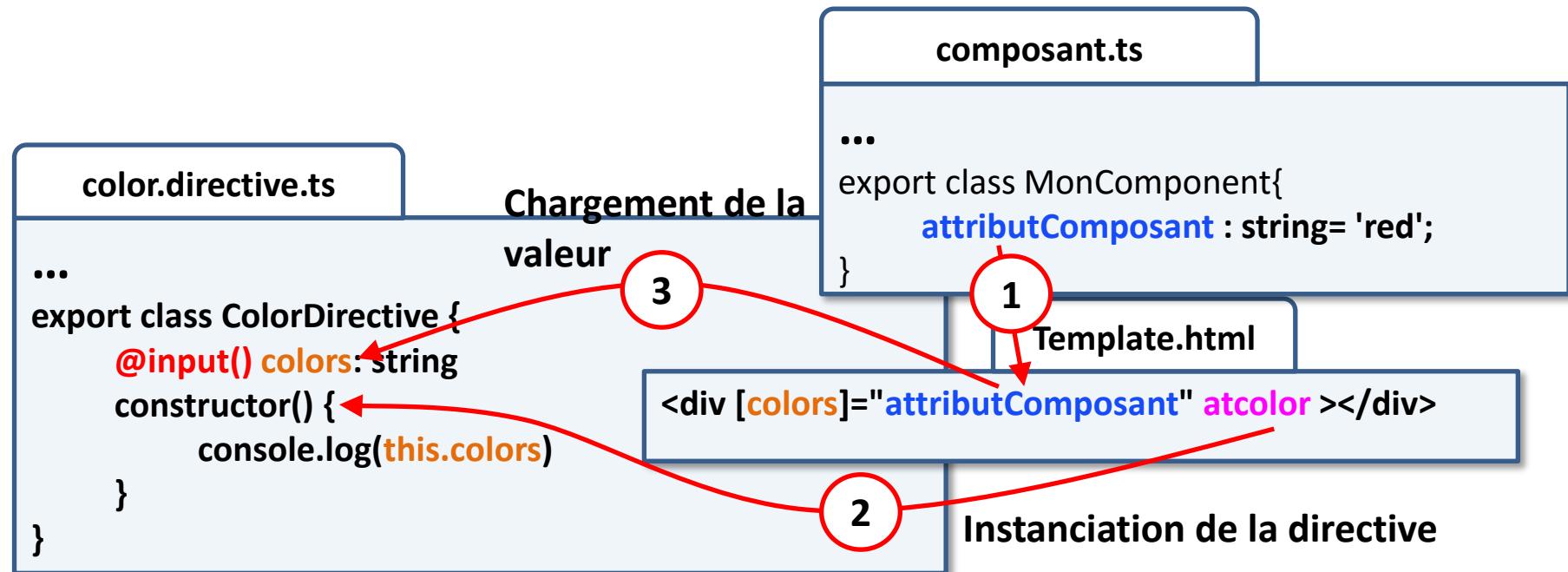
3 – Angular trouve l'attribut colors et fournit sa valeur au setter

4 – Changement de couleur de l'élément.

# Les directives : Cycle de vie

## Bind de valeurs :

- Il est possible de lier la valeur d'un attribut DOM à un de la directive



`console.log(this.colors)` affiche **undefined** car au moment de l'instanciation (appel au constructeur), Angular n'a pas encore récupéré la valeur de **colors** !

# Les directives : Cycle de vie

## Cycle de vie :

- Le cycle de vie d'une directive passe par des appels de méthodes bien définies

Les méthodes sont appelées par Angular à certains moments lors du fonctionnement de la directive après son instanciation.

ngOnChanges(objAttribut) : Au chargement et au changement des valeurs des attributs

ngOnInit() : Lorsque tous les attributs sont chargés

color.directive.ts

```
export class ColorDirective {  
    @input() colors: string  
    constructor() {  
        console.log("Directive chargée");  
    }  
    ngOnChanges(objetAttribut) {  
        console.log('Attribut chargé', objetAttribut );  
    }  
    ngOnInit() {  
        console.log('Tous les attributs sont chargés', this.colors);  
    }  
    ngOnDestroy() {  
        console.log('La directive est détruite');  
    }  
}
```

<div [colors]=""red"" atcolor></div>

Template.html

affiche

Directive chargée

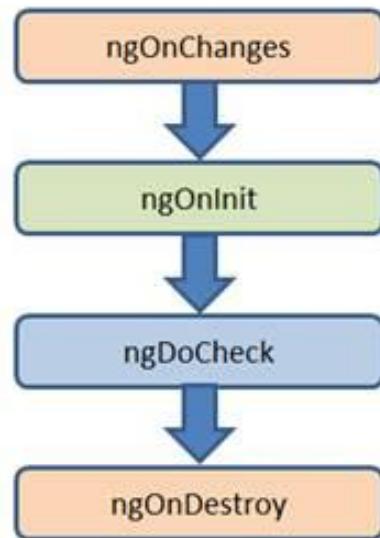
Attribut chargé , Object {colors: ...}

Tous les attributs sont chargés, red

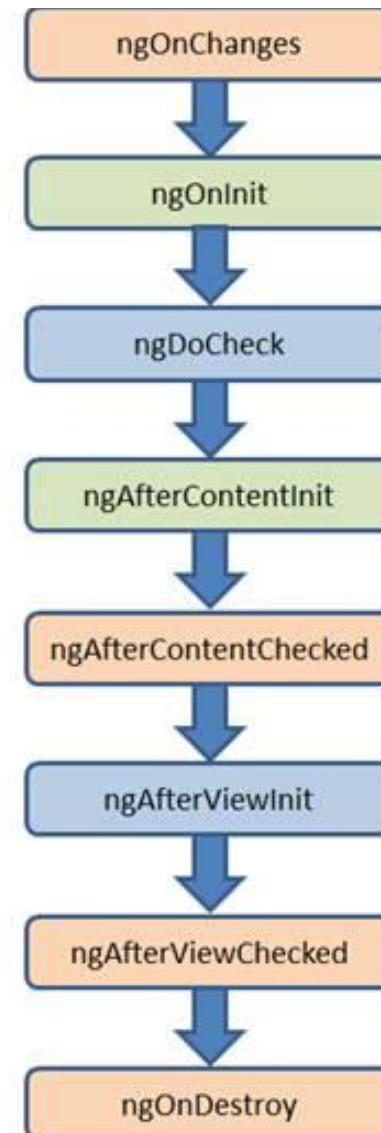
# Cycle de vie

## Cycle de vie :

### Cycle de vie pour les directives structurelles



### Cycle de vie des composants



# Les directives : Provider

Les directives tout comme les modules et les composants peuvent avoir leur propre provider

## Décorateur d'une directive

```
@Directive({
    selector: 'div',
    providers: [ExempleService]
})
```

# Composant : Style et template



# Composant : Style et template



Les directives et composants sont très proches

Un composant **doit avoir** un **template**.

A ce template peut s'ajouter du CSS.

## Décorateur d'un composant

```
@Component({  
    selector: 'sel-comp',  
    template: '<h2>coucou</h2> <div class="truc">une div</div>',  
    styles: ['h2{ color: red; }', '.truc{background: green;}']  
})
```

## Décorateur d'un composant avec utilisation de fichiers

```
@Component({  
    selector: 'app-article',  
    templateUrl: './article.component.html',  
    styleUrls: [ './article.component.css' ]  
})
```

# Composant : Encapsulation du style



## Le CSS d'un composant a 3 niveaux d'encapsulations

- **ViewEncapsulation.Emulated** : Valeur par défaut. Angular va ajouter un sélecteur pour que le CSS ne s'applique qu'au template du composant
- **ViewEncapsulation.Native** : Place le CSS dans le Shadow DOM, il ne s'applique qu'au template du composant sans changer les sélecteur
- **ViewEncapsulation.None** : Le CSS n'est pas encapsulé et s'applique à toute la page

### Décorateur d'un composant avec utilisation de fichiers

```
@Component({  
    selector: 'app-article',  
    templateUrl: './article.component.html',  
    styleUrls: [ './article.component.css' ],  
    encapsulation: ViewEncapsulation.Emulated  
})
```

# Formulaires et Binding Two Way



# Formulaires :



La validation des formulaires peut se faire :

- dans le template (pilotage par le template)  
On utilise les directives
- dans le composant (pilotage par le code)  
On définit le comportement des champs en se servant des objets FormControl et FormGroup que l'on liera par des directives dans le template

# Formulaires :



La validation des formulaires peut se faire :

- dans le template (pilotage par le template)  
On utilise les directives
- dans le composant (pilotage par le code)  
On définit le comportement des champs en se servant des objets FormControl et FormGroup que l'on liera par des directives dans le template

# Formulaires : Pilotage template

## Pilotage par template :

- Ajouter le module : **FormsModule**

### Configuration du module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
```

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

### composant.ts

```
@Component({
  selector: 'app-form',
  templateUrl: './form.component.html',
  styleUrls: ['./form.component.css']
})
export class FormComponent {
  constructor() { }
  register(value) {
    console.log("soumission", value);
  }
}
```

# Formulaires : Pilotage template

Form dans un template est une directive

Capture de l'événement "submit" du formulaire et transmission à la méthode "register" du composant

```
template.html
<form (ngSubmit)="register()">
  <div>
    <label>Login</label>
    <input name="login" ngModel>
  </div>
  <div>
    <label>Password</label>
    <input type="password" name="password" ngModel>
  </div>
  <button type="submit">Envoyer</button>
</form>
```

Le bouton submit émet l'événement "submit" du formulaire



**Ne pas oublié les attributs `name`, il permet de lier la valeur d'un champ à un nom.**

# Formulaires : Pilotage template

La balise Form va créer un objet FormGroup qui contient toutes les champs du formulaire.

Les champs à remplir avec l'attribut ngModel vont créer des objets FormControl contenant les informations du champ.

Les FormControl seront ajouté dans FormGroup.

**template.html : Transmission des valeurs au composant**

```
<form (ngSubmit)="register(monForm.value)" #monForm="ngForm" >
  ...
</form>
```

The code block shows a snippet of template.html. It contains a form element with an ngSubmit event binding to a register method and a local variable monForm set to ngForm. Three numbered callouts point to specific parts of the code: 1 points to the #monForm="ngForm" directive, 2 points to the ngForm directive, and 3 points to the monForm.value attribute.

1 - Lors de l'utilisation de ngForm dans un formulaire, la directive form la remplace par une référence à FormGroup

2 – Une variable locale enregistre la référence de FormGroup

3 – La valeur de FormGroup est fournie à la méthode du composant

# Formulaires : Avec Binding Two Way

Nous pouvons avoir un résultat équivalent avec le Binding Two Way

Le Binding Two Way lie le template et le composant dans les deux sens.

La modification de log et pwd que ce soit dans le template ou le composant sera répercuté des deux côtés.

Template.html

```
<form (ngSubmit)="register()">
  <div>
    <label>Login</label>
    <input name="login" [(ngModel)]="log">
  </div>
  <div>
    <label>Password</label>
    <input type="password" name="password" [(ngModel)]="pwd">
  </div>
  <button type="submit">Envoyer</button>
</form>
```

composant.ts

```
export class FormComponent {
  log: string;
  pwd: string;

  constructor() { }

  register() {
    // Méthode appelé lors de la soumission
    console.log(this.log, this.pwd);
  }
}
```

# Formulaires : Avec Binding Two Way

## Démonstration Binding Two Way

### Template.html

```
<div>
  <input name="login" [(ngModel)]="log">
  {{ log }}
</div>
<div>
  <input type="password" name="password" [(ngModel)]="pwd">
  {{ pwd }}
</div>
```

### composant.ts

```
export class Component {
  log: string;
  pwd: string;
  constructor() { }
```

adrien

• • • • •

adrien

secret

# Formulaires : Pilotage par le code



## Formulaire piloté par le code :

- Plus verbeux mais apportant plus de possibilité

## FormControl :

- Classe représentant un champ à remplir

Attribut	Description
valid	true si la valeur est correcte
errors	liste des erreurs (objet)
dirty	true si l'utilisateur modifie le champ
pristine	false si l'utilisateur modifie le champ
touched	true si l'utilisateur atteint le champ
untouched	false si l'utilisateur atteint le champ
value	valeur du champ
utilisation	valueChanges Observable pour les changements de valeur

```
const password = new FormControl();
const login = new FormControl("adrien");

console.log(password.value); // affiche null
console.log(login.value); // affiche : adrien
```

# Formulaires : Pilotage par le code



## FormGroup :

- Classe représentant un groupe de champs à remplir

Attribut	Description
valid	true si tous les champs ont des valeurs correctes
errors	objet avec toutes les erreurs des champs sinon null
dirty	true si l'utilisateur modifie un des champ
pristine	false si l'utilisateur modifie un des champ
touched	true si l'utilisateur atteint un des champ
untouched	false si l'utilisateur atteint un des champ
value	liste des champs et de leurs valeurs
valueChanges	Observable sur les changements des champs

get("nomDuControl") : retourne le FormControl demandé

### utilisation

```
const form = new FormGroup({  
    login: new FormControl( "adrien" ),  
    password: new FormControl()  
});
```

```
const fc:FormControl = form.get('login');
```

# Formulaires : Pilotage par le code



## FormBuilder :

- Classe utilitaire injectable
- Simplifie la création des FormGroup/FormControl

composant.ts

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'register-comp',
  templateUrl: 'register.component.html'
})

export class FormComponent {
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      login: "adrien",
      password: ""
    });
  }
}
```

Injection par Angular de FormBuilder

**FormBuilder Fournit une méthode control() :**

```
this.userForm = new FormGroup({
  login: fb.control( "adrien" ),
  password: fb.control()
});
```

# Formulaires : Pilotage par le code

Lier le formulaire et le composant :

template.html

```
<form (ngSubmit)="register()" [formGroup]="userForm">  
  <input formControlName="login">  
  <input type="password" formControlName="password">  
  
  <button type="submit">Envoyer</button>  
  
</form>
```

1

composant.ts

```
export class FormComponent {  
  userForm: FormGroup;  
  
  constructor(fb: FormBuilder) {  
    this.userForm = fb.group({  
      login: "adrien",  
      password: ""  
    });  
  }  
  
  register() {  
    console.log(this.userForm.value);  
  }  
}
```

module.ts

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';  
  
@NgModule({  
  declarations: [FormComponent],  
  imports: [FormsModule, ReactiveFormsModule ],  
})  
export class UnModule { }
```

# Formulaires : Pilotage par le code

Lier le formulaire et le composant :

**formGroup / formControlName :**

- directives de **ReactiveFormsModule**
- Permet de lier des champs aux **FormControl** d'un **FormGroup**

# Formulaires : Validation par code



**Validation pour un formulaire piloté par le code :**

- Classe Validator

Contrainte	Description
Validators.required	La valeur ne doit pas être vide
Validators.minLength(n)	La valeur doit faire n caractères
Validators.maxLength(n)	La valeur ne doit pas dépasser n caractères
Validators.pattern(p)	La valeur doit respecter une regex

# Formulaires : Validation par code

Validation pour un formulaire piloté par le code :

composant.ts

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({ ... })
export class FormComponent {
    userForm: FormGroup;

    constructor(fb: FormBuilder) {
        this.userForm = fb.group({
            login: fb.control( "", [ Validators.required, Validators.minLength(5) ]),
            password: fb.control( "", Validators.required)
        });
    }
    register() {
        console.log(this.userForm.valid);
    }
}
```

# Formulaires : Validation par template



Validation pour un formulaire piloté par le template :

template.html

```
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">

    <input name="username" ngModel required minlength="3">

    <input type="password" name="password" ngModel required>

    <button type="submit">Envoyer</button>
</form>
```

# Formulaires : Validateur personnalisée



## Validateur personnalisé :

- Fonction recevant en paramètre un objet FormControl
- Retourne un objet {nomErreur: boolean} en cas d'erreur
- Retourne null si la valeur passe la validation

composant.ts

```
const majeur = (fc: FormControl) => {
    return fc.value >= 18 ? null : {mineur: true};
}

this.age = fb.control("", [Validators.required, majeur]);
```

# Requêtes HTTP



# Requête HTTP :

## Requête HTTP :

- Utilisation du module HttpModule pour faire les requêtes
- C'est le service Http qui fournit le requétage Ajax

module.ts

```
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/http';

@NgModule({
  imports: [HttpClientModule]
})
export class AppModule { }
```

composant.ts

```
@Component({ ... })

export class PonyRacerAppComponent {
  constructor(http: Http) {
  }
}
```

# Requête HTTP :

Méthodes fournies par le service Http

Méthode	Description
<b>get</b>	Demander une ressource au serveur (exemple: un fichier, une page html, un flux JSON, etc.) Généralement en tapant une adresse ou en suivant un lien.
<b>post</b>	Créer une ressource
<b>put</b>	Ajout ou modification d'une ressource
<b>delete</b>	Supprimer une ressource sur le serveur
<b>head</b>	Demande d'information sur une ressource.
<b>patch</b>	Modification d'une ressource

# Requête HTTP :

Les méthodes de requêtage du service Http retourne un Observable.

## Exemple

```
constructor(http: Http) {  
  
    http.get("https://na14.salesforce.com/services/data/")  
        .subscribe( response) => {  
            console.log(response.status); // code http  
            console.log(response.headers); // en-tête  
            console.log(response.json() ); // corps  
        });  
  
}
```

Si nous n'attendons pas du json, utiliser la méthode text() de la réponse.

Pour envoyer des données, cela doit être sous forme de chaîne de caractères :

- `http.post("url", JSON.stringify({ nom:adrien }) )`

# Requête HTTP :

Utiliser des options pour la requête :

- Utiliser un objet RequestOptions
- RequestOptions reçoit un objet lors de son instantiation ayant les attributs :  
**{method, headers, body, url, search, withCredentials, responseType}**

Attribut	Description
method	méthode HTTP
headers	objet de type Headers du module @angular/http
body	information à envoyer
url	url de la requête
search	Ajoute des paramètres de recherche à l'url
withCredentials	Booléen (requête contenant des habilitations)
responseType	Le type de format attendu en retour

# Requête HTTP :

Exemple :

## Exemple

```
constructor(http: Http) {  
  
    const options = new RequestOptions ( {method: "get"} );  
  
    http.request("https://na14.salesforce.com/services/data/", options)  
        .subscribe( response) => {  
            console.log(response.status); // code http  
            console.log(response.headers); // en-tête  
            console.log(response.text() ); // corps  
        );  
}  
}
```

# Les routes





## Les routes :

- Angular est un framework mono-pagé, il simule donc le changement de page
- Selon l'URL, Angular fera le lien avec la page à afficher
- Le module **RouterModule** est en version 3.0 contrairement aux autres qui sont en 2
- Il est responsable du futur passage d'Angular de la version 2 à 4 pour aligner les versions

# Les routes : création

Il est conseillé de mettre les routes dans un fichier séparé.

Nous fabriquons un tableau de type Routes contenant la liste des "urls -> composant"

**app.routes.ts : contient la configuration des routes**

```
import { Routes } from '@angular/router';
import { TestComponent } from './components/test/test.component';
import { TrucComponent } from './components/truc/truc.component';

export const ROUTES: Routes = [
  { path: "", component: HomeComponent },
  { path: 'film', component: FilmComponent }
];
```

Url tel que : **monsite.com/film**

# Les routes : création

Il faut charger le module principal avec :

- le module Router,
- le fichier de config
- les composants utilisés dans les routes.

## app.module.ts : configuration du module principal

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

import { RouterModule } from '@angular/router';
import { ROUTES } from './app.routes';

import { HomeComponent } from './home/home.component';
import { FilmComponent } from './film/film.component';

@NgModule({
  imports: [BrowserModule, RouterModule.forRoot(ROUTES)],
  declarations: [AppComponent, HomeComponent, FilmComponent],
  bootstrap: [AppComponent]
})

export class AppModule {
```

# Les routes : création

Dans le template du composant principal nous allons placer une balise spéciale :  
**<router-outlet></router-outlet>**

## app.component.html

```
<header>
  <span>En-tête du site</span>
</header>

<main>
  <b><router-outlet></router-outlet></b>
</main>

<footer>
  <span>pied de page</span>
</footer>
```

En-tête du site

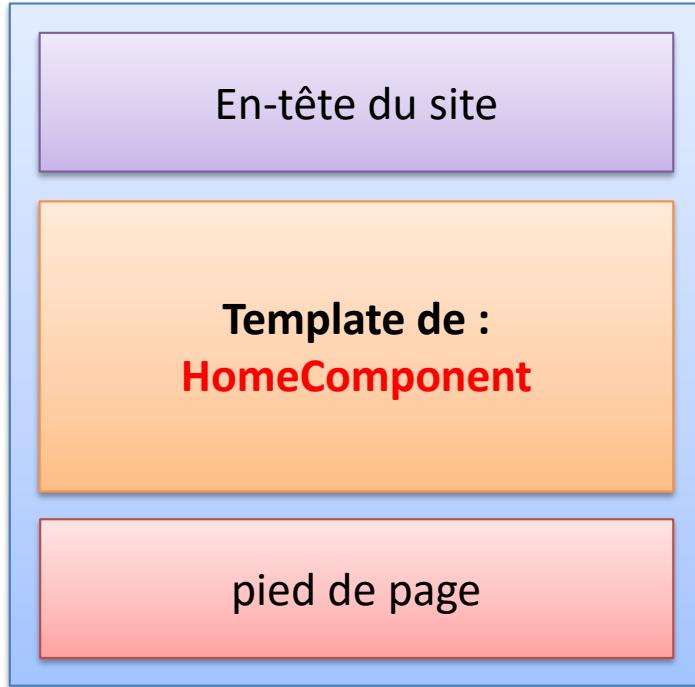
Le corps, qui sera l'élément qui  
changera d'une url à l'autre

pied de page

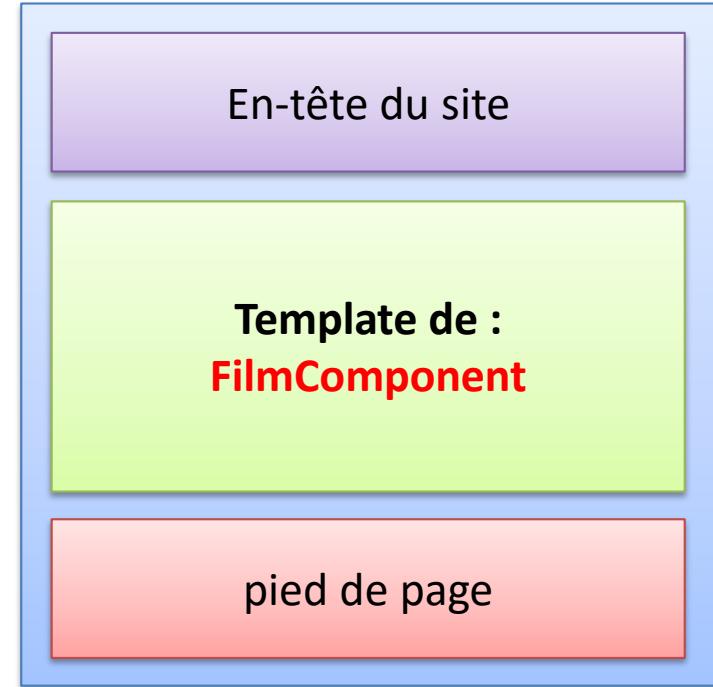
# Les routes : création

Résultat qui sera attendu :

url : site.com



url : site.com/**film**



# Navigation



## Navigation dans le template :

- Lien menant d'une page à l'autre
- **Eviter les `<a href="url"></a>` car recharge l'application. MAL !**

- Utiliser la directive routerLink :

```
<a href="" routerLink="/url">Ma page</a>  
Ou  
<a href="" [routerLink]=["/url"]>Ma page</a>
```



En considérant que nous sommes sur la page : **monsite.com/film**  
L'url : `/livre` renvoie sur **monsite.com/livre**  
L'url : `livre` renvoie sur **monsite.com/film/livre**

## Navigation par le code :

- Utiliser le **service Router**
- La méthode **navigate** de Router permet la redirection. Prend un tableau en argument
- `router.navigate(['/voiture']);` redirige vers : monsite.com/voiture

# Navigation : URL avec paramètres



## URL avec paramètres :

- URL dynamiques
- Aller sur la page du film dont l'id est 18 : monsite.com/film/**18**

La création de la route se fait ainsi :

### app.routes.ts : contient la configuration des routes

```
export const ROUTES: Routes = [
  { path: "", component: HomeComponent },
  { path: 'films', component: FilmsComponent },
  { path: 'films/:filmId', component: FilmComponent }
];
```

L'URL se note ainsi :

```
<a href="" [routerLink]="/films", film.id>Terminator</a>
```

# Navigation : URL avec paramètres



## URL avec paramètre:

- Le composant d'une route peut récupérer par injection **ActivatedRoute** qui contient les informations sur la route
- Utile pour récupérer les paramètres dynamiques

composant.ts

```
export class FilmComponent implements OnInit {
    constructor(private route: ActivatedRoute) {
    }

    ngOnInit() {
        const id = this.route.snapshot.params["filmId"];
    }
}
```



Nous aurons la même instance du composant si nous passons de la page **/films/2** à **/films/3**

Dans ce cas, ngOnInit ne sera plus appelé, ni le constructeur

# Navigation : URL avec paramètres



## URL avec paramètre:

- Il est possible de s'abonner à un observable à chaque changement de paramètres

composant.ts

```
export class FilmComponent implements OnInit {
    constructor(private route: ActivatedRoute) {}

    ngOnInit() {
        this.route.params.subscribe(params => {
            console.log("changement des paramètres", params);
        });
    }
}
```

# Les routes : page 404

Mettre en place la page 404 :

**app.routes.ts : contient la configuration des routes**

```
import { Routes } from '@angular/router';
import { TestComponent } from './components/test/test.component';
import { TrucComponent } from './components/truc/truc.component';

export const ROUTES: Routes = [
    { path: "", component: HomeComponent },
    { path: 'film', component: FilmComponent }
    {path: '404', component: NotFoundComponent},
    {path: '**', redirectTo: '/404'}
];
```

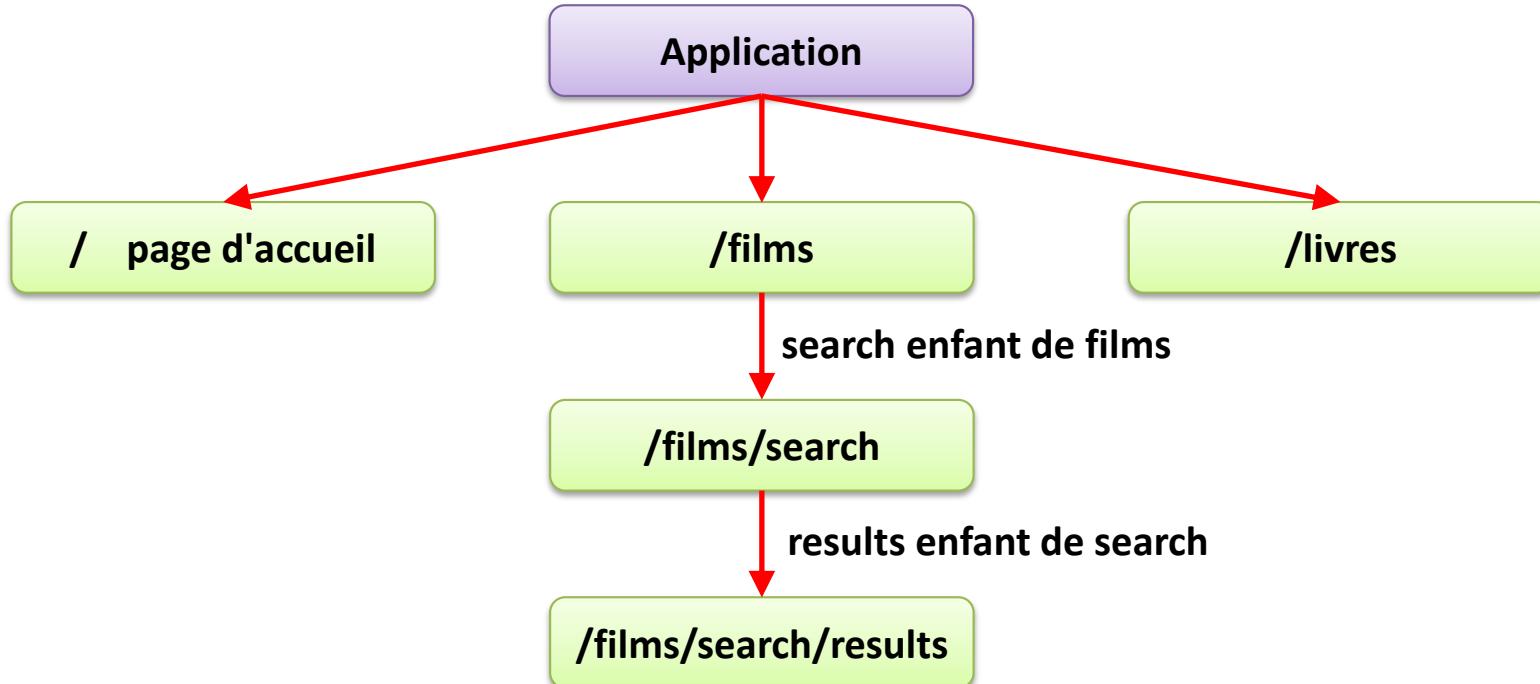
Désigne toutes les routes et doit donc être placé en dernier.  
S'il n'est pas à la fin, les routes qui suivront ne seront pas atteintes.

Redirige vers l'URL 404

# Les routes : Les routes imbriquées

Les routes imbriquées :

Arborescence :



# Les routes : Les routes imbriquées



## Les routes imbriquées :

- Permettent de garder les éléments des templates des pages ascendantes

url : site.com/films

url : site.com/films/search

url : site.com/films/search/results

En-tête du site

Template de :  
**FilmsComponent**

pied de page

En-tête du site

Template de :  
**FilmsComponent**

Template de :  
**FilmsSearchComponent**

pied de page

En-tête du site

Template de :  
**FilmsComponent**

Template de :  
**FilmsSearchComponent**

Template de :  
**FilmsResultsComponent**

pied de page

# Les routes : Les routes imbriquées

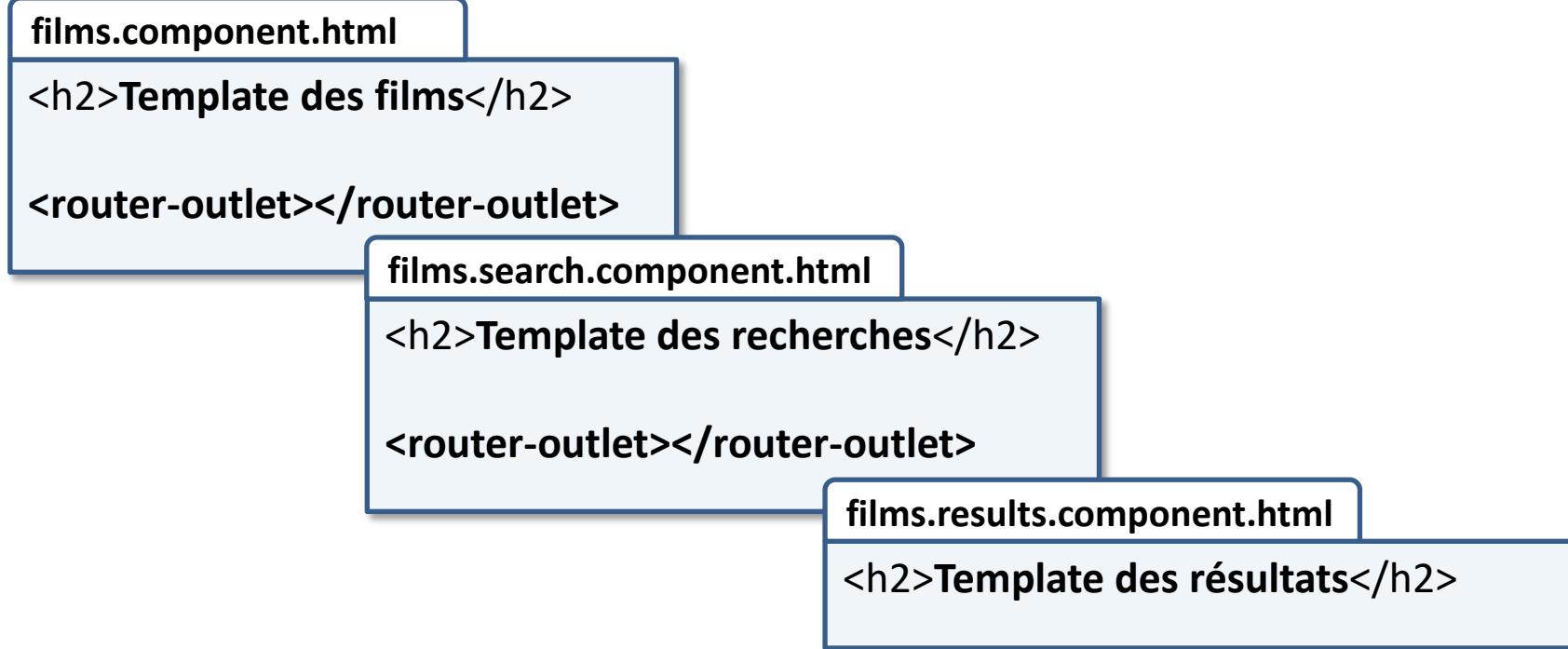
## Configuration des routes imbriquées :

app.routes.ts : contient la configuration des routes

```
export const ROUTES: Routes = [
  { path: "",      component: HomeComponent },
  {
    path: 'films',
    component: FilmsComponent,
    children: [
      {
        path: 'search',
        component: FilmsSearchComponent,
        children: [
          {
            path: 'results',
            component: FilmsResultsComponent
          }
        ]
      }
    ]
  }
];
```

# Les routes : Les routes imbriquées

## Templates des composantes imbriqués



# Les tests



# Les tests



## Deux types de tests:

- Les tests unitaires
  - vérifient qu'une portion du code fonctionne fournit le résultat escompté
- Les tests end-to-end (e2e)
  - Émule une interaction utilisateur avec l'application (clique, remplissage des champs, etc.)

# Les tests unitaires



## Les tests unitaires :

- Très rapide à exécuter
- Efficace pour tester l'intégralité du code
- Généralement en isolation (sans les dépendances de l'élément testé)
- Les dépendances de l'élément cible sont remplacés par des mocks
- C'est **Karma** qui a été choisi pour les exécuter sous Angular

## Mocks :

- Objets factices utilisés pour les tests unitaires
- Permet de remplacer et de simulés des fonctionnalités dont dépendent la cible du test

## Jasmine :

- Framework Behavior-driven development (BDD) pour les tests

## BDD :

- Méthode de test base sur des scénarios (Et si... Dans le cas...)
- Utilisation d'outils pour reprendre les scénarios et appliquer les comportements
- Utilisation de mocks pour les fonctionnalités manquantes pour le test

**fixture** : Permet la configuration de l'environnement du test

# Les tests unitaires

Jasmine :

- **describe()** : Groupe de tests
- **it()** : Définit un test
- **expect()** : Indique le résultat attendu. Il est possible de le chaîner indéfiniment.

## Exemple de test sur une classe

```
class Foo {  
    constructor(public age: number, public prenom: string) {  
    }  
    isPlusVieux(age) {  
        return this.age > age;  
    }  
}  
  
describe("Tests pour la classe Foo", () => {  
    it("Construction de la classe", () => {  
        const foo = new Foo(35, "Adrien");  
        expect(foo.age).toBe(35);  
        expect(foo.prenom).not.toBe("");  
        expect(foo.isPlusVieux(8)).toBe(true);  
    });  
});
```

# Les tests unitaires

Les tests devront se trouver dans un fichier tel que : nomdelaclass.spec.ts

**Initialise l'environnement avant de lancer les tests.  
Évite de répéter le même code pour chaque test**

**Test 1**

**Test 2**

**foo.spec.ts**

```
describe("Tests sur la classe Foo", () => {
    let foo: Foo;

    beforeEach( () => {
        foo = new Foo(35, "adrien");
    });

    it("Doit avoir un nom", () => {
        expect(foo.name). .not.toBe("");
    });

    it("L'âge ne doit pas être négatif", () => {
        expect(foo. age). toBeGreaterThan(0);
    });
});
```

Il existe **afterEach** qui s'exécute après les tests. Peut servir à refermer un flux.



## Éléments de tests

Doublures : Élément de test qui va remplacer la classe originale de l'application.

Dummy (fantôme) : Objet ou fonction qui ne fait que remplacer un autre pour les tests. Ne fait rien.

Fake (substitut) : Implémentation simplifiée d'un comportement attendu et ne fera appel à aucune dépendance. On peut l'utiliser comme prototype de l'élément final.

Stub (bouchon) : Fake le plus simple possible. Contient des valeurs prédéfinies.

Mock (simulacre) : Objet généré par un outil de test à partir de la classe qu'elle doit remplacer. On peut le configurer pour modifier son comportement lors des tests.

Spy (espions) : Mock qui surveille des appels sur des éléments

# Les tests unitaires

Poser un spy avec Jasmine :

foo.spec.ts

```
describe("Tests sur la classe Foo", () => {
  let foo: Foo;

  beforeEach( () => {
    foo = new Foo(35, "adrien");
    spyOn(foo, "isPlusVieux").and.returnValue(true);
  });

  it("test methode isPlusVieux", () => {
    pony. isPlusVieux(20);
    expect(foo.isPlusVieux).toHaveBeenCalledWith();
    expect(foo.isPlusVieux).toHaveBeenCalledWith(20);
  });
});
```

Espion



La méthode a été appelé ?



La méthode a dû être appelé avec la valeur 20



# Les tests unitaires

Il est possible d'utiliser l'injection de dépendances d'Angular pour les tests :

## Exemple d'injection de dépendances

```
import { TestBed } from '@angular/core/testing';

describe("Test récupération d'un service", () => {
    beforeEach( () => TestBed.configureTestingModule({
        providers: [ClasseService]
    }));

    it('should return races when list() is called', () => {
        const service = TestBed.get(ClasseService);
        expect(service.methode()).toBe(2);
    });
});
```

Comme avec les modules en Angular, il faut donner la liste des dépendances possibles.

Récupération du service en utilisant l'injecteur de dépendances d'Angular

# Les tests unitaires

Tester un service asynchrone :

## Exemple d'injection de dépendances

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

describe("Test récupération d'un service", () => {
  beforeEach( () => TestBed.configureTestingModule({
    providers: [ClasseService]
  });

  it('should return races when list() is called', async(() => {
    service.list().then( races => {
      expect(races.length).toBe(2);
    });
  }));
});
});
```

Comme avec les modules en Angular, il faut donner la liste des dépendances possibles.

Récupération du service en utilisant l'injecteur de dépendances d'Angular

# Bibliographie

# Bibliographie

<https://angular.io/docs/js/latest/>

<https://nodejs.org/en/docs/>

## npm

<https://docs.npmjs.com/>

## Angular CLI

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

## zone.js

<https://blog.thoughttram.io/angular/2016/01/22/understanding-zones.html>

## @types/core-js

<http://definitelytyped.org/>

## TS déclaration

<https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html>

## TS Fichier de définitions

<https://typescript.codeplex.com/wikipage?title=Writing%20Definition%20%28.d.ts%29%20Files>

## Patron de conception Observateur :

[https://fr.wikipedia.org/wiki/Observateur\\_\(patron\\_de\\_conception\)](https://fr.wikipedia.org/wiki/Observateur_(patron_de_conception))

<http://www.dofactory.com/javascript/observer-design-pattern>

[https://en.wikibooks.org/wiki/Computer\\_Science\\_Design\\_Patterns/Observer](https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Observer)

# Bibliographie

## RxJS :

<https://github.com/Reactive-Extensions/RxJS> : code

<https://xgrommx.github.io/rx-book/index.html> : livre

## Cycle de vie :

<https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>

<https://embed.plnkr.co/?show=preview>

## Polymer :

<https://www.polymer-project.org/1.0/>

## X-TAG :

<http://x-tag.github.io/>

## ES6 et les navigateurs :

<https://kangax.github.io/compat-table/es6/>

## Karma :

<https://www.grafikart.fr/formations/javascript-unit-test/karma>

<http://javamind-fr.blogspot.fr/2015/01/karma-lexecuteur-de-tests-javascript.html>

## Un livre gratuit :

<https://www.gitbook.com/book/rangle-io/ngcourse2/details>

## Livre pas gratuit :

<https://www.packtpub.com/web-development/mastering-typescript>

<https://www.packtpub.com/web-development/learning-angular-2>