

# Angular

Web application platform  
Mobile and desktop  
By Google

VISEO

# 0 - Introduction & Modalités pratiques



VISEO

[www.viseo.com](http://www.viseo.com)

# Votre formateur

Nom

Expérience  
Front/Angular

Consultant  
Viseo

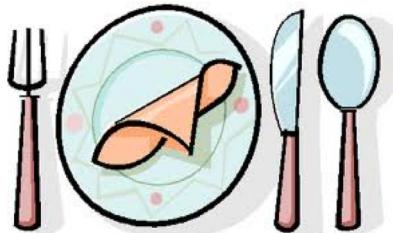
# Qui êtes vous ?

Votre nom ?

Vous et  
le RIA ?

Vous et cette  
formation ?

# Notre environnement



# Plan:

1. Introduction
2. ECMASCIPT 6
3. TypeScript
4. Les Web Components
5. La philosophie Angular
6. Application Angular
7. La syntaxe des templates
8. Composants et directives
9. Pipes
10. Injection de dépendances
11. Services
12. Routeur
13. Module http
14. Formulaires
15. Programmation réactive



# 1 - Introduction



# Introduction

- **Angular est conçu pour le web d'aujourd'hui :**
  - **avec ECMAScript 6**
  - **les Web Components**
  - **le mobile et le référencement en tête**
- **Il est activement maintenu par Google**

# 2 - ECMASCIPT 6



VISEO

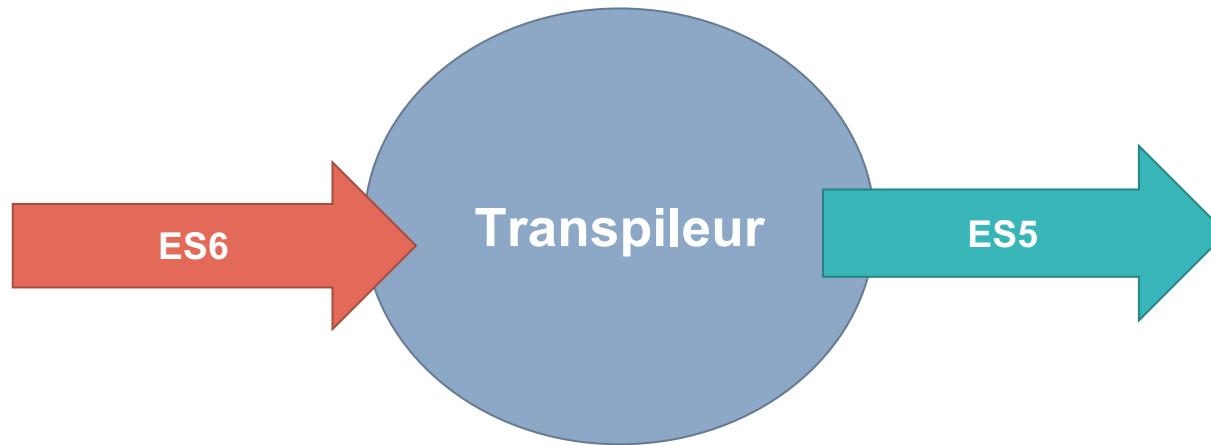
[www.viseo.com](http://www.viseo.com)

# ECMASCRIPT 6

- **ECMAScript, une spécification et JavaScript un implémentation**
- **ES6 modernise énormément JavaScript**
- **Les récents navigateurs supportent ES6 à plus de 90%**

# Transpileur

- Un **transpileur** prend du **code source ES6** en entrée et génère du **code ES5**, qui peut tourner dans n'importe quel navigateur



# Transpileur

- Il y a deux outils principaux pour transpiler de l'ES6:

- **Traceur**: un projet Google



- **Babeljs**: un projet démarré par Sebastian McKenzie, et qui a reçu beaucoup de contributions extérieures



# let

- ES6 introduit un nouveau mot-clé pour la déclaration de variable: **let**
- **let** a été pensé pour remplacer définitivement **var**
- **let** permet de déclarer des variables scopées

```
function getMovieTitle(movie) {  
  if (movie.genre) {  
    ➔ var title = movie.title + ':' + movie.genre;  
    return title;  
  }  
  console.log(title);  
  //title is still accessible here  
  return movie.title;  
}
```



```
function getMovieTitle(movie) {  
  if (movie.genre) {  
    ➔ let title = movie.title + ':' + movie.genre;  
    return title;  
  }  
  console.log(title);  
  //title is not accessible here  
  return movie.title;  
}
```

# Constantes

- ES6 introduit le mot-clé **const** pour déclarer des constantes

```
const MOVIES_PER_PAGE = 20;  
MOVIES_PER_PAGE = 15; // SyntaxError
```



```
const MOVIE = {};  
MOVIE.title = 'Superman'; // Works
```



```
const MOVIE = {};  
MOVIE = { title: 'Superman' }; // SyntaxError
```



# Constantes

- Même chose avec les tableaux

```
const MOVIES = [];
MOVIES.push({ title: 'Superman' }); // Works
```



```
const MOVIES = [];
MOVIES = [{ title: 'Superman' }, { title: 'Lucy' }]; // SyntaxError
```



# Création d'objets

- **Un nouveau raccourci pour créer des objets**

```
function createMovie() {  
    let title = 'Lucy';  
    let genre = 'Science Fiction';  
    ➔ return { title: title, genre: genre };  
}
```

- peut être simplifié en :

```
function createMovie() {  
    let title = 'Lucy';  
    let genre = 'Science Fiction';  
    ➔ return { title, genre };  
}
```

# Affectations déstructurées

- **Un raccourci pour affecter des variables à partir d'objets ou de tableaux.**

## ES5

```
var httpOptions = { timeout: 2000, isCache: true };
// later
var httpTimeout = httpOptions.timeout;
var httpCache = httpOptions.isCache;
```

## ES6

```
let httpOptions = { timeout: 2000, isCache: true };
// later
let { timeout: httpTimeout, isCache: httpCache } = httpOptions;
```

# Affectations déstructurées

- Si la variable qu'on veut affecter a le même nom que la propriété de l'objet à lire

```
let httpOptions = { timeout: 2000, isCache: true };
// later
let { timeout, isCache } = httpOptions;
// you now have a variable named 'timeout'
// and one named 'isCache' with correct values
```

ça marche aussi avec des objets imbriqués

```
let httpOptions = { timeout: 2000, cache: { age: 2 } };
// later
let { cache: { age } } = httpOptions;
// you now have a variable named 'age' with value 2
```

# Affectations déstructurées

- **la même chose est possible avec des tableaux :**

```
let timeouts = [1000, 2000, 3000];
// later
let [shortTimeout, mediumTimeout] = timeouts;
// you now have a variable named 'shortTimeout' with value 1000
// and a variable named 'mediumTimeout' with value 2000
```

- **ça fonctionne avec des tableaux de tableaux, des tableaux dans des objets, etc...**

# Valeurs par défaut

- ES6 offre une façon plus formelle de déclarer des paramètres optionnels

```
function getMovies(page) {  
    page = page || 1;  
    // ...  
    server.get(page);  
}
```

```
function getMovies(page = 1) {  
    // ...  
    server.get(page);  
}
```

- La valeur par défaut peut aussi être un appel de fonction

```
function getMovies(genre = defaultGenre(), page = 1) {  
    // the defaultGenre method will be called if genre is not provided  
    // ...  
    server.get(genre, page);  
}
```

# Valeurs par défaut

- Ce mécanisme s'applique aussi aux valeurs de variables:

```
let { timeout = 1000 } = httpOptions;  
// you now have a variable named 'timeout',  
// with the value of 'httpOptions.timeout' if it exists  
// or 1000 if not
```

# Rest operator

- **Une nouvelle syntaxe pour déclarer un nombre variable de paramètres dans une fonction**

```
function addMovies(...movies) {  
    let moviesList = [];  
  
    for (let movie in movies) {  
        moviesList.push(movie);  
    }  
  
    return moviesList;  
}  
  
addMovies('Superman', 'Lucy', 'Batman');
```

# Spread operator

- Le spread operator est l'opposé de Rest operator : il prend un itérable et le décompose en valeurs

```
function addMovies(...movies) {  
    let moviesList = [];  
    ↴  
    moviesList.push(...movies);  
  
    return moviesList;  
}  
  
addMovies('Superman', 'Lucy', 'Batman');
```

# Classes

- ES6 introduit les classes en JavaScript

```
class Movie {  
    constructor(title, genre) {  
        this.title = title;  
        this.genre = genre;  
    }  
  
    toString() {  
        return this.title + '(' + this.genre + ')';  
    }  
}  
  
let movie = new Movie('Titanic', 'Romance');  
console.log(movie.toString()); // Titanic (Romance)
```

# Classes

- **Une classe peut avoir des attributs et des méthodes statiques**

```
class Movie {  
    // ....  
    static defaultPopularity() {  
        return 10;  
    }  
}
```

- **Les méthodes statiques ne peuvent être appelées que sur la classe directement**

```
let popularity = Movie.defaultPopularity();
```

# Classes

- **Une classe peut avoir des accesseurs (getters, setters), si on veut implémenter du code sur ces opérations**

```
class Movie {  
    // ...  
    get genre() {  
        console.log('get genre');  
        return this._genre;  
    }  
  
    set genre(newGenre) {  
        console.log(`set genre ${ newGenre }`);  
        this._genre = newGenre;  
    }  
}  
  
let movie = new Movie();  
movie.genre = 'Drama'; // set genre Drama  
console.log(movie.genre) // get genre
```

# Classes

- L'héritage en ES6

```
class Record {  
    play() {  
        return 'playing';  
    }  
}  
  
class Audio extends Record {  
    constructor() {  
        super();  
    }  
}  
  
let audio = new Audio();  
console.log(audio.play()); // playing, as Audio inherits the parent method
```



# Classes

- Redéfinition des méthodes

```
class Record {
  play() {
    return 'playing';
  }
}

class Audio extends Record {
  play() {
    return super.play() + ' sound';
  }
}

class Video extends Audio {
  play() {
    return super.play() + ' and images';
  }
}

let audio = new Audio();
console.log(audio.play()); // playing sound

let video = new Video();
console.log(video.play()); // playing sound and images
```

# Arrow functions

- Nouvelle syntaxe utilisant l'opérateur fat arrow : =>

Exemple avec des *promises* en ES5

```
getUser(login)
  .then(function (user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function (rights) {
    updateMenu(rights);
  })
```

en ES6

```
getUser(login) ↴
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

# Arrow functions

- Les arrow functions gardent le **this** attaché lexicalement



## ES5

```
var maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(
      function (element) {
        if (element > this.max) {
          this.max = element;
        }
      });
  }
};

maxFinder.find([2, 3, 4]);
//Cannot read property 'max' of undefined
```

## ES6

```
let maxFinder = {
  max: 0,
  find: function (numbers) { ↗
    numbers.forEach(element => {
      if (element > this.max) {
        this.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
console.log(maxFinder.max);
// 4
```

# Set et Map

- On a maintenant de vraies collections en ES6
- Classe Map :

```
let user = { id: 1, name: 'VISEO' };

let users = new Map();

users.set(user.id, user); // adds a user
console.log(users.has(user.id)); // true
console.log(users.size); // 1
users.delete(user.id); // removes the user
```

# Set et Map

- **Classe Set**

```
let user = { id: 1, name: 'VISEO' };

let users = new Set();

users.add(user); // adds a user
console.log(users.has(user)); // true
console.log(users.size); // 1
users.delete(user); // removes the user
```

- **itérer sur une collection, avec la nouvelle syntaxe: `for ... of`**

```
for (let user of users) {
    console.log(user.name);
}
```

# Template de string

- Les templates de string fournissent un moteur de template basique avec support du multi-ligne (backticks `):

en ES5

```
let fullName = 'Mr' + firstName + ' ' + lastName;
```

en ES6

```
let fullName = `Mr ${firstName} ${lastName}`;
```

```
let template = `<div>
    <h1>Hello</h1>
</div>`;
```

# Modules

- **Une nouvelle syntaxe se charge de déclarer ce qu'on exporte depuis des modules, et ce qu'on importe dans d'autres modules**

```
//module "nom-module.js"
export function cube(x) {
    return x * x * x;
}
export const SQRT2 = Math.sqrt(2);
```

```
import { cube, SQRT2 } from "nom-module"
console.log(cube(3)); // 27
console.log(SQRT2);   // 1.4142135623730951
```

# 3 - TypeScript



# TypeScript

- **TypeScript est un sur-ensemble de JavaScript**
- **Il permet un typage statique des variables et des fonctions**
- **Il supporte la spécification ECMAScript 6**
- **Par convention les fichiers sources TypeScript ont l'extension .ts**

```
npm install -g typescript  
tsc test.ts
```

# Les types

- **Les types en TypeScript :**

```
let variable: type;
```

- **Exemples**

```
let movieName: string = 'The lord of the Rings';  
let moviePopularity: number = 7.5;
```

# Les types

- Une variable pouvant recevoir plusieurs types

```
let changing: any = 2;  
changing = true; // no problem
```

- Union des types

```
let changing: number|boolean = 2;  
changing = true; // no problem
```

# Les types

- Le type peut être aussi une classe

```
let movie: Movie = new Movie();
```

- TypeScript supporte les types génériques

```
let movies: Array<Movie> = [new Movie()];
```

- Cet Array ne peut contenir que des movies

```
movies.push('Lucy'); // error TS2345  
// Argument of type 'string' is not assignable to parameter of type 'Movie'.
```



# Les types

```
class ArrayList<T> {
    list: T[];

    constructor() {
        this.list = [];
    }

    public add(o: T): Boolean {
        this.list.push(o);
        return true;
    }

    public get(index: number): T {
        if (index < 0 || index >= this.list.length) { return null; }
        else{ return this.list[index]; }
    }

    public contains(o: T): Boolean {
        return this.list.indexOf(o) > -1;
    }
}

let movies: ArrayList<string> = new ArrayList<string>();
movies.add('Superman');
movies.add('Lucy');
movies.add('Interstellar');

console.log(movies.get(1)); // Lucy
console.log(movies.contains('Interstellar'));// true
```

# Return types

- On peut spécifier le type de retour d'une fonction

```
function bestMovie(movies Array<Movie>): Movie {  
  
    let best: Movie = movies[0];  
  
    for (let movie of movies) {  
        if (movie.popularity > best.popularity) {  
            best = movie;  
        }  
    }  
  
    return best;  
}
```

- Si la fonction ne retourne rien on la déclare avec le mot clé **void**

# Valeurs énumérées (enum)

- **TypeScript propose des valeurs énumérées**

```
enum MovieGenres { Action, Drama, Romance, ScienceFiction };

let movie: Movie = new Movie();

movie.genre = MovieGenres.ScienceFiction;
```

# Paramètre optionnel

- En JavaScript, les paramètres des fonctions sont optionnels
- Si on ne les passe pas, leur valeur sera **undefined**
- Mais en TypeScript ça déclenche une erreur 
- Pour montrer qu'un paramètre est optionnel on ajoute « ? »

```
function addPointsToScore(movie: Movie, points?: number): void {  
    points = points || 0;  
    movie.score += points;  
}
```

# Interfaces

- Si on veut obliger un paramètre passé à une fonction de posséder une propriété, on définit une interface

```
function addPointsToScore(movie: { score: number }, points: number): void {
    movie.score += points;
}
```

- On peut aussi nommer ces interfaces :

```
interface HasScore {
    score: number;
}

function addPointsToScore(movie: HasScore, points: number): void {
    movie.score += points;
}
```

# Classes

- Une classe peut implémenter une interface

```
interface CanBePlayed {  
    play(title: string): void;  
}  
  
class Movie implements CanBePlayed {  
    play(title: string): void {  
        console.log(title);  
    }  
}
```

# Classes

- On peut implémenter plusieurs interfaces

```
class Movie implements CanBePlayed, CanBeStopped {  
    play(title: string): void {  
        console.log(`playing ${ title }`);  
    }  
  
    stop(title: string): void {  
        console.log(`stopping ${ title }`);  
    }  
}
```

# Classes

- Une interface peut étendre une ou plusieurs autres interfaces

```
interface Record extends CanBePlayed, CanBeStopped {}

class Audio implements Record {
    //...
}
```

# Classes

- Une classe en TypeScript peut avoir des propriétés et des méthodes

```
class Movie {  
    title: string;  
  
    constructor(title: string) {  
        this.title = title;  
    }  
  
    play(): void {  
        console.log(`playing ${ this.title }`);  
    }  
}
```



Avoir des propriétés dans une classe n'est pas une fonctionnalité standard d'ES6, c'est seulement possible en TypeScript

# Classes

- Tout est public par défaut, mais on peut utiliser le mot-clé **private** pour protéger une propriété ou une méthode

```
class Movie {  
    public title: string;  
    ← private popularity: number;  
  
    constructor(title: string, popularity: number) {  
        this.title = title;  
        this.popularity = popularity;  
    }  
}  
  
let movie: Movie = new Movie('Superman', 8.5);  
console.log(movie.title); // Superman  
console.log(movie.popularity); // Error: '_popularity' is private;
```

# Decorateurs

## Exemple

```
let Log = function () {
    return (target: any, name: string, descriptor: any) => {
        console.log(`call to ${ name }`);
        return descriptor;
    };
};
```

```
class MovieService {
    @Log()
    getMovie(id: number) {
        //...
    }

    @Log()
    getMovies() {
        //...
    }
}
```

# Decorateurs

- Ils peuvent être appliquées sur:
  - une classe,
  - une propriété de classe,
  - une fonction
  - un paramètre de fonction
- **mais pas sur un constructeur**



# Configuration de TypeScript

- Utiliser **tsc** pour initialiser un projet

```
tsc --init
```

- Cela va créer un fichier, **tsconfig.json**, qui stockera les options de compilation TypeScript

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true,  
    "sourceMap": true,  
    "module": "commonjs",  
    "noImplicitAny": false,  
    "outDir": "dist",  
    "rootDir": "src"  
  }  
}
```

# Compilateur en mode Watch

- Lancer le compilateur TypeScript, en utilisant le watch mode

```
tsc --watch
```

- Cela devrait afficher quelque chose comme :

```
Compilation complete. Watching for file changes.
```

# 4 - Web Components



VISEO

[www.viseo.com](http://www.viseo.com)

# Web Components

- **Les composants web permettent de créer des balises HTML personnalisées et réutilisables**
- **Ça permet de combiner plusieurs éléments pour créer des composants d'interface graphique (widgets) réutilisables**
- **Web Component = composant réutilisable et encapsulé**

# Web Components

- Ils reposent sur un ensemble de standards:
  - Custom elements (éléments personnalisés)
  - Shadow DOM
  - Template
  - HTML imports

# Custom elements

- Un nouveau standard qui permet au développeur de créer ses propres éléments du DOM
- Déclarer un élément custom se fait avec `document.registerElement()`

```
// new element
var MovieCmp = document.registerElement('movie-cmp');
// insert in current body
document.body.appendChild(new MovieCmp());
```



La convention veut que le nom soit préfixé par un **identifiant**, suivi d'un **tiret**, pour indiquer que c'est un élément custom

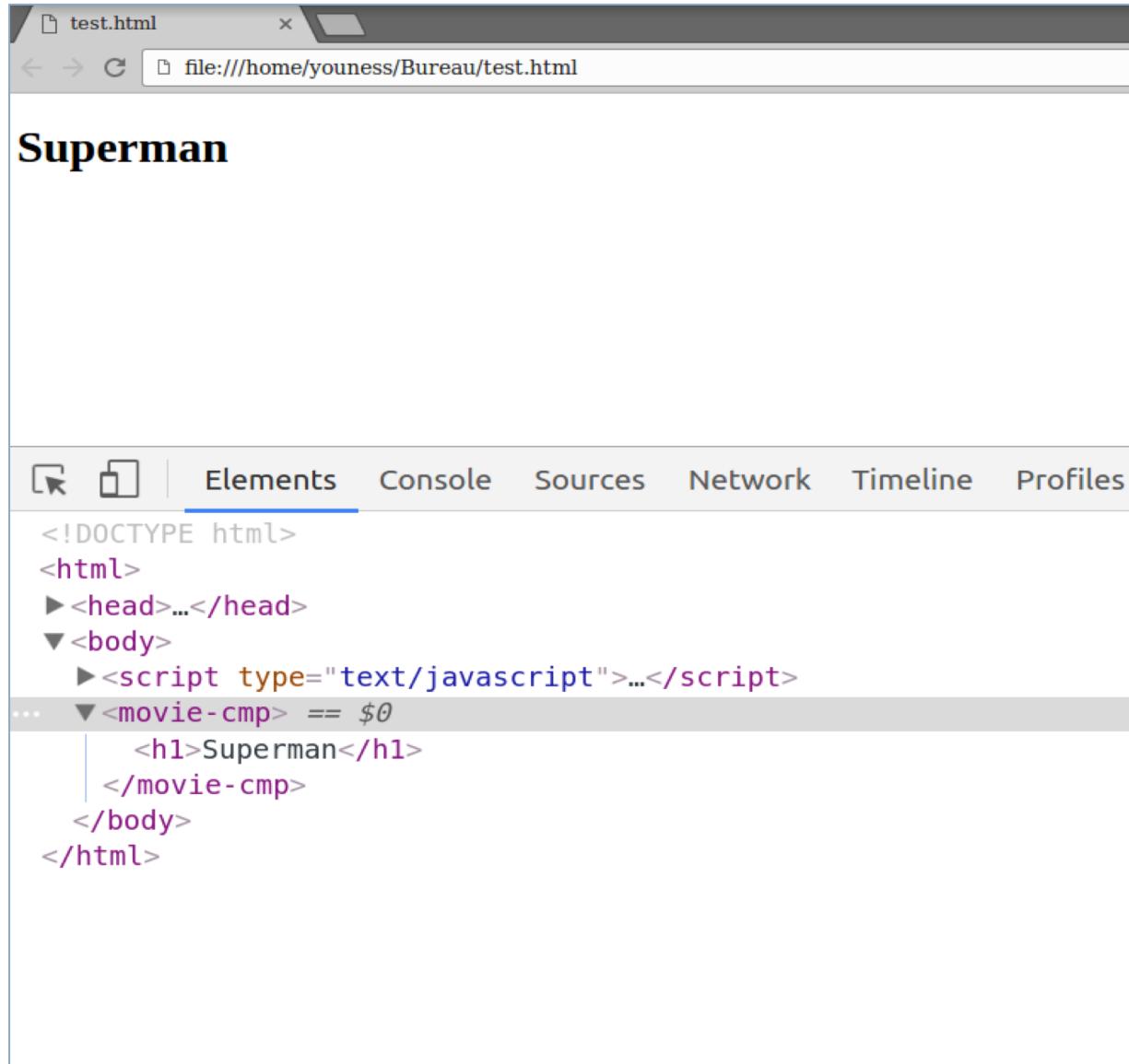
# Custom elements

- Un élément custom peut avoir :
  - des propriétés et des méthodes
  - des callbacks liés au cycle de vie
  - son propre template

```
// let's extend HTMLElement
var MovieCmpProto = Object.create(HTMLElement.prototype);
// and add some template using lifecycle
MovieCmpProto.createdCallback = function () {
    this.innerHTML = '<h1>Superman</h1>';
};

// new element
var MovieCmp = document.registerElement('movie-cmp', { prototype: MovieCmpProto });
// insert in current body
document.body.appendChild(new MovieCmp());
```

# Custom elements



# Custom elements

- Si on jette un coup d'œil au DOM, on verra  
`<movie-cmp><h1>Superman</h1></movie-cmp>`



Le CSS ou la logique JavaScript de l'application peut avoir des effets indésirables sur le composant

- En général, le template doit être encapsulé dans un Shadow DOM

# Shadow DOM

Si on retourne à notre exemple précédent :

```
var MovieCmpProto = Object.create(HTMLElement.prototype);
MovieCmpProto.createdCallback = function () {
  var shadow = this.createShadowRoot(); ↵
  shadow.innerHTML = '<h1>Superman</h1>';
};

var MovieCmp = document.registerElement('movie-cmp', { prototype: MovieCmpProto });
document.body.appendChild(new MovieCmp());
```

Si on essaie maintenant d'observer le DOM, on verra :



The screenshot shows a browser's DOM inspector. The tree structure is as follows:

- <!DOCTYPE html>
- <html>
- ▶ <head>...</head>
- ▼ <body>
- ▶ <script type="text/javascript">...</script>
- ▼ <movie-cmp>
- ...   ▼ #shadow-root (open) == \$0
- |    <h1>Superman</h1>
- |    </movie-cmp>
- </body>
- </html>

The shadow root is highlighted with a grey background. Inside the shadow root, there is a single <h1> element with the text "Superman".

# Shadow DOM

- Le Shadow DOM permet d'encapsuler le DOM du composant
- Ça signifie que le CSS et JavaScript de l'application ne vont pas s'appliquer sur le composant et le ruiner accidentellement
- Il permet de :
  - dissimuler le fonctionnement interne du composant
  - s'assurer que rien n'en fuit à l'extérieur

# Shadow DOM

- Même si on ajoutera du style aux éléments **h1**, rien ne changera
- le Shadow DOM agit comme une barrière

# Template

```
<html>
<body>

<template id="movie-tpl"> ↗
  <style>
    h1 { color: red }
  </style>
  <h1>Superman</h1>
</template>

<script>
  var MovieCmpProto = Object.create(HTMLElement.prototype);

  // add some template using the template tag
  MovieCmpProto.createdCallback = function () {
    var template = document.querySelector('#movie-tpl'); ↗
    var clone = document.importNode(template.content, true);
    this.createShadowRoot().appendChild(clone);
  }

  document.registerElement('movie-cmp', { prototype: MovieCmpProto });
</script>

<movie-cmp></movie-cmp> ↗

</body>
</html>
```

# Template

- **Un template est spécifié dans un élément <template>**
- **Il n'est pas affiché par le navigateur**
- **Son but est d'être cloné dans un autre élément**
- **Ce qu'on déclarera à l'intérieur sera inerte:**
  - les scripts ne s'exécuteront pas
  - les images ne se chargeront pas, etc...
- **Son contenu peut être requêté avec la méthode classique  
`getElementById()`**



# HTML imports

- Les imports HTML permettent d'importer du HTML dans du HTML
- `<link rel="import" href="movie-cmp.html">`
- Le fichier, `movie-cmp.html`, contiendrait tout ce qui est requis : le template, les scripts, les styles, etc...
- Si on veut ensuite utiliser un composant, il suffit simplement d'utiliser un import HTML

# Polyfill

- **En programmation web, un polyfill est un ensemble de fonctions, le plus souvent écrites en Javascript ou en Flash, permettant de simuler sur un navigateur web ancien des fonctionnalités qui ne sont pas nativement disponibles.**
- **Par exemple, accéder à des fonctions HTML5 sur des navigateurs ne proposant pas ces fonctionnalités.**

# Polyfill web component

- **Les Web Components ne sont pas complètement supportés par tous les navigateurs**
- **Il y a un polyfill à inclure pour être sûr que ça fonctionne**
- **Ce polyfill est appelé [web-component.js](#)**
- **Il est le fruit d'un effort commun entre Google, Mozilla et Microsoft, entre autres**

# Polymer et X-tag

- Au dessus de ce polyfill, quelques bibliothèques ont vu le jour
- Elles permettent de faciliter le travail avec les Web Components
- Elles viennent souvent avec un lot de composants tout prêts.
- Parmi les initiatives notables, on peut citer :
  - Polymer de Google ;
  - X-tag de Mozilla et Microsoft.

# Polymer et X-tag

## Exemple de composants Polymer existant

An example of using `<google-castable-video>`:

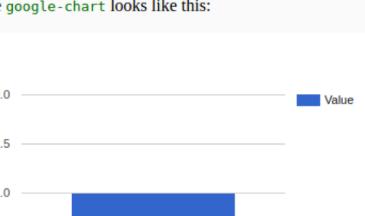


google-castable-video 1.0.2

HTML5 Video Element with extended Chromecast functionality.

Docs Demo Source Add to Collection Bower Command bower install --save Google

A simple `google-chart` looks like this:



google-chart 1.1.0

Encapsulates Google Charts into a web component

Docs Demo

Charts can be resized with CSS, but you'll need to call the `redraw` method when the size changes.

Here's a basic responsive example using only CSS and JS (You could also use `<iron-media-query>`):

Responsive chart



Category	Value
Col1	5.0
Col2	5.0
Col3	5.0

URL /bower\_components/app-route/demo/index.html#search/

Search Youtube

Route prefix: /search · Route path: / - Query params:

app-route 0.9.2

App routing expressed as Polymer Custom Elements.

Docs Demo Demo Source Add to Collection Bower Command bower install --save Polymer

Bundled Elements

app-route app-location app-route-converter

Golden boy Calum Scott hits the right note... Steven Spielberg vs Alfred Hitchcock. Epi...

VISEO

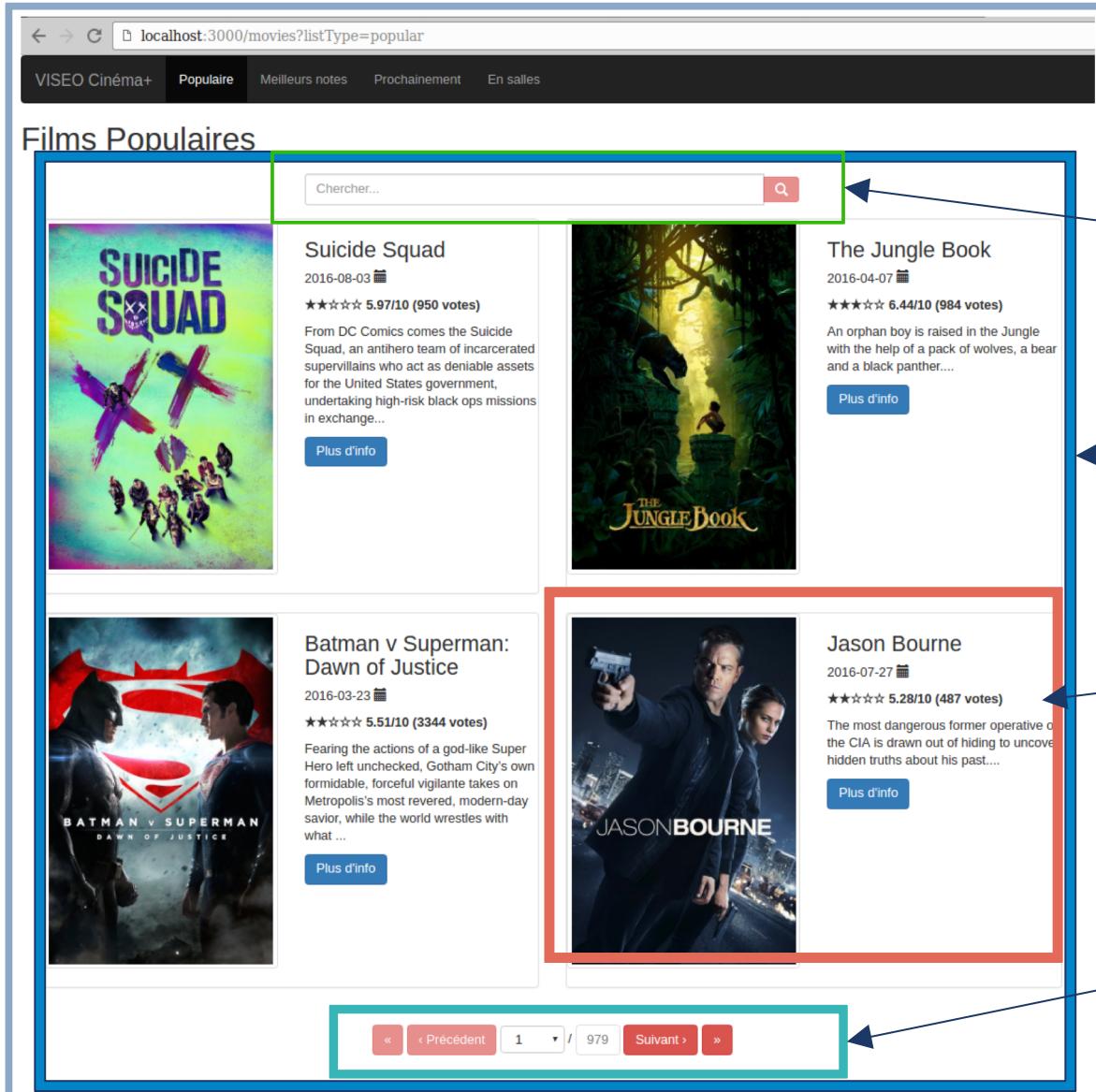
# 5- La philosophie d'Angular



# La philosophie d'Angular

- Angular est un framework orienté composant
- Des composants assemblés constituent une application
- Un composant est un groupe d'éléments HTML, dans un template, avec une logique métier derrière
- Les composants seront organisés de façon hiérarchique, comme le DOM
- Un composant racine aura des composants enfants, qui auront chacun des composants enfants, etc.

# La philosophie d'Angular



MoviesApp

SearchCmp

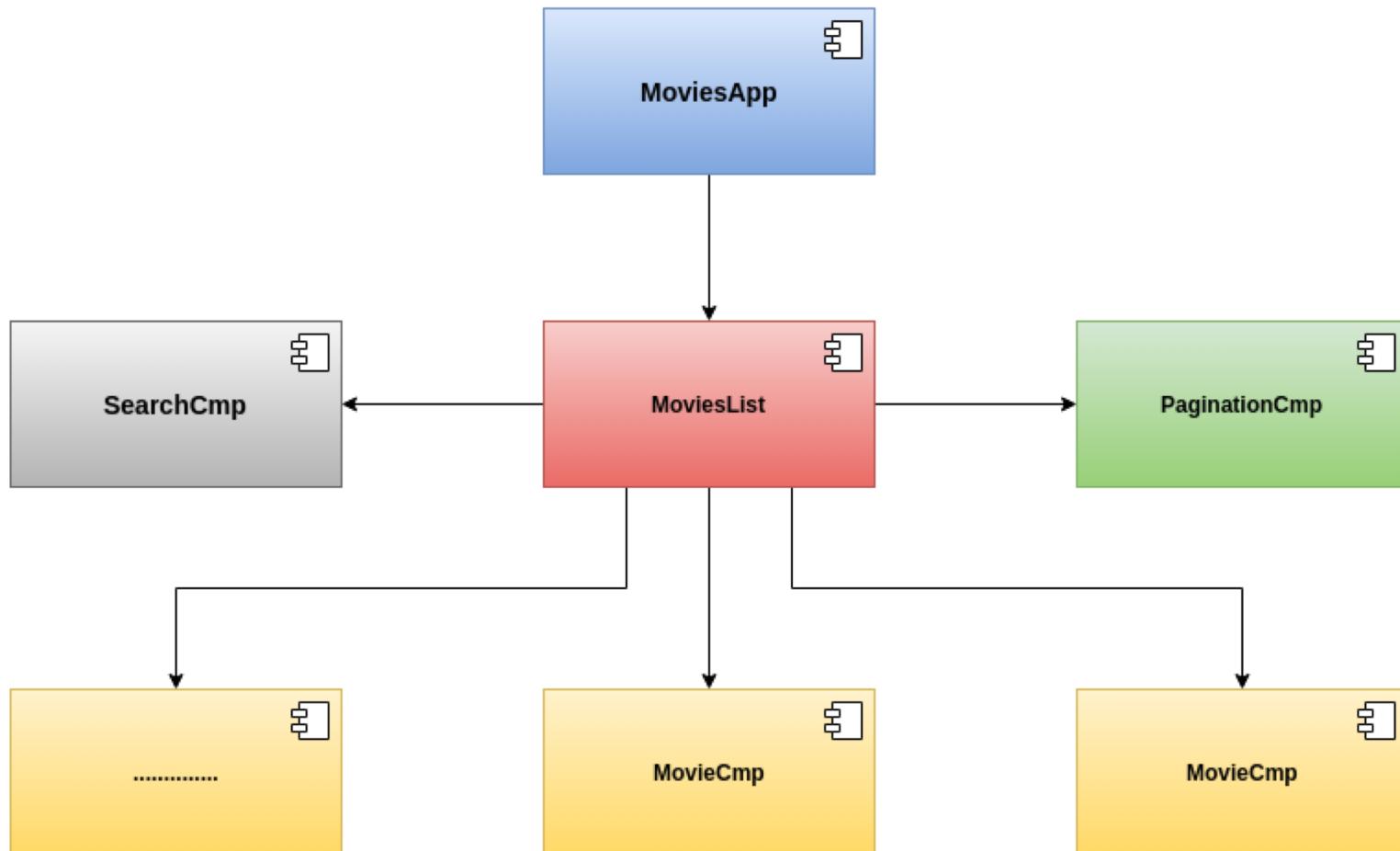
MoviesList

MovieCmp

PaginationCmp

# La philosophie d'Angular

Exemple : une application de cinéma « Popular movies »



# 6- Créeer une application Angular



# Outilage

- installer Node.js et NPM



- installer TypeScript: `npm install -g typescript`



# Configuration de TypeScript

- **Créer un nouveau répertoire**
- **Utiliser tsc depuis ce répertoire pour y initialiser un projet**

```
tsc --init
```

- **Cela va créer un fichier, tsconfig.json, qui stockera les options de compilation TypeScript**

```
{
  "compilerOptions": {
    "target": "es5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "sourceMap": true,
    "module": "commonjs",
    "noImplicitAny": false,
    "outDir": "dist",
    "rootDir": "src"
  },
  "exclude": [
    "node_modules"
  ]
}
```

# Compilateur en mode Watch

- Lancer le compilateur TypeScript, en utilisant le watch mode

```
tsc --watch
```

- Cela devrait afficher quelque chose comme :

```
Compilation complete. Watching for file changes.
```

- Laisser ce compilateur tourner en fond et ouvrir une nouvelle ligne de commande pour la suite

# Dépendances angular

- **Créer le fichier package.json**

```
npm init
```

- **package.json devrait contenir les dépendances suivantes**

```
"dependencies": {  
    "@angular/common": "2.0.0",  
    "@angular/compiler": "2.0.0",  
    "@angular/core": "2.0.0",  
    "@angular/platform-browser": "2.0.0",  
    "@angular/platform-browser-dynamic": "2.0.0",  
    "@angular/http": "2.0.0",  
    "@angular/forms": "2.0.0",  
    "@angular/router": "3.0.0",  
    "reflect-metadata": "^0.1.3",  
    "rxjs": "5.0.0-beta.12",  
    "zone.js": "^0.6.23"  
}
```

```
npm install
```

# Dépendances angular

- les différents packages @angular
- reflect-metadata, parce que nous utilisons les décorateurs
- rxjs, une bibliothèque pour la programmation réactive
- zone.js, pour faire tourner notre code dans des zones isolées et y détecter les changements

# typings

- **installer les typings**

```
npm install -g typings  
typings init  
typings install --save --global dt~core-js
```

- **Ajouter “node\_modules” dans la section exclude du fichier tsconfig.json**

```
"exclude": [  
  "node_modules"  
]
```

# Notre premier composant

- Créer un répertoire **src**
- Dans le répertoire **src**, créer un nouveau fichier, nommé **app.component.ts**
- Dès qu'on sauvera ce fichier, un fichier **app.component.js** apparaît dans le répertoire **dist** (compilateur **TypeScript**)
- Un composant est la combinaison d'une vue (**template**) et de logique (**Classe TS**).
- Créons une classe :

```
export class MoviesAppComponent {  
}
```

# Notre premier composant

- Notre application elle-même est un simple composant
- Pour indiquer à Angular que c'en est un, on utilise le décorateur **@Component**

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

@Component()
export class MoviesAppComponent {  
}
```

# Notre premier composant

- Le décorateur **@Component** attend un objet de configuration
- Une propriété est requise : **selector**

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

@Component({
  selector: 'movies-app'
})
export class MoviesAppComponent {
```

- Chaque fois que notre HTML contiendra un élément **<movies-app></movies-app>**, Angular créera une nouvelle instance de la classe **MoviesAppComponent**

# Notre premier composant

- Généralement un composant a un template

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

@Component({
  selector: 'movies-app',
  template: '<h1>Films populaires</h1>'
})
export class MoviesAppComponent {
```



On peut externaliser le template dans un autre fichier en utilisant la propriété **templateUrl** à la place de **template**

# Notre premier module Angular

- Une application angular aura toujours au moins un module
- Pour définir un module Angular nous devons créer une class
- Cette classe doit être décorée avec `@NgModule`
- Créer un fichier séparé, appelé `app.module.ts` pour le module racine

```
import { NgModule } from '@angular/core';
@ NgModule({})
export class AppModule {  
}
```

# Notre premier module Angular

- Puisque nous construisons une application pour le navigateur, le module racine devra importer **BrowserModule**

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports: [BrowserModule],
})
export class AppModule { }
```

# Notre premier module Angular

- Puisque nous construisons une application pour le navigateur, le module racine devra importer **BrowserModule**
- Nous devons également indiquer à Angular quel composant est le composant racine

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

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

# Bootstrapping

- Pour démarrer l'application, on utilise la méthode **bootstrapModule**
- Cette méthode est présente sur un objet retourné par une méthode appelée **platformBrowserDynamic**
- Il nous faut l'importer, depuis **@angular/platform-browser-dynamic**
- Créons un nouveau fichier **main.ts** pour séparer la logique de démarrage

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

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

# Bootstrapping

- Crée un autre fichier à la racine nommé **index.html**

```
<html>

  <head>
  </head>

  <body>
    <movies-app>
      loading...
    </movies-app>
  </body>

</html>
```

- Comment ajouter nos scripts dans notre fichiers HTML ?

# Bootstrapping

- Il y a cependant quelques problèmes



→ Les navigateurs ne supportent pas encore les modules ES6

- Pour charger nos modules, il nous faudra donc s'appuyer sur un outil : **SystemJS**
- **SystemJS est un petit chargeur de modules**

```
npm install --save systemjs
```

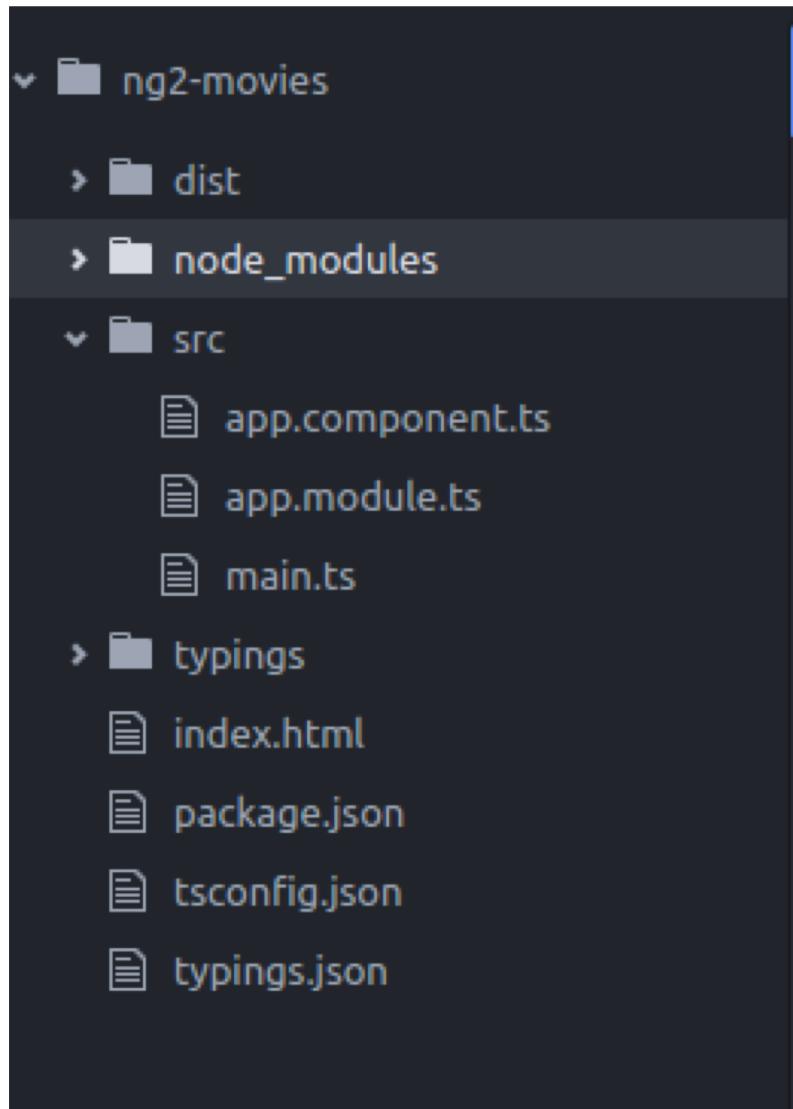
# Bootstrapping

- On doit charger **SystemJS** statiquement
- Lui indiquer où se situe notre module contenant la logique de démarrage (**dist/main**)
- On doit aussi lui montrer où trouver les dépendances de notre application

# Bootstrapping

```
<html>
<head>
  <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>
    System.config({
      defaultJSExtensions: true, // we want to import modules without writing .js at the end
      map: { // the app will need the following dependencies
        '@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',
        '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
        '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
        '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
        'rxjs': 'node_modules/rxjs'
      }
    });
    System.import('./dist/main'); // and to finish, let's boot the app!
  </script>
</head>
<body>
  <movies-app>loading...</movies-app>
</body>
</html>
```

# Bootstrapping



# Bootstrapping

- Démarrons maintenant un serveur HTTP pour servir notre mini application
- On va utiliser **lite-server** comme serveur web

```
npm install -g lite-server
```

- Pour le démarrer, allez dans le répertoire, et entrez :

```
lite-server
```

# 7 - La syntaxe des templates



# Interpolation

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

@Component({
  selector: 'movies-app',
  template: `
    <h1>Films populaires</h1>
    <h2>Nombre de films: {{ numberOfMovies }}</h2>
  `
})
export class MoviesAppComponent {
  numberOfMovies: number = 20;
}
```



# Interpolation

- Quand Angular détecte un élément <movies-app>, il crée une instance de la classe MoviesAppComponent
- Cette instance sera le contexte d'évaluation des expressions dans le template.
- La classe MoviesAppComponent affecte 20 à la propriété **numberOfMovies**, donc '20' affiché à l'écran.



Quand la valeur de **numberOfMovies** sera modifiée dans l'objet, le template sera automatiquement mis à jour! (*change detection*)

# Interpolation

- On peut accéder à des propriétés dans un objet

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

@Component({
  selector: 'movies-app',
  template: `
    <h1>Films populaires</h1>
    <h2>Bienvenue {{ user.name }}</h2>
  `,
})
export class MoviesAppComponent {
  user: any = { name: 'VISEO' };
}
```

# Utiliser d'autres composants dans le template

- Créer un nouveau composant <movies-list></movies-list>

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

@Component({
  selector: 'movies-list',
  template: `<h2>Nombre de films: {{ movies.length }}</h2>`
})
export class MoviesListComponent {
  movies: Array<string> = [
    'Batman vs Superman',
    'Star Wars',
    'The Revenant',
    'Mad max: Fury Road'
  ]
}
```

# Utiliser d'autres composants dans le template

- Ajouter le composant dans le template du composant parent

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

@Component({
  selector: 'movies-app',
  //add the movies-list component
  template: `
    <h1>Films populaires</h1>
    <h2>Bienvenue {{ user.name }}</h2>
    <movies-list></movies-list> ←
  `,
})
export class MoviesAppComponent {
  user: any = { name: 'VISEO' };
}
```

# Utiliser d'autres composants dans le template

- Déclarer le nouveau composant au niveau du module

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

// do not forget to import the component
import { MoviesListComponent } from './components/movies-list.component';

@NgModule({
  imports: [BrowserModule],           ↓
  declarations: [MoviesAppComponent, MoviesListComponent],
  bootstrap: [MoviesAppComponent]
})
export class AppModule { }

}
```

# Binding de propriété

```
<element [property]="expression"></element>
```

```
<p>{{ user.name }}</p>
```

=

```
<p [textContent]="user.name"></p>
```

# Binding de propriété

- L'interpolation n'est qu'une simplification du concept de **property binding**
- En Angular, on peut écrire dans toutes les propriétés du DOM via des attributs spéciaux sur les éléments HTML entourés de `[]`

```
<element [property]="expression"></element>
```

# Propriétés du DOM

Nom	Description	Type
<u>className</u>	Définit ou obtient la classe de l'élément.	<u>String</u>
<u>clientWidth</u>	La largeur intérieure d'un élément.	<u>Number</u>
<u>firstChild</u>	Le premier nœud enfant direct d'un élément, ou null si l'élément n'a pas de nœuds enfants.	<u>Node</u>
<u>id</u>	Définit ou obtient l'identifiant (id) de l'élément courant.	<u>String</u>
<u>innerHTML</u>	Renvoie ou définit l'ensemble du balisage et du texte contenu au sein d'un élément donné.	<u>String</u>
<u>lastChild</u>	Le dernier nœud enfant direct d'un élément, ou null si cet élément n'a pas de nœuds enfants.	<u>Node</u>
<u>name</u>	Définit ou obtient l'attribut name d'un élément.	<u>String</u>
<u>offsetHeight</u>	La hauteur d'un élément tel qu'affiché.	<u>Number</u>
<u>offsetWidth</u>	La largeur d'un élément tel qu'affiché.	<u>Number</u>
<u>parentNode</u>	Le nœud parent d'un élément, ou null si le nœud n'est pas dans <u>undocument DOM</u> .	<u>Node</u>
<u>scrollHeight</u>	La hauteur de la zone défilable d'un élément.	<u>Number</u>
<u>scrollWidth</u>	La largeur de la zone défilable d'un élément.	<u>Number</u>
<u>style</u>	Un objet représentant les déclarations de l'attribut style de l'élément.	<u>CSSStyleDeclaration</u>
<u>tabIndex</u>	Définit ou obtient la position de l'élément dans l'ordre de tabulation.	<u>Number</u>
<u>textContent</u>	Définit ou obtient le contenu textuel d'un élément et de ses descendants.	<u>String</u>
.....	.....	

# Binding de propriété

- L'interpolation que nous utilisions pour afficher le nom de l'utilisateur :

```
<p>{{ user.name }}</p>
```

- Est une simplification de :

```
<p [textContent]="user.name"></p>
```

- La syntaxe à base de `[]` permet de modifier la propriété **textContent** du DOM

# Binding de propriété



## Sensibilité à la casse

- **textcontent ou TEXTCONTENT ne fonctionnent pas, il faut écrire `textContent`**
- **Un Web Component agit comme un élément natif, sans aucune différence:**

```
<movie-cmp [name]="movie.name"></movie-cmp>
```

# Binding de propriété

- Aucune directive spécifique à apprendre !
- Si on veut cacher un élément, on peut utiliser la propriété standard **hidden**

```
<div [hidden]="isHidden">Hidden or not</div>
```

- On peut aussi accéder à des propriétés comme l'attribut **color** de la propriété **style**

```
<p [style.color]="foreground">ma couleur est {{ foreground }}</p>
```

# Binding de propriété

- Une expression peut aussi contenir un appel de fonction :

```
<movie-cmp [name]="movie.getName()"></movie-cmp>
```

```
<movie-cmp name="{{ movie.getName() }}"></movie-cmp>
```

# Événements

- Réagir à un événement peut être fait comme suit :

```
<element (event)="instruction"></element>
```

- Exemple :

```
<button (click)="onButtonClick()">Click me!</button>
```

Un clic sur le bouton déclenchera un appel à la méthode **onButtonClick()** du composant

# Événements

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

@Component({
  selector: 'movies-list',
  template: `
    <h2>Nombre de films: {{ movies.length }}</h2>
    <button (click)="refreshMovies()">Actualiser</button>
  `
})
export class MoviesListComponent {
  movies: Array<string> = [
    'Batman vs Superman',
    'Star Wars',
    'The Revenant',
    'Mad max: Fury Road'
  ]

  refreshMovies() {
    this.movies.push('Forest Gump');
  }
}
```

# Événements

- En Angular, il n'y a plus de directives spécifiques, ça fonctionne avec les événements standards du DOM
- Et aussi avec tous les événements spécifiques déclenchés par un composants Angular

```
<element (event)="instruction"></element>
```

# Événements

Événement	Action pour le déclencher
<b>click</b>	Cliquer (appuyer puis relâcher) sur l'élément
<b>dblclick</b>	Double-cliquer sur l'élément
<b>mouseover</b>	Faire entrer le curseur sur l'élément
<b>mouseout</b>	Faire sortir le curseur de l'élément
<b>mousedown</b>	Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément
<b>mouseup</b>	Relâcher le bouton gauche de la souris sur l'élément
<b>mousemove</b>	Faire déplacer le curseur sur l'élément
<b>keydown</b>	Appuyer (sans relâcher) sur une touche de clavier sur l'élément
<b>keyup</b>	Relâcher une touche de clavier sur l'élément
<b>keypress</b>	Frapper (appuyer puis relâcher) une touche de clavier sur l'élément
<b>focus</b>	« Cibler » l'élément
<b>blur</b>	Annuler le « ciblage » de l'élément
<b>change</b>	Changer la valeur d'un élément spécifique aux formulaires (input,checkbox, etc.)
<b>select</b>	Sélectionner le contenu d'un champ de texte (input,textarea, etc.)

# Événements

- L'instruction peut être un appel de fonction, mais ça peut être aussi n'importe quelle instruction exécutable

```
<button (click)="firstName='VISEO'; lastName='Technologies'>  
    Click to change name to "VISEO Technologies"  
</button>
```

# Événements

- Différence entre

```
<component [property]="doSomething()"></component>
```

- et

```
<component (event)="doSomething()"></component>
```



# Événements

- Dans le premier cas de binding de propriété

```
<component [property]="doSomething()"></component>
```

- La valeur doSomething() est une expression
- Elle sera évaluée à chaque cycle de détection de changement pour déterminer si la propriété doit être modifiée

# Événements

- Dans le second cas de binding d'événement

```
<component (event)="doSomething()"></component>
```

- La valeur doSomething() est une instruction (statement)
- Elle ne sera évaluée que lorsque l'événement est déclenché

# Variables locales

```
<input type="text" #nameInput>  
  
<p>Name : {{ nameInput.value }}</p>  
  
<button (click)="nameInput.focus()">Click me to focus the input</button>
```



# Variables locales

- **Les variables locales sont des variables qu'on peut déclarer dans un template avec la notation #**

- **Si on veut afficher la valeur d'un input :**

```
<input type="text" #name>  
{{ name.value }}
```

- **Avec la notation #, on crée une variable locale name qui référence l'objet HTMLInputElement du DOM**
- **Cette variable locale peut être utilisée n'importe où dans le template**

# Variables locales

- **Un autre cas d'usage des variables locales est l'exécution d'une action sur un autre élément**

```
<input type="text" #name>  
  
<button (click)="name.focus()">Focus the input</button>
```

- **Ça peut aussi être utilisé avec un composant spécifique**

```
<google-youtube #player></google-youtube>  
  
<button (click)="player.play()">Play!</button>
```

# Directives de structure

- Dans Angular, une directive est assez proche d'un composant,  
**mais n'a pas de template**
- On les utilise pour ajouter un comportement à un élément
- Les directives structurelles fournies par Angular s'appuient sur  
l'élément **<template>**
- Les directives ont la capacité d'utiliser le contenu de **<template>**,  
pour l'afficher ou non, le répéter, etc...

```
<template>
  <div>Liste des
  films</div>
</template>
```

# ngIf

- Si on veut instancier le template seulement lorsqu'une condition est réalisée, on utilise la directive **ngIf**

```
<template [ngIf]="movies.length > 0">  
    <div><h2>Films</h2></div>  
</template>
```

- Version raccourcie :

```
<div *ngIf="movies.length"><h2>Films</h2></div>
```

 \* : pour montrer que c'est une instantiation de template

 On utilisera toujours cette version courte

# ngIf

- **Les directives fournies par le framework sont déjà pré-chargées**
- **Pas besoin d'importer et de déclarer nglf dans l'attribut directives du décorateur @Component**

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

@Component({
  selector: 'movies-list',
  template: `
    <button (click)="refreshMovies()">Actualiser</button>
    &gt; <div *ngIf="movies.length >0">
      <h2>Nombre de films: {{ movies.length }}</h2>
      <!-- Affichage des films -->
    </div>
  `
})
export class MoviesListComponent {
  movies: Array<string> = [];

  refreshMovies() {
    this.movies.push('Forest Gump');
  }
}
```

# ngFor

- **ngFor** permet d'instancier un template par élément d'une collection

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

@Component({
  selector: 'movies-list',
  template: `
    <button (click)="refreshMovies()">Actualiser</button>
    <ul *ngIf="movies.length >0">
      <li *ngFor="let movie of movies">{{ movie }}</li>
    </ul>
  `
})
export class MoviesListComponent {
  movies: Array<string> = [
    'Batman vs Superman',
    'Star Wars',
    'The Revenant',
    'Mad max: Fury Road'
  ]

  refreshMovies() {
    this.movies.push('Forest Gump');
  }
}
```

# ngFor

- C'est possible de déclarer une autre variable locale, associée à l'indice de l'élément courant dans la collection

```
<ul *ngIf="movies.length >0">  
  <li *ngFor="let movie of movies; let i=index">{{i}}-{{ movie }}</li>  
</ul>
```

- La variable locale **i** recevra l'indice de l'élément courant, commençant à zéro
- index est une variable exportée 

# ngFor

- Il y aussi d'autres variables exportées par ngFor qui peuvent être utiles :
  - even, un booléen qui sera vrai si l'élément a un index pair
  - odd, un booléen qui sera vrai si l'élément a un index impair
  - first, un booléen qui sera vrai si l'élément est le premier de la collection
  - last, un booléen qui sera vrai si l'élément est le dernier de la collection

# ngSwitch

- **ngSwitch permet de switcher entre plusieurs templates selon une condition**

```
<div [ngSwitch]="messageCount">

  <p *ngSwitchCase="0">You have no message</p>

  <p *ngSwitchCase="1">You have a message</p>

  <p *ngSwitchDefault>You have some messages</p>

</div>
```

# ngStyle

- Nous avons déjà vu que nous pouvions agir sur le style d'un élément en utilisant

```
<p [style.color]="foreground">ma couleur est {{ foreground }}</p>
```

- Si on veut changer plusieurs styles en même temps, on peut utiliser la directive **ngStyle** :

```
<div [ngStyle]="{{fontWeight: fontWeight, color: color}}>I've got style</div>
```



Notez que la directive attend un objet dont les clés sont les styles à définir

# ngClass

- La directive **ngClass** permet d'ajouter ou d'enlever dynamiquement des classes sur un élément.
- On peut soit définir une seule classe avec le binding de propriété

```
<div [ngStyle]="{fontWeight: fontWeight, color: color}">I've got style</div>
```

- Si on veut en définir plusieurs classes en même temps, on utilise **ngClass** :

```
<div [ngClass]="{ 'awesome-div': isAnAwesomeDiv(), 'colored-div': isAColoredDiv() }">I've got style</div>
```

# Syntaxe canonique

- Chacune des syntaxes indiquées a une version plus verbuse appelée la **syntaxe canonique (canonical syntax)**

binding de propriété

```
<movie-cmp [name]="movie.name"></movie-cmp>
```

Syntax canonique

```
<movie-cmp bind-name="movie.name"></movie-cmp>
```

# Syntaxe canonique

binding d'événement

```
<button (click)="onButtonClick()"></button>
```

Syntax canonique

```
<button on-click="onButtonClick()"></button>
```

# Syntaxe canonique

Variables locales

```
<input type="text" #name>  
<button (click)="name.focus()">Focus the input</button>
```

Syntax canonique

```
<input type="text" ref-name>  
<button on-click="name.focus()">Focus the input</button>
```

# Résumé

- **{}** pour l'interpolation
- **[]** pour le binding de propriété
- **()** pour le binding d'événement
- **#** pour la déclaration de variable
- **\*** pour les directives structurelles (**ngIf**, **ngFor**, **ngSwitch**)

# TP 1-1 : Initier notre QCM

- Javascript
- HTML 5 
- Angular

HTML 5 sélectionné

- **Créer un tableau de questionnaires**
- **Afficher ce tableau**
- **Si je fais un clique sur un questionnaire, il devient le QCM courant**
- **Afficher quel est le QCM courant**

# TP 1-1 : Initier notre QCM

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

@Component({
  selector: 'qcms-app',
  template: `
    <h1>QCMs disponibles</h1>

    <ul>

      <li *ngFor="let qcm of qcms" (click)="setCurrentQcm(qcm)"><h2>{{ qcm }}</h2></li>

    </ul>

    <h1 *ngIf="currentQcm">{{ currentQcm }} sélectionné</h1>
  `
})

export class QcmsAppComponent {
  qcms: Array<string> = [
    'JAVA',
    'PHP',
    'JavaScript',
    'HTML'
  ];

  currentQcm: string = null;

  setCurrentQcm(qcm: string): void {
    this.currentQcm = qcm;
  }
}
```

# TP 1-2 : Affiner notre QCM

- **Ajouter des questions à chaque QCM**
- **Ajouter des réponses à chaque question**
- **Lorsqu'un QCM est sélectionné, la première question apparaît sous la liste. Elle présente la liste des réponses possibles.**
- **Quand on sélectionne une réponse, l'application passe automatiquement à la question suivante.**
- **Quand on a atteint la dernière question, le résultat est affiché**

# TP 1-2 : Affiner notre QCM

- Javascript
- HTML 5
- Angular

Javascript sélectionné

1<sup>ère</sup> question : le mot clé this représente :

- L'objet fonction lui-même
- L'objet qui contient la fonction
- Un objet de contexte (window par défaut)

# TP 1-2 : Affiner notre QCM

- Javascript
- HTML 5
- Angular

Javascript sélectionné

Bravo, vous avez 12 réponses justes sur 15 !

# TP 1-3 : Affiner notre QCM

- **Quand on a répondu à un questionnaire, celui-ci apparaît dans la liste avec une autre couleur**
- **Faire un mode « replay » si l'on sélectionne un questionnaire déjà rempli.**
- **Les questions apparaissent les unes après les autres.**
- **La bonne réponse pour chaque question apparaît en vert**
- **Si la réponse de l'utilisateur n'était pas correcte, elle doit apparaître en rouge**
- **On clique sur n'importe quelle réponse pour passer à la question suivante**

# TP 1-3 : Affiner notre QCM

- Javascript
- HTML 5
- Angular

## Javascript rejoué

1<sup>ère</sup> question : le mot clé this représente :

- L'objet fonction lui-même
- L'objet qui contient la fonction
- Un objet de contexte (window par défaut)

# TP 1-4 : Affiner notre QCM

- Ajouter des boutons « précédent » et « suivant » sous chaque question
  - En mode « normal », ces boutons permettent de naviguer dans le questionnaire sans toucher aux réponses éventuellement apportées
  - Le bouton précédent n'apparaît pas sur la première question
  - Le bouton suivant n'apparaît pas sur la dernière question, à la place, apparaît un bouton « évaluer »
  - En mode « replay », il faudra utiliser ces boutons pour naviguer (le click sur une option n'est plus accepté)

# 8- Composants et directives



# Introduction

- **Jusque-là, on a vu des petits composants**
- **Dans la vraie vie, les composants de nos applications seront bien plus complexes:**
  - Comment leur fournir des données ?
  - Comment gérer leur cycle de vie ?
  - Quelles sont les bonnes pratiques pour construire ses composants ?
- **Et les directives : c'est quoi, pourquoi, comment ?**



# Directives

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

@Directive({
  selector: '[loggable]'
})
export class LoggableDirective {

  constructor() {
    console.log('Loggable directive is called!');
  }
}
```

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

@Component({
  selector: 'movies-app',
  template: `
    <h1 loggable>Films populaires</h1>
  `
})
export class MoviesAppComponent {
```

# Directives

- Une directive est semblable à un composant, sauf qu'elle n'a pas de template
- la classe **Component** hérite de la classe **Directive** dans le framework
- La directive sera annotée d'un décorateur **@Directive**
- Les directives attachent du comportement aux éléments du DOM
- On peut avoir plusieurs directives appliquées à un même élément
- Une directive doit avoir un sélecteur CSS, qui indique au framework où l'activer dans notre template

# Sélecteurs

- Les sélecteurs peuvent être de différents types :
  - un élément, comme c'est généralement le cas pour les composants : `footer`
  - une classe, mais c'est plutôt rare : `.alert`
  - un attribut, ce qui est le plus fréquent pour une directive : `[color]`
  - un attribut avec une valeur spécifique : `[color=red]`
  - une combinaison de ceux précédents : `footer[color=red]`

# Sélecteurs

- Ne nommez pas les sélecteurs avec un préfixe **bind-**, **on-**, **ref-** ou **let-**;
- ils ont une autre signification pour le parseur, car ils font partie de la syntaxe canonique des templates.



# Entrées

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

@Directive({
  selector: '[loggable]',
  inputs: ['text: logText']
})
export class LoggableDirective {

  set text(value) {
    console.log(value);
  }
}
```

```
<div loggable logText="Some text"></div>
/* our Directive will log "Some text" */
```

# Entrées

- Un composant peut fournir des données à ses enfants en utilisant le binding de propriété
- Pour ce faire, il faut définir toutes les propriétés qui acceptent du binding de données, grâce à l'attribut **inputs** du décorateur `@Directive`
- **inputs** accepte un tableau de chaînes de caractères, chacune sous la forme « **property: binding** » où:
  - **property** désigne la propriété de l'instance du composant
  - **binding** une propriété du DOM qui contiendra l'expression

# Entrées



Si la propriété du DOM et l'attribut portent le même nom, on peut écrire simplement **property** au lieu de **property: binding** :

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

@Directive({
  selector: '[loggable]',
  inputs: ['logText']
})
export class LoggableDirective {

  set logText(value) {
    console.log(value);
  }
}
```

```
<div loggable logText="Hello"></div>
/* our Directive will log "Hello" */
```

# Entrées

- Il y a une autre façon de déclarer une entrée dans la directive avec le décorateur `@Input`

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

@Directive({
  selector: '[loggable]'
})
export class LoggableDirective {

  ➔ @Input('logText')
  set text(value) {
    console.log(value);
  }
}
```

# Entrées

- Si la propriété et le binding ont le même nom :

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

@Directive({
  selector: '[loggable]',
})
export class LoggableDirective {

  @Input()
  set text(value) {
    console.log(value);
  }
}
```

# Entrées

- Les entrées servent à passer des données d'un élément supérieur à un élément inférieur

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

@Component({
  selector: 'movies-list',
  template: `
    <h2>Films populaires</h2>

    <movie-cmp *ngFor="let currentMovie of movies" [movie]="currentMovie"></movie-cmp>
  `
})

export class MoviesListComponent {
  movies: Array<any> = [
    { title: 'Batman vs Superman', score: 8 },
    { title: 'Star Wars', score: 9 },
    { title: 'The Revenant', score: 10 },
    { title: 'Mad max: Fury Road', score: 7 }
  ];
}
```



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

@Component({
  selector: 'movie-cmp',
  template: `
    <p>{{ movie.title }}</p>
  `
})

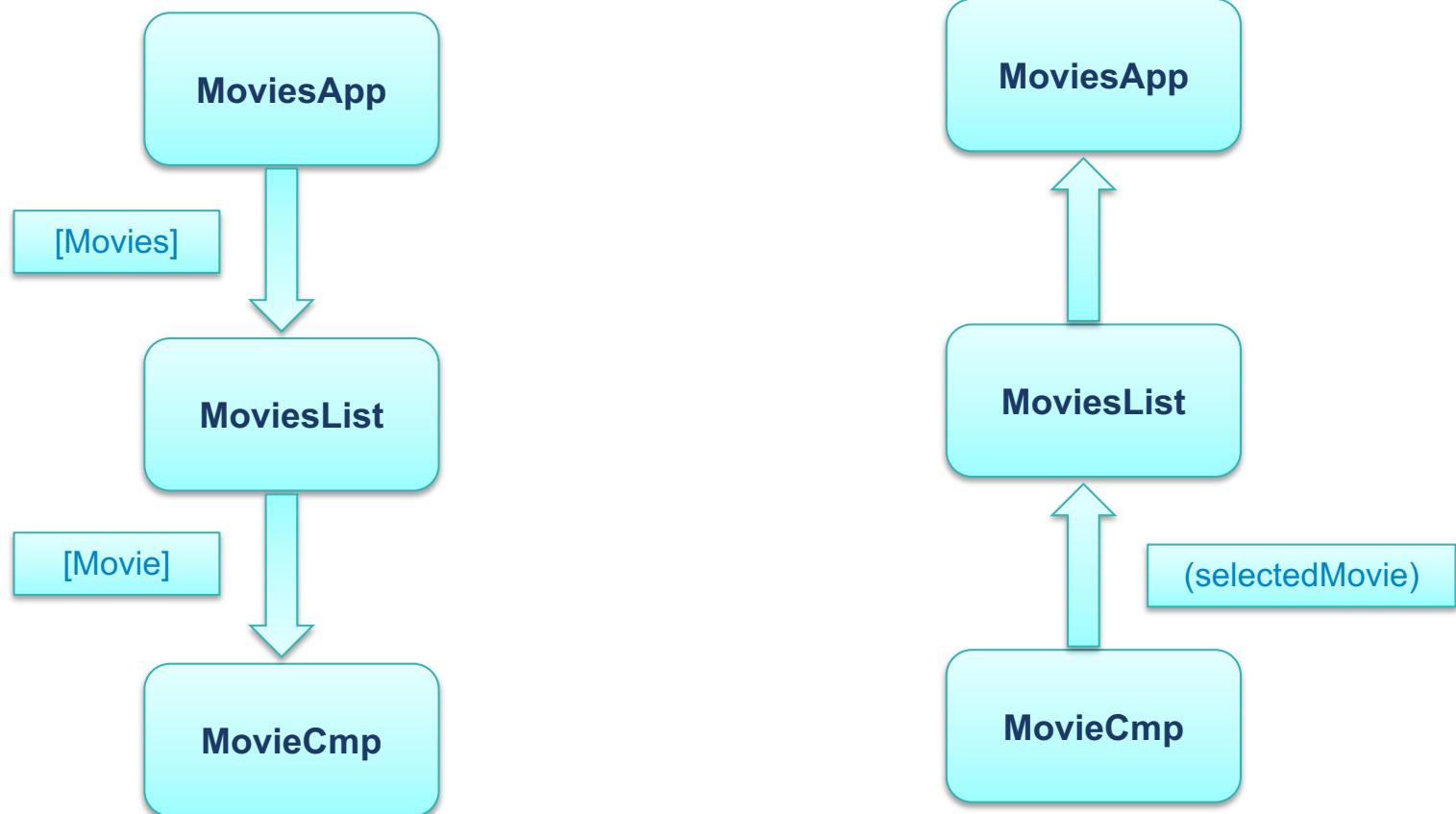
export class MovieComponent {
  @Input() movie: any;
}
```

- **Comment passe-t-on des données vers le haut ?**
- **On ne peut pas utiliser des propriétés pour passer des données de MovieComponent vers MovieListComponent**



# Sorties

- En Angular, les données entrent dans un composant via des propriétés, et en sortent via des événements



# Sorties

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

@Component({
  selector: 'movie-cmp',
  inputs: ['movie'],
  // we declare the custom event as an output
  outputs: ['movieSelected'],
  template: `
    <p>{{ movie.title }} <button (click)="selectMovie()">Sélectionner</button></p>
  `
})
export class MovieComponent {
  movie: any;

  // the EventEmitter is used to emit the event
  movieSelected: EventEmitter<Movie> = new EventEmitter<Movie>();

  /**
   * Selects a movie when the component is clicked.
   * Emits a custom event.
   */
  selectMovie() {
    this.movieSelected.emit(this.movie);
  }
}
```

# Sorties

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

@Component({
  selector: 'movies-list',
  template: `
    <h2>Films populaires</h2>
    <movie-cmp>
      *ngFor="let currentMovie of movies"
      [movie]="currentMovie"
      (movieSelected)="showMovieDetails($event)"
    </movie-cmp>
  `
})
export class MoviesListComponent {
  movies: Array<any> = [
    { title: 'Batman vs Superman', score: 8 },
    { title: 'Star Wars', score: 9 },
    { title: 'The Revenant', score: 10 },
    { title: 'Mad max: Fury Road', score: 7 }
  ];

  showMovieDetails(movie) {
    console.log(movie);
  }
}
```

# Sorties

- Les événements spécifiques sont émis grâce un **EventEmitter**
- Ils doivent être déclarés dans le décorateur, via l'attribut **outputs**
- **outputs** accepte un tableau contenant la liste des événements que la directive ou le composant peuvent déclencher
- Si on veut émettre un événement appelé **movieSelected**, il y aura trois étapes à réaliser :
  - déclarer la sortie dans le décorateur
  - créer un **EventEmitter**
  - et émettre un événement quand le movie est sélectionné

# Sorties

- **Comme pour les inputs, il est possible d'utiliser un décorateur pour déclarer un événement:**

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

@Component({
  selector: 'movies-cmp',
  template: `
    <p>{{ movie.title }} <button (click)="selectMovie()">Sélectionner</button></p>
  `
})
export class MovieComponent {
  @Input() movie: any;
  @Output() movieSelected: EventEmitter<Movie> = new EventEmitter<Movie>();

  selectMovie() {
    this.movieSelected.emit(this.movie);
  }
}
```

# Cycle de vie

- On peut avoir besoin que la directive réagisse à certains moments de sa vie



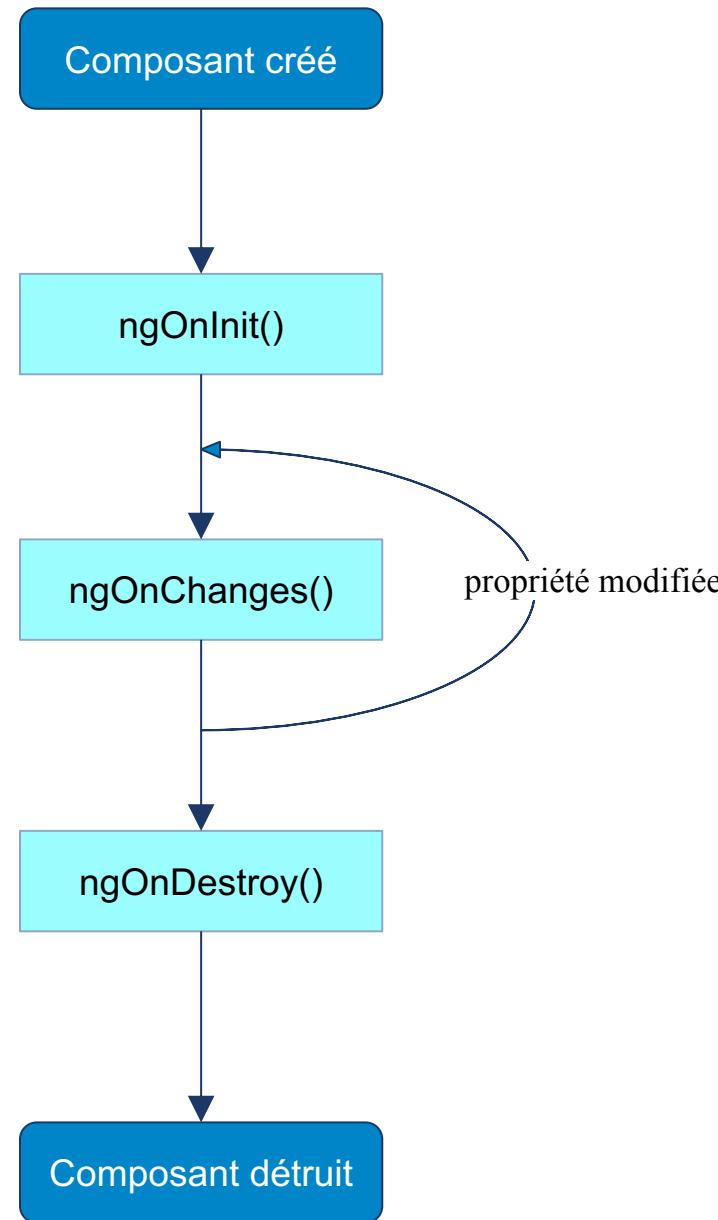
**les entrées d'un composant ne sont pas évaluées dans son constructeur**

```
@Directive({
  selector: '[loggable]'
})
export class LoggableDirective {

  @Input() msg: string;

  constructor() {
    console.log(`Message is ${this.msg}`);
    // will always log "Message is undefined"
  }
}
```

# Cycle de vie



# Cycle de vie

- Il y a plusieurs phases accessibles:
  - **ngOnInit** sera appelée une seule fois après le premier changement, c'est la phase parfaite pour du travail d'initialisation
  - **ngOnChanges** sera appelée quand la valeur d'une propriété bindée est modifiée (appelée à chaque changement)
  - **ngOnDestroy** est appelée quand le composant est supprimé. Utile pour y faire du nettoyage
- D'autres phases sont disponibles, mais pour des cas d'usage plus avancés : **ngDoCheck**, **ngAfterContentInit**, **ngAfterViewInit** ...

# Cycle de vie

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

@Directive({
  selector: '[loggable]'
})
export class LoggableDirective {

  @Input() msg: string;

  → ngOnInit() {
    console.log(`Message is ${this.msg}`);
    // msg is not undefined \o/
  }
}
```

- Maintenant on a accès aux entrées :)

# Cycle de vie

- La phase **ngOnDestroy** est parfaite pour nettoyer le composant

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

@Directive({
  selector: '[hello]'
})
export class HelloDirective {

  sayHello: number;

  constructor() {
    this.sayHello = window.setInterval(() => console.log('Hello'),
1000);
  }

  ➔ ngOnDestroy() {
    window.clearInterval(this.sayHello);
  }
}
```

# Composants

- **Un composant n'est pas très différent d'une directive**
- **Il a simplement des attributs supplémentaires, optionnels, et doit avoir une vue associée.**
- **Il n'apporte pas beaucoup d'attributs nouveaux comparés à la directive.**

# Composants (Template / URL de template)

- On peut soit déclarer le template en ligne, avec l'attribut **template**, ou utiliser une URL pour le placer dans un fichier séparé avec **templateURL**

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

@Component({
  selector: 'app-movie',
  templateUrl: 'path/to/movie.component.html'
})
export class MovieComponent {
  @Input() movie: any;
}
```

- On peut utiliser des chemins absous pour l' URL, ou un relatif, ou même une URL HTTP complète

# Composants (`styleUrls`)

- On peut aussi définir les styles du composant
- C'est utile si on compte faire des composants vraiment isolés
- On peut spécifier cela avec `styles` ou `styleUrls`

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

@Component({
  selector: 'app-movie',
  templateUrl: 'path/to/movie.component.html',
  styleUrls: ['path/to/movie.component.css']
})
export class MovieComponent {
  @Input() movie: any;
}
```

# Déclarations

- Il te faut ajouter dans les “declarations” de `@NgModule` chaque directive et composant qu’on utilise
- Si on ne le fais pas, le composant ne sera pas déclenché par le template, et on perdra beaucoup de temps à comprendre pourquoi



# TP 2-1 : Créer une administration qui permette de créer/supprimer un QCM

- Un bouton « édition » est ajouté à la page
  - Quand on le clique, tous les éléments jusque-là présents disparaissent
  - A la place apparaît une nouvelle liste de QCM
  - Un bouton « supprimer » apparaît à droite de chaque élément de la liste
- Un bouton « créer » apparaît sous la liste
- Chaque bouton supprimer, supprime le QCM associé
- Le bouton « créer »
  - fait apparaître sous la liste un formulaire ne contenant qu'un champ : le nom du QCM et un bouton valider.
  - La validation du QCM rajoute le QCM à la liste

# TP 2-1 : Créer une administration qui permette de créer/supprimer un QCM

Mode normal

- Javascript
- HTML 5
- Angular

Suppr

Suppr

Suppr

Créer

QCM : JQuery

Valider

# 9- *Pipes*



VISEO

[www.viseo.com](http://www.viseo.com)

# Pipes

- **Les pipes permettent de transformer les données**
- **Un pipe peut être utilisé dans le HTML, ou dans le code applicatif**
- **Angular propose un jeu de pipes standard**
- **On peut créer ses propres pipes**

# Pipes

```
<p>{{ 'Le Loup de Wall Street' | uppercase }}</p>  
<!-- will display 'LE LOUP DE WALL STREET' --&gt;</pre>
```

# Json

- **json** est un pipe pas utile en production, mais bien pratique pour le débug
- Ce pipe applique **JSON.stringify()** sur les données
- On peut l'utiliser dans n'importe quelle expression, au sein du HTML

```
<p>{{ movies | json }}</p>
```

- Et cela affichera la représentation JSON de l'objet

```
<p>[ { "name": "Superman" }, { "name": "Star Wars" }, { "name": "Forest Gump" } ]</p>
```

# Json

- Il est possible de chaîner plusieurs pipes:

```
<p>{{ movies | slice:0:2 | json }}</p>
```

- On peut en utiliser dans une expression interpolée, ou une expression de propriété

```
<p [textContent] = "movies | json"></p>
```

# Json

- On peut aussi l'utiliser dans le code :

```
import { Component } from '@angular/core';
// you need to import the pipe you want to use
import { JsonPipe } from '@angular/common';

@Component({
  selector: 'movies-list',
  template: `
    <div>{{ moviesAsJson }}</div>
  `
})
export class MoviesListComponent {
  movies: Array<any> = [
    { name: 'Batman vs Superman' },
    { name: 'Star Wars' },
    { name: 'The Revenant' }
  ];

  moviesAsJson: String;

  private _jsonPipe: JsonPipe

  constructor() {
    this._jsonPipe = new JsonPipe();
    this.moviesAsJson = this._jsonPipe.transform(this.movies);
  }
}
```



# Slice

- **slice** permet de n'afficher qu'un sous-ensemble d'une collection

```
<p>{{ movies | slice:0:2 }}</p>
/* will display the two first elements of movies */

<p>{{ 'VISEO Technologies' | slice:0:5 }}</p>
/* will display 'VISEO' */

<p>{{ 'VISEO Technologies' | slice:6 }}</p>
/* will display 'Technologies' */

<p>{{ 'VISEO Technologies' | slice:-12 }}</p>
/* will display 'Technologies' */
```

# Uppercase / Lowercase

```
<p>{{ 'viseo technologies' | uppercase }}</p>

/* will display 'VISEO TECHNOLOGIES' */
```

```
<p>{{ 'VISEO TECHNOLOGIES' | lowercase }}</p>

/* will display 'viseo technologies' */
```

# Number

```
<p>{{ 12345 }}</p>
<!-- will display '12345' -->

<p>{{ 12345 | number }}</p>
<!-- will display '12,345' -->

<p>{{ 12345 | number:'6.' }}</p>
<!-- will display '012,345' -->

<p>{{ 12345 | number:'2' }}</p>
<!-- will display '12,345.00' -->

<p>{{ 12345.13 | number:'1-1' }}</p>
<!-- will display '12,345.1' -->

<p>{{ 12345.16 | number:'1-1' }}</p>
<!-- will display '12,345.2' -->
```

{integerDigits}.{minFractionDigits}-{maxFractionDigits}

# Number

- **number** permet de formater un nombre
- Il prend un seul paramètre, une chaîne de caractères, sous la forme:  
**{integerDigits}.{minFractionDigits}-{maxFractionDigits}**
- où chaque partie est facultative
- Ces parties indiquent:
  - combien de chiffres veut-on dans la partie entière
  - combien de chiffres de précision minimale veut-on dans la partie décimale
  - combien de chiffres de précision maximale veut-on dans la partie décimale

# Percent

```
<p>{{ 0.8 | percent }}</p>
<!-- will display '80%' -->

<p>{{ 0.8 | percent:'.'3' }}</p>
<!-- will display '80.000%' -->
```

# Currency

```
<p>{{ 10.6 | currency:'EUR' }}</p>
```

*<!-- will display 'EUR10.6' -->*

```
<p>{{ 10.6 | currency:'USD':true }}</p>
```

*<!-- will display '\$10.6' -->*

```
<p>{{ 10.6 | currency:'USD':true:' .2' }}</p>
```

*<!-- will display '\$10.60' -->*

# Currency

- Ce pipe permet de formater une somme d'argent dans la devise qu'on veut
- Il faut lui fournir au moins un paramètre:
  - le code ISO de la devise ('EUR', 'USD'...)
  - Optionnellement, un flag booléen indiquant si l'on souhaite afficher le symbole ('€', '\$') ou le code ISO. Par défaut, le flag est à false
  - Optionnellement, une chaîne de formatage du montant, avec la même syntaxe que number

# Date

```
<p>{{ birthday | date:'dd/MM/yyyy' }}</p>
```

*<!-- will display '16/07/1986' -->*

```
<p>{{ birthday | date:'longDate' }}</p>
```

*<!-- will display 'July 16, 1986' -->*

```
<p>{{ birthday | date:'HH:mm' }}</p>
```

*<!-- will display '15:30' -->*

```
<p>{{ birthday | date:'shortTime' }}</p>
```

*<!-- will display '3:30 PM' -->*

# Exemple Custom Pipe : Rating pipe



Interstellar

2014-11-05

★★★★ 8.1/10 (5614 votes)

Interstellar chronicles the adventures of a group of explorers who make use of a newly discovered wormhole to surpass the limitations on human space travel and conquer the vast distances involved in a...

Plus d'info

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

@Pipe({ name: 'rating' })
export class RatingPipe implements PipeTransform {
  transform(value, args) {
    let stars: string = '';
    for (let i = 0; i < value; i++) {
      stars += '★';
    }

    return stars;
  }
}
```

# Créer ses propres pipes

- En Angular on peut créer nos propres pipes
- Custom Pipe = Une classe qui implémente l'interface **PipeTransform**, ce qui amène à écrire une méthode **transform()** qui fait tout le travail

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

export class CustomPipe implements PipeTransform {
    transform(value, args) {
        // do something here
    }
}
```

# Créer ses propres pipes

- Exemple : Utilisation de Rating pipe

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

@Component({
  selector: 'movie-cmp',
  template: `
    <h3>{{ movie.title }} {{ movie.score | rating }}</h3>
  `,
})
export class MoviesListComponent {
  movie: any = { title: 'Titanic', score: 5 };
}
```



# Créer ses propres pipes

- Déclarer le nouveau pipe au niveau du module

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

// do not forget to import the pipe
import { RatingPipe } from './pipes/rating.pipe'; ↴

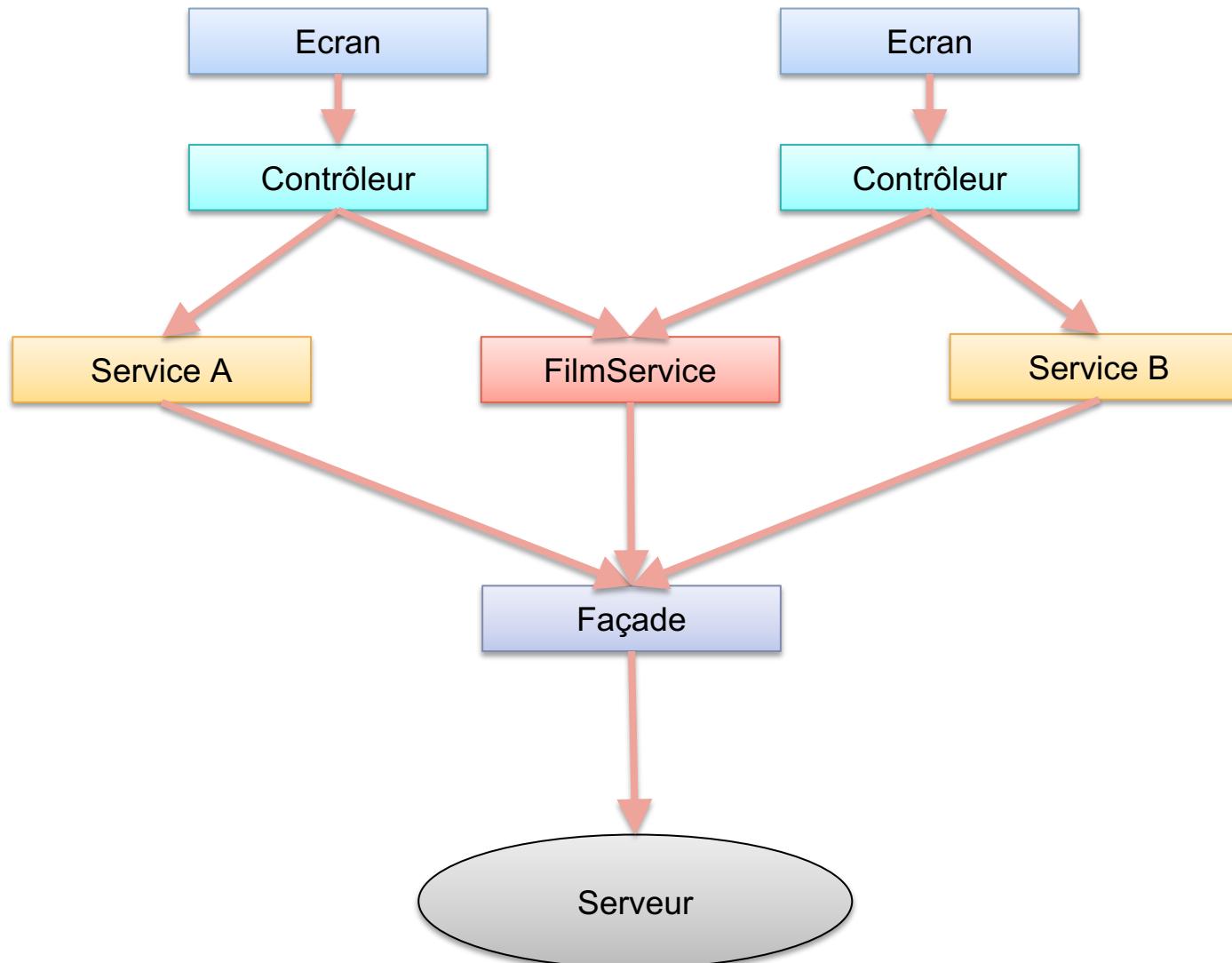
@NgModule({
  imports: [BrowserModule], ↓
  declarations: [MoviesAppComponent,...., RatingPipe],
  bootstrap: [MoviesAppComponent]
})
export class AppModule {

}
```

# 10 - Injection de dépendances



# Injection de dépendances



# Injection de dépendances

- L'injection de dépendances est un design pattern
- Un composant peut avoir besoin de fonctionnalités qui sont définies dans un service (**dépendance**)
- Au lieu de créer une instance du service au niveau composant, le framework crée l'instance pour nous et la fournit au composant (**inversion de contrôle**)

# Injection de dépendances

- Pour faire de l'injection de dépendances, on a besoin :
  - d'enregistrer une dépendance, pour la rendre disponible à l'injection dans d'autres composants/services.
  - de déclarer quelles dépendances sont requises dans chaque composant/service.

Une dépendance peut être un service fourni par Angular, ou un des services que nous avons écrits.



# Injection de dépendances

VISEO Cinéma+ Populaire Meilleurs notes Prochainement En salles

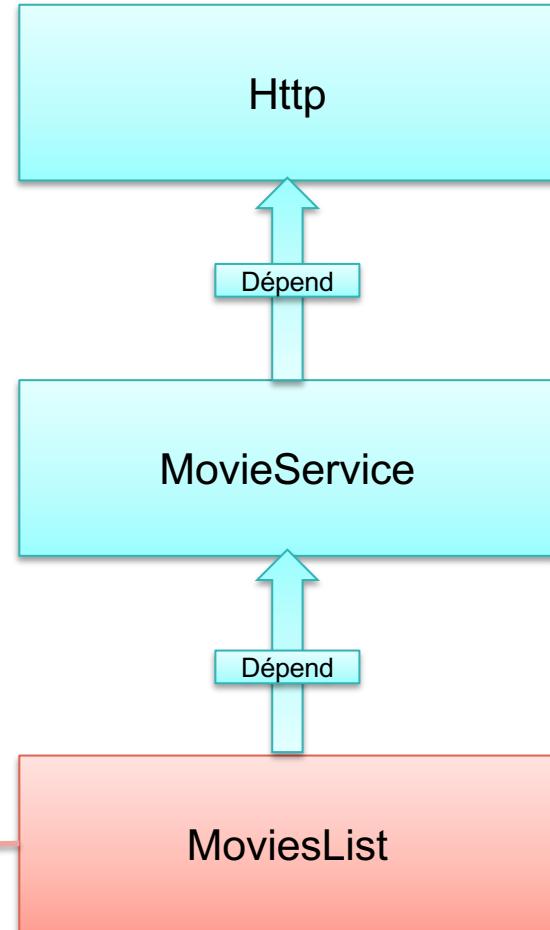
## Films Populaires

Captain America: Civil War  
2016-04-27 6.95/10 (756 votes)  
Following the events of Age of Ultron, the collective governments of the world pass an act designed to regulate all superhuman activity. This polarizes opinion amongst the Avengers, causing two factio...  
[Plus d'info](#)

Deadpool  
2016-02-09 7.23/10 (3089 votes)  
Based upon Marvel Comics' most unconventional anti-hero, DEADPOOL tells the origin story of former Special Forces operative turned mercenary Wade Wilson, who after being subjected to a rogue experimen...  
[Plus d'info](#)

Captain America: The Winter Soldier  
2014-03-20 7.62/10 (3181 votes)  
After the cataclysmic events in New York with The Avengers, Steve Rogers, aka Captain America is living quietly in Washington, D.C. and trying to adjust to the modern world. But when a S.H.I.E.L.D. co...  
[Plus d'info](#)

Captain America: The First Avenger  
2011-07-22 6.42/10 (4433 votes)  
Predominantly set during World War II, Steve Rogers is a sickly man from Brooklyn who's transformed into super-soldier Captain America to aid in the war effort. Rogers must stop the Red Skull - Adolf ...  
[Plus d'info](#)



# Injection de dépendances

- Avec TypeScript, pour déclarer une dépendance en utilisant le système de type

```
import { Http } from '@angular/http';

export class MovieService {

    baseUrl: string = 'url-to-server';

    constructor(private http: Http) { }

    getMovies(): any {
        return this.http.get(` ${this.baseUrl}/movies`);
    }

}
```

- Angular récupérera le service Http et l'injectera dans notre constructeur

# Injection de dépendances

- Pour indiquer à Angular qu'une classe est un service on ajoute un décorateur `@Injectable()`

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

→ @Injectable()
export class MovieService {

    baseUrl: string = 'url-to-server';

    constructor(private http: Http) { }

    getMovies(): any {
        return this.http.get(`${this.baseUrl}/movies`);
    }
}
```

# Injection de dépendances

- Enregister **MovieService** au niveau du module pour le rendre disponible à l'injection dans d'autres services et composants

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

import { MoviesAppComponent } from './app.component';
import { MoviesListComponent } from './components/movies-list.component';

→ import { MovieService } from './services/movie.service';

@NgModule({
  imports: [BrowserModule, HttpModule],
  declarations: [MoviesAppComponent, MoviesListComponent],
  → providers: [MovieService],
  bootstrap: [MoviesAppComponent]
})
export class AppModule {}
```

# Injection de dépendances

- On peut désormais utiliser notre nouveau service où on souhaite

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

import { MovieService } from '../../../../../services/movie.service'; ←

@Component({
  selector: 'movies-list',
  template: `
    <h2>Films populaires</h2>
    <movie-cmp *ngFor="let movie of movies" [movie]="movie"></movie-cmp>
  `
})
export class MoviesListComponent {
  movies: Array<any>;
  constructor(private movieService: MovieService) {} ↓

  ngOnInit() {
    this.movies = this.movieService.getMovies();
  }
}
```

# Injection de dépendances

- **Notre serveur n'existe pas**
  - Soit l'équipe backend n'est pas prête
  - soit on veut le faire plus tard
- **Dans tous les cas, on voudrait mocker tout ça**
- **C'est un intérêt majeur de l'injection de dépendances**

# Injection de dépendances (Configuration)

- **Un provider associe un token à un service**
- **L'injecteur retourne la même instance chaque fois que le même token est demandé (singleton design pattern)**
- **On peut définir un token différent de la classe de l'objet à injecter**

# Injection de dépendances (Configuration)

```
@NgModule({
    //...
    providers: [MovieService]
})
```

- C'est la version courte de :

```
@NgModule({
    //...
    providers: [{ provide: MovieService, useClass: MovieService }]
});
```

- On explique à l'injecteur qu'on veut un binding entre un **token (type)** et la **classe MovieService**



Si le token et la classe sont les mêmes, on utilise la version raccourcie

# Mocker un service

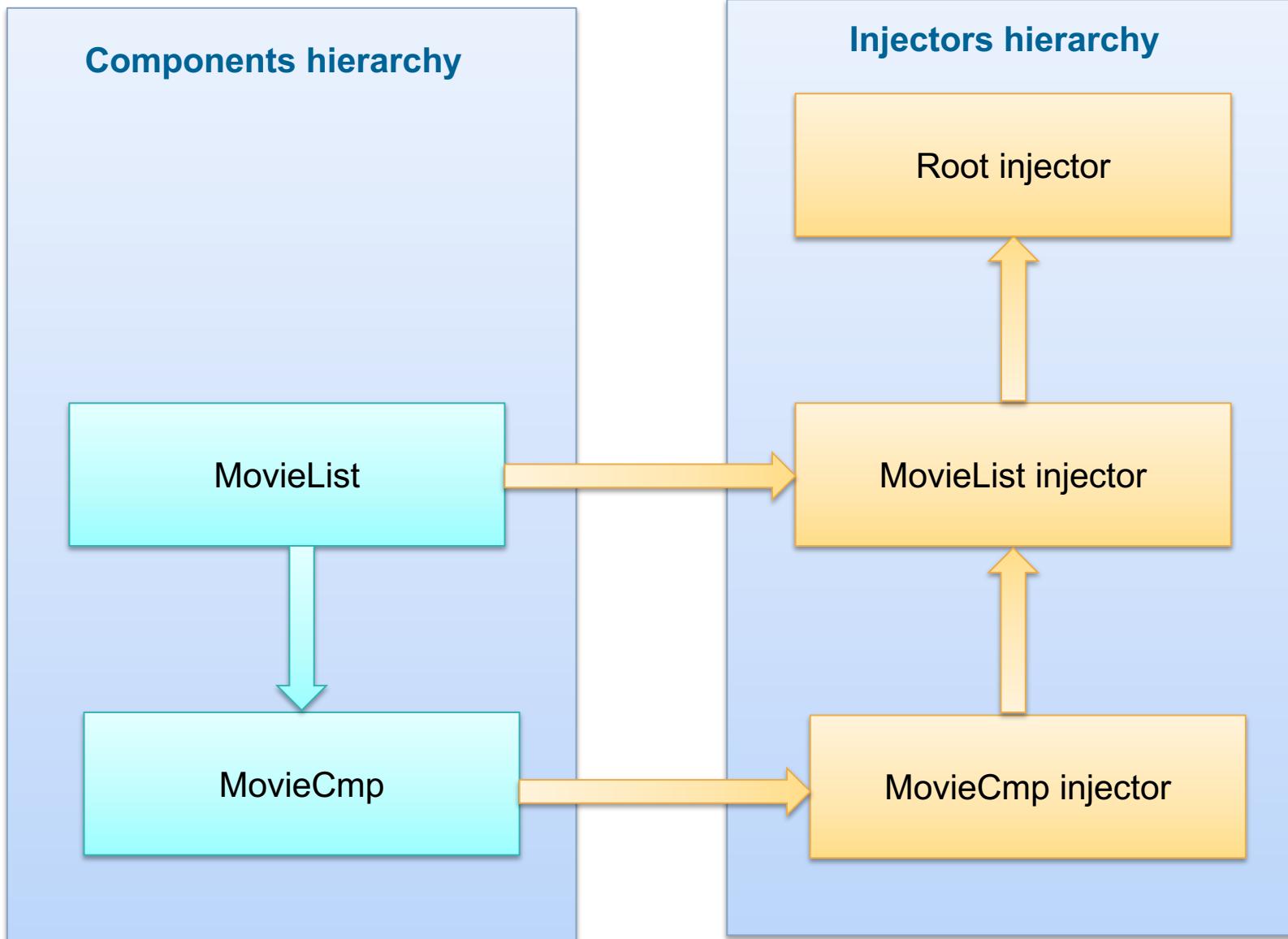
```
import { Injectable } from '@angular/core';

@Injectable()
export class MockMovieService {
  getMovies() {
    return [
      { title: 'Interstellar'},
      { title: 'Fight Club'},
      { title: 'Forrest Gump'},
      { title: 'Inception'}
    ];
  }
}
```

```
@NgModule({
  providers: [{ provide: MovieService, useClass: MockMovieService }]
});
```



# Injecteurs hiérarchiques



# Injecteurs hiérarchiques

- Il y a plusieurs injecteurs dans une application Angular
- Il y a un injecteur par composant, et chaque injecteur hérite de l'injecteur de son parent



**Les providers déclarés dans l'injecteur racine sont disponibles pour tous les composants de l'application**

# Injecteurs hiérarchiques

- On peut déclarer des dépendances à d'autres niveaux que l'injecteur racine
- Le décorateur `@Component` peut recevoir une autre option de configuration, appelée providers

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

import { MovieService } from '../../../../../services/movie.service';
import { MockMovieService } from '../../../../../services/mock-movie.service';

@Component({
  selector: 'movies-list',
  template: `
    <h2>Films populaires</h2>
    <movie-cmp *ngFor="let movie of movies" [movie]="movie"></movie-cmp>
  `,
  providers: [{ provide: MovieService, useClass: MockMovieService }]
})
export class MoviesListComponent {
  movies: Array<any>;

  constructor(private _movieService: MovieService) {}

  ngOnInit() {
    this.movies = this._movieService.getMovies();
  }
}
```

# Injecteurs hiérarchiques

- Le provider avec le token **MovieService** retournera toujours une instance de **MockMovieService**, quelque soit le provider défini dans l'injecteur racine
- C'est pratique si:
  - On veut utiliser une instance différente d'un service dans un composant donné
  - On tient à avoir des composants parfaitement encapsulés qui déclarent tout ce dont ils dépendent

# 11 - Services



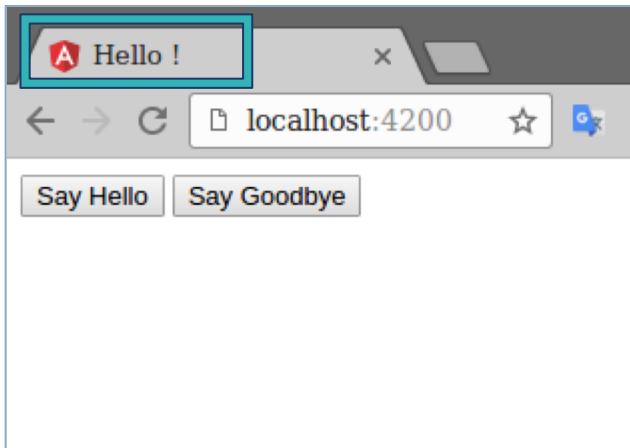
VISEO

[www.viseo.com](http://www.viseo.com)

# Services

- Angular propose le concept de services (des classes qu'on peut injecter dans une autre)
- Les services sont fournis par:
  - le cœur du framework (peu de services)
  - les modules communs
  - Et on peut en construire d'autres

# Service Title



```
import { Component } from '@angular/core';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'app-root',
  template: `
    <button (click)="sayHello()">Say Hello</button>
    <button (click)="sayGoodbye()">Say Goodbye</button>
  `,
  providers: [Title]
})
export class AppComponent {

  constructor(private title: Title) {}

  sayHello() {
    this.title.setTitle('Hello !');
  }

  sayGoodbye() {
    this.title.setTitle('Goodbye !');
  }
}
```

# Créer son propre service

- Pour créer un service, il suffit d'écrire une classe !

```
export class MovieService {  
    getMovies() {  
        return [  
            { title: 'Interstellar'},  
            { title: 'Fight Club'},  
            { title: 'Forrest Gump'},  
            { title: 'Inception'},  
            { title: 'Intouchables'}  
        ];  
    }  
}
```

- Un service est un singleton, donc la même instance unique sera injectée partout

# Créer son propre service

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


@Injectable()
export class MovieService {
  getMovies() {
    return [
      { title: 'Interstellar' },
      { title: 'Fight Club' },
      { title: 'Forrest Gump' },
      { title: 'Inception' },
      { title: 'Intouchables' }
    ];
  }
}
```

# TP 3 : Affiner notre QCM

- Remplacer la gestion actuelle de la liste des QCM par un service qui se charge de centraliser cette gestion
- Faire les ajustements nécessaires dans les composants
- Il ne doit pas être possible de modifier/supprimer un QCM qui est en cours d'utilisation

# 13 - Programmation réactive



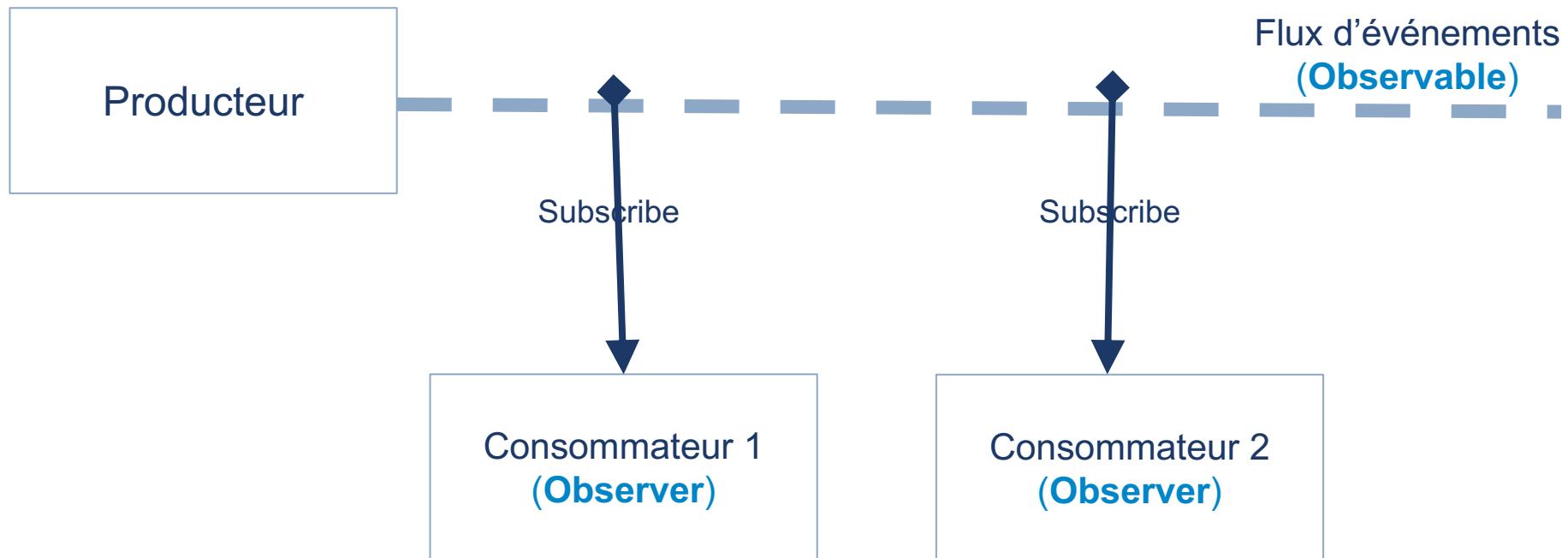
# Programmation réactive

- La programmation réactive est une façon de construire une application avec des événements, et d'y réagir (d'où le nom)
- Les événements peuvent être combinés, filtrés, groupés, etc... en utilisant des fonctions comme **map**, **filter**, etc...
- Dans la programmation réactive, toute donnée entrante sera dans un flux
- Ces flux peuvent être écoutés, modifiés (filtrés, fusionnés, ...) , et même devenir un nouveau flux que l'on pourra aussi écouter.

# Programmation réactive

- **Ça permet d'obtenir des programmes faiblement couplés**
- **On se contente de déclencher un événement, et toutes les parties de l'application intéressées réagiront en conséquence**

# Principes généraux



# Principes généraux

- Dans la programmation réactive, tout est un flux
  - Un flux est une séquence ordonnée d'événements
  - Ces événements représentent des **valeurs**, des **erreurs**, ou des **terminaisons**
  - Ces événements sont poussés par un producteur de données, vers un consommateur
  - Il faut s'abonner (**subscribe**) à ces flux, i.e. définir un **listener** capable de gérer ces trois possibilités.
  - Un tel listener sera appelé un **observer**, et le flux, un **observable**
-  Ça constituent un design pattern bien connu : l'**observer**

# Observable

- **Les observables sont très similaires à des tableaux**
- **Un observable est une collection de valeurs, comme un tableau**
- **Un observable ajoute juste la notion de valeur reportée dans le temps**
- **La bibliothèque la plus populaire de programmation réactive dans l'écosystème JavaScript est RxJS. Et c'est celle choisie par**

# Exemple Observable avec RxJS

```
import { Observable } from 'rxjs/Rx';

//.....

Observable.from([1, 2, 3, 4, 5])
  .map(x => x * 2)
  .filter(x => x > 5)
  .subscribe(x => console.log(x)); // 6, 8, 10
```

- **Tout observable, comme un tableau, peut être transformé avec des fonctions classiques :**

<b>take(n)</b>	pioche les n premiers éléments
<b>map(fn)</b>	applique la fonction fn sur chaque événement et retourner le résultat
<b>filter(predicate)</b>	laisse passer les seuls événements qui répondent positivement au prédicat
<b>reduce(fn)</b>	appliquera la fonction fn à chaque événement pour réduire le flux à une seule valeur unique
<b>merge(s1, s2)</b>	fusionne les deux flux
<b>subscribe(fn)</b>	applique la fonction fn à chaque événement qu'elle reçoit
...	.....

# RxJS

- La méthode `subscribe` accepte un deuxième callback, consacré à la gestion des erreurs

```
Observable.from([1, 3, 8, 5])
  .map(x => {
    if (x % 2 === 0) {
      throw new Error('something went wrong');
    } else {
      return x;
    }
  })
  .subscribe(
    x => console.log(x),
    error => console.log(error)
  );
}

// 1 3 something went wrong
```

# Programmation réactive en Angular

- Angular propose un adaptateur autour de l'objet Observable : **EventEmitter**
  - **EventEmitter** a une méthode **subscribe()** pour réagir aux événements, et cette méthode reçoit trois paramètres :
    - une méthode pour réagir aux événements
    - une méthode pour réagir aux erreurs
    - une méthode pour réagir à la terminaison
- Un **EventEmitter** peut émettre un événement avec la méthode **emit()**

# Programmation réactive en Angular

```
let emitter = new EventEmitter();

emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.emit('hello');
emitter.emit('there');
emitter.complete();
// logs "hello", "there", "done"
```

# Programmation réactive en Angular

```
let emitter = new EventEmitter();

let subscription = emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.emit('hello');
subscription.unsubscribe(); // unsubscribe
emitter.emit('there');
// logs "hello" only
```

# 14 - Envoyer et recevoir des données par HTTP



# HttpModule

- Angular fournit un module **HttpModule**
- Le module **HttpModule** propose un service nommé **Http** qu'on peut injecter dans n'importe quel constructeur
- Pour rendre **Http** disponible à nos composants et services il faut l'importer dans le module racine

# HttpModule

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

@NgModule({
  imports: [BrowserModule, HttpModule], ↓
  //...
})
export class AppModule { }
```

# Injecter le service Http

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

@Injectable()
export class MovieService {

  baseUrl: string = 'url-to-server';
  constructor(private http: Http) { }

  getMovies(): Observable<any> {
    return this.http.get(`${this.baseUrl}/movies`);
  }
}
```

# le service Http

- le service Http propose des méthodes correspondant au verbes HTTP communs :

get

post

put

delete

patch

head

- toutes ces méthodes retournent un objet Observable

# Obtenir la réponse

```
http.get(`${baseUrl}/api/movies`)
  .subscribe(response => {
    console.log(response.status); // logs 200
    console.log(response.headers); // logs []
    console.log(response.json()); // movies
  });
}
```

- **text()** si on s'attend à du texte;
- **json()** si on s'attend à un objet JSON

# Envoyer des données

```
// you need to stringify the object you send  
http.post(`${baseUrl}/api/movies` , newMovie)
```

# Transformer des données

```
import 'rxjs/add/operator/map';

import 'rxjs/add/operator/filter';

http.get(`${baseUrl}/api/movies`)

// extract json body

.map(res => res.json())

.subscribe(movies => {

    // store the array of the movies in the component

    this.movies = movies;

});
```

# Options avancées

- Chaque méthode Http accepte un objet **RequestOptions** en paramètre optionnel, où on peut configurer la requête

```
const searchParams = new URLSearchParams();
searchParams.set('sort', 'ascending');

const options = new RequestOptions({ search: searchParams });

http.get(` ${baseUrl}/api/movies`, options)
  .subscribe(response => {
    // will return the movies sorted
    this.movies = response.json();
  });
}
```

# Options avancées

- L'option **headers** permet d'ajouter quelques headers custom à la requête

```
const headers = new Headers();
headers.append('Authorization', `${token}`);

http.get(` ${baseUrl}/api/movies`, new RequestOptions({ headers }))
  .subscribe(response => {
    // will return the movies visible for the authenticated user
    this.movies = response.json();
});
```

# MovieService

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class MovieService {

  baseUrl: string = 'http://api.themoviedb.org/3';
  apiKey: string = '5192eb6331a3db50b6b388ae8941edc6';

  popularUrl: string = `${this.baseUrl}/movie/popular?api_key=${this.apiKey}`;
  topRatedUrl: string = `${this.baseUrl}/movie/top_rated?api_key=${this.apiKey}`;
  upcomingUrl: string = `${this.baseUrl}/movie/upcoming?api_key=${this.apiKey}`;
  searchUrl: string = `${this.baseUrl}/search/movie?api_key=${this.apiKey}`;

  constructor(private _http: Http) { }

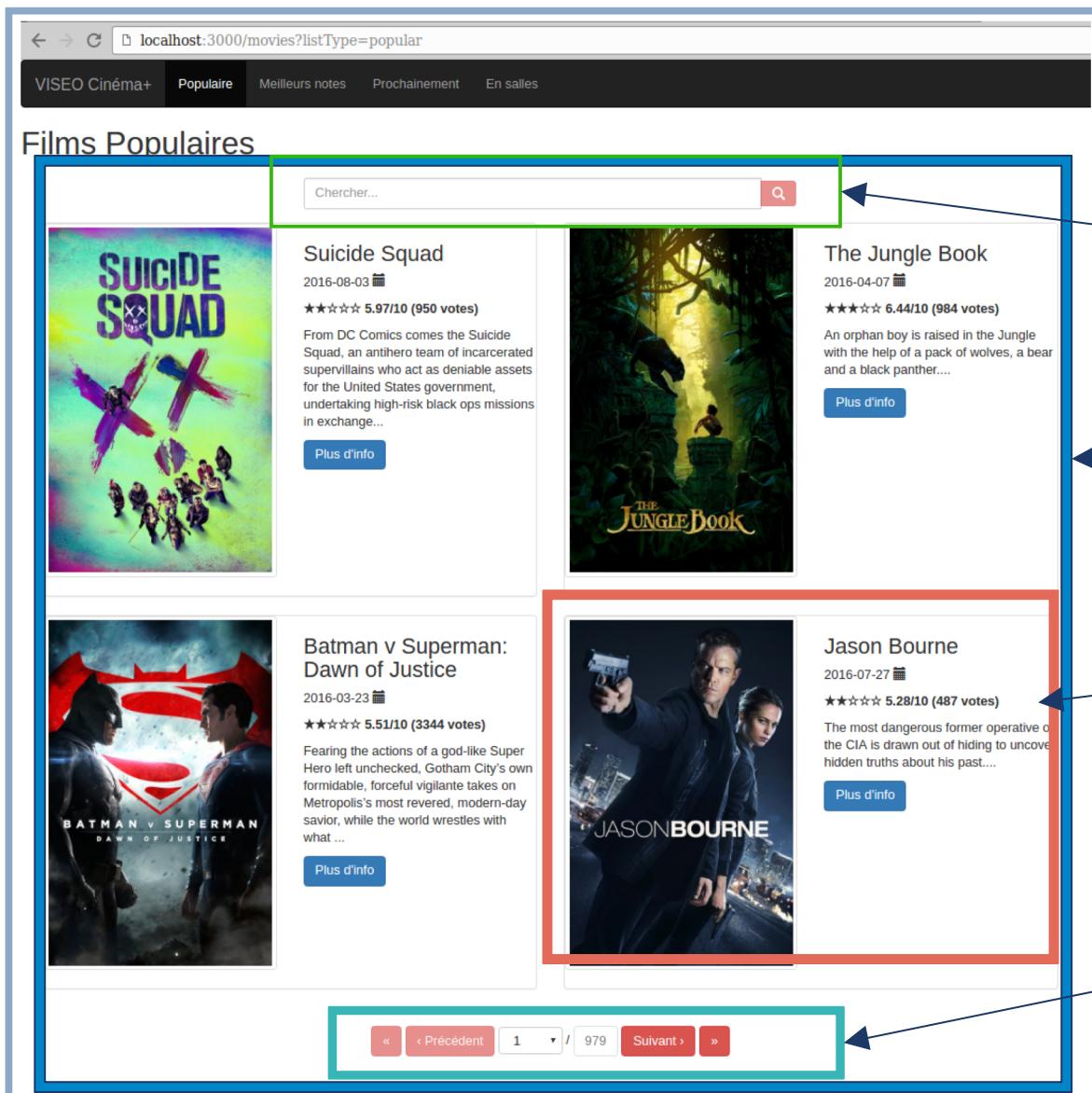
  getMovie(id: number): Observable<any> {
    return this._http.get(`${this.baseUrl}/movie/${id}?api_key=${this.apiKey}`);
  }

  getPopularMovies(page: number = 1): Observable<any> {
    return this._http.get(`${this.popularUrl}&page=${page}`);
  }

  getTopRatedMovies(page: number = 1): Observable<any> {
    return this._http.get(`${this.topRatedUrl}&page=${page}`);
  }

  getUpcommingMovies(page: number = 1): Observable<any> {
    return this._http.get(`${this.upcomingUrl}&page=${page}`);
  }

  search(title: string): Observable<any> {
    return this._http.get(`${this.searchUrl}&query=${title}`);
  }
}
```



MoviesApp

SearchCmp

MoviesList

MovieCmp

PaginationCmp

# 15 - Routeur



# Routeur

- Il est classique de vouloir associer une URL à un état de l'application
- On veut qu'un utilisateur puisse mettre une page en favori et y revenir plus tard pour donner une meilleure expérience utilisateur
- La partie en charge de ce travail s'appelle un **routeur**

# Routeur

- **Le routeur d'Angular a un objectif simple :**
  - permettre d'avoir des URLs compréhensibles qui reflètent l'état de l'application
  - déterminer pour chaque URL quels composants initialiser et insérer dans la page

# Installation

```
"dependencies": {  
    //....  
    "@angular/router": "3.0.0-rc.1",  
    //...  
}
```

```
System.config({  
  
    //...  
  
    packages: {  
        //...  
        '@angular/router': { main: 'index.js' }  
    }  
});
```

# Création des routes

- On définit nos routes dans un fichier séparé `app.routes.ts` situé à la racine du répertoire `src`

```
import { Routes } from '@angular/router';
import { MoviesListComponent } from './components/movies-list/movies-list.component';
import { RegisterFormComponent } from './components/register-form/register-form.component';

export const ROUTES: Routes = [
  { path: '', component: MoviesListComponent },
  { path: 'register', component: RegisterFormComponent }
];
```

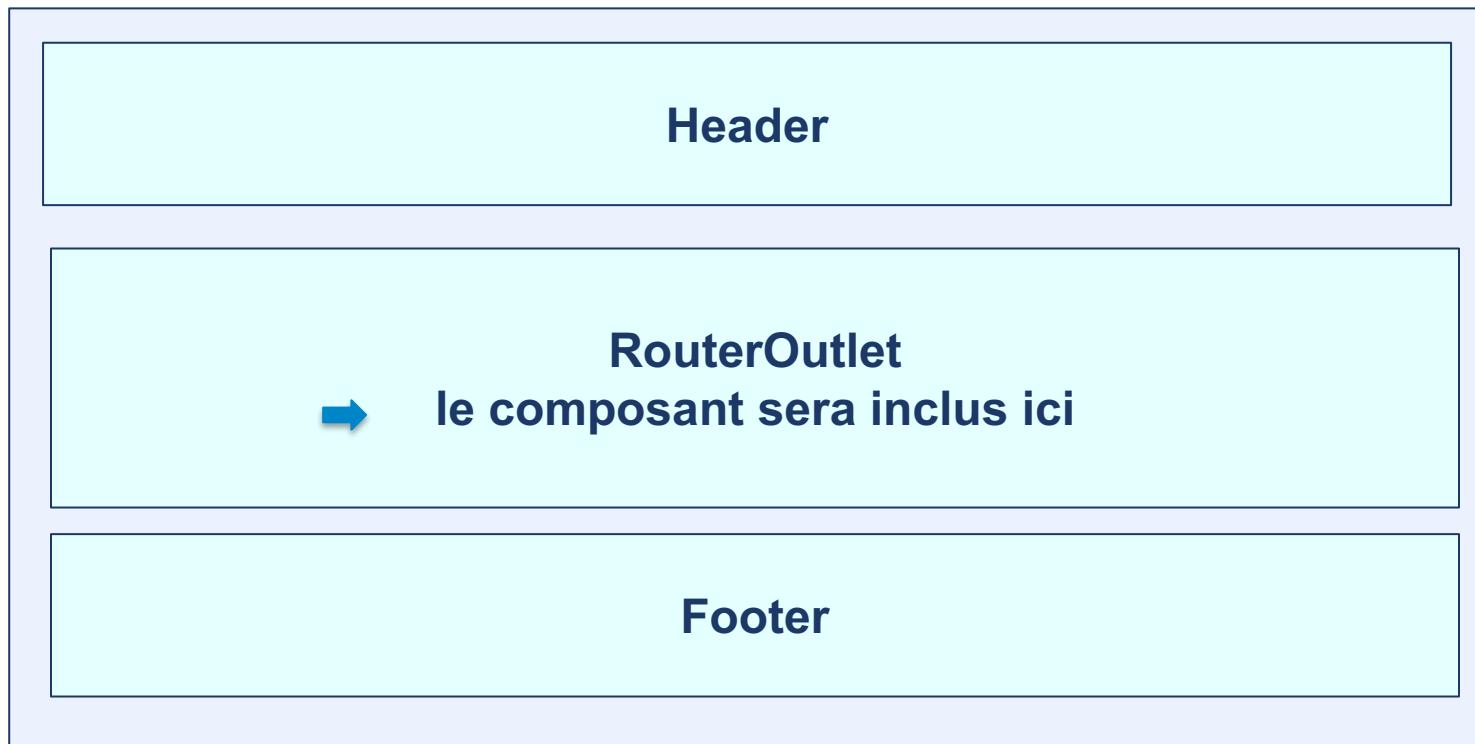
# Importer le RouterModule

```
//...  
import { RouterModule } from '@angular/router';  
  
import { ROUTES } from './app.routes';  
  
//..  
  
@NgModule({  
    imports: [..., FormsModule, RouterModule.forRoot(ROUTES)],  
    //...  
})  
export class AppModule {}
```



# RouterOutlet

- Pour qu'un composant soit inclus dans notre application par le routeur, il faut utiliser un tag spécial dans le template du composant principal : <router-outlet>



# Routeur

```
import { Component } from '@angular/core';
import { MovieService } from './services/movie.service';

@Component({
  selector: 'movies-app',
  template: `
    <a href="" routerLink="/">Movies</a>
    <a href="" routerLink="/register">Sign up</a>
    <router-outlet></router-outlet>
  `,
})
export class MoviesAppComponent { }
```

# Routeur

- La directive **RouterLink** peut recevoir soit une constante représentant le chemin vers lequel on veut naviguer, soit un tableau de chaînes de caractères, représentant le chemin de la route et ses paramètres

```
<a href="" routerLink="/">Home</a>  
  
<!-- same as -->  
  
<a href="" [routerLink]="['/']">Home</a>
```

# Url de base

- Il faut ajouter `<base href="/">` à l'index.html 
- L'élément `<base>` définit l'URL de base à utiliser pour recomposer toutes les URL relatives contenues dans un document

```
<html>
<head>
  <base href="/">
  ...
</head>
....
```

# Style

- La directive **RouterLink** peut être utilisée avec la directive **RouterLinkActive**, qui peut ajouter une classe CSS automatiquement si le lien pointe sur la route courante

```
<a href="" routerLink="/" routerLinkActive="selected-menu">Home</a>
```

# Passer des paramètres

- Il est également possible d'avoir des paramètres dans l'URL

```
export const ROUTES: Routes = [  
    //...  
    { path: 'movies/:id', component: MovieDetailsComponent },  
    //...  
];
```

```
<a href="" [routerLink]="['/movies', movie.id]">Détail film</a>
```

# Passer des paramètres

- On peut récupérer les paramètres passés dans le composant cible

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

import { MovieService } from '../../../../../services/movie.service';

@Component({
  selector: 'movie-details',
  template: `<h2>Détails du film {{ movie.title | json }}</h2>`
})
export class MovieDetailsComponent {
  movie:any = {};
  constructor(private _movieServices: MovieService, private _route: ActivatedRoute) {}

  ngOnInit() {
    let id = this._route.snapshot.params['movieId'];
    this._movieServices.getMovie(id).subscribe(response => {
      this.movie = response.json();
    });
  }
}
```

# 12 - Formulaires



# Formulaires

- **Angular permet de :**
  - **valider les saisies de l'utilisateur**
  - **avoir des champs obligatoires ou non, ou qui dépendent d'un autre champ**
  - **afficher des erreurs si format non valide**
  - **réagir sur les changements de certaines saisies**

# Formulaires

- Angular propose deux façons d'écrire les formulaires:
  - Formulaire piloté par le **template**
  - Formulaire piloté par le **modèle**

# Importer le module forms d'angular

```
import { NgModule } from '@angular/core';
//...
import { FormsModule } from '@angular/forms'; 
//...

@NgModule({
  imports: [BrowserModule, HttpClientModule, FormsModule], 
  //...
})
export class AppModule {

}
```

# Formulaire piloté par le template

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

@Component({
  selector: 'register-form',
  template: `
    <h2>Sign up</h2>
    <!-- we use a local variable #userForm -->
    <!-- and give its value to the register method -->
    <form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
      <div>
        <label>Username</label><input name="username" ngModel>
      </div>
      <div>
        <label>Password</label><input type="password" name="password" ngModel>
      </div>
      <button type="submit">Register</button>
    </form>
    {{ userForm.value | json }}
  `
})
export class RegisterFormComponent {
  register(user) {
    console.log(user);
  }
}
```

# Formulaire piloté par le template

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

@Component({
  selector: 'register-form',
  template: `
    <h2>Sign up</h2>
    <form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
      <div>
        <label>Username</label><input name="username" ngModel>
      </div>
      <div>
        <label>Password</label><input type="password" name="password" ngModel>
      </div>
      <button type="submit">Register</button>
    </form>
    {{ userForm.value | json }}
  `
})
export class RegisterFormComponent {
  register(user) {
    console.log(user);
    user.username = '';
    user.password = '';
  }
}
```



C'est seulement du binding uni-directionnel  
Mettre à jour le modèle ne mettra pas à jour la valeur du champ

# Binding bi-directionnel

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

class User {
  username: string;
  password: string;
}

@Component({
  selector: 'register-form',
  template: `
    <h2>Sign up</h2>
    <form (ngSubmit)="register()">
      <div>
        <label>Username</label><input name="username" [(ngModel)]="user.username">
      </div>
      <div>
        <label>Password</label><input type="password" name="password" [(ngModel)]="user.password">
      </div>
      <button type="submit">Register</button>
    </form>
    {{ user | json }}
  `
})

export class RegisterFormComponent {
  user: any = new User();

  register() {
    console.log(this.user);
    this.user.username = '';
    this.user.password = '';
  }
}
```

The diagram illustrates the bidirectional nature of the binding. Two blue arrows point downwards from the template code to the corresponding lines in the component code. The first arrow points from the 'username' input field in the template to the 'user.username' assignment in the component's constructor. The second arrow points from the 'password' input field in the template to the 'user.password' assignment in the component's constructor.

# Binding bi-directionnel

```
<input name="username" [(ngModel)]="user.username">
```

=

```
<input name="username" [ngModel]="user.username" (ngModelChange)="user.username = $event">
```

- La directive **NgModel** met à jour le modèle lié `user.username` à chaque changement sur l'input: `[ngModel]="user.username"`
- Et elle génère un événement depuis un output nommé **ngModelChange** à chaque fois que le modèle est modifié:  
`(ngModelChange)="user.username = $event"`

# Formulaires (Sous le capot)

- En Angular, un champ du formulaire est représenté par un **FormControl**
- Un **FormControl** a plusieurs attributs et méthodes:

<b>valid</b>	True si le champ est valide, au regard des validations qui lui sont appliquées
<b>errors</b>	un objet contenant les erreurs du champ
<b>dirty</b>	false jusqu'à ce que l'utilisateur modifie la valeur du champ
<b>pristine</b>	l'opposé de dirty
<b>touched</b>	false jusqu'à ce que l'utilisateur soit entré dans le champ
<b>untouched</b>	l'opposé de touched
<b>value</b>	la valeur du champ
<b>valueChanges</b>	un <i>Observable</i> qui émet à chaque changement sur le champ
<b>hasError()</b>	pour contrôler si le contrôle a une erreur donnée

# Formulaires (Sous le capot)

- Un formulaire est lui-même représenté par un **FormGroup**
- Un **FormGroup** a les mêmes propriétés qu'un **FormControl**, avec quelques différences :

<b>valid</b>	true si tous les champs sont valides, alors le groupe est valide
<b>errors</b>	un objet contenant les erreurs du groupe
<b>dirty</b>	false jusqu'à ce qu'un des contrôles devienne "dirty"
<b>pristine</b>	l'opposé de dirty
<b>touched</b>	false jusqu'à ce qu'un des contrôles devienne "touched"
<b>untouched</b>	l'opposé de touched
<b>value</b>	un objet dont les clé/valeurs sont les contrôles et leur valeur respective
<b>valueChanges</b>	un Observable qui émet à chaque changement sur un contrôle du groupe
<b>hasError()</b>	pour contrôler si le contrôle a une erreur donnée
<b>get()</b>	pour récupérer un contrôle dans le groupe

# Validation des formulaires

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit">Register</button>
</form>
```



# Erreurs et soumission

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```



# Afficher les erreurs

```
<h2>Sign up</h2>  
  
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">  
  <div>  
    <label>Username</label><input name="username" ngModel required #username="ngModel">  
    <div *ngIf="username.control.dirty && !username.control.valid">Username is required</div>  
  </div>  
  <div>  
    <label>Password</label><input type="password" name="password" ngModel required #password="ngModel">  
    <div *ngIf="password.control.dirty && !password.control.valid">Password is required</div>  
  </div>  
  <button type="submit" [disabled]="!userForm.valid">Register</button>  
</form>
```

# Style des formulaires

- Angular ajoute et enlève automatiquement des classes CSS sur chaque champ (et le formulaire) pour nous permettre d'affiner le style visuel

State	Class if true	Class if false
Control has been visited	ng-touched	ng-untouched
Control's value has changed	ng-dirty	ng-pristine
Control's value is valid	ng-valid	ng-invalid

# Style des formulaires

```
<style>
  input.ng-invalid {
    border-left: 5px red solid;
  }
  input.ng-valid {
    border-left: 5px green solid;
  }
</style>
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required #username="ngModel">
    <div *ngIf="username.control.dirty && !username.control.valid">Username is required</div>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required #password="ngModel">
    <div *ngIf="password.control.dirty && !password.control.valid">Password is required</div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

# Validateurs angular

- les validateurs fournies par Angular
  - **required**
  - **minlength**
  - **maxlength**
  - **pattern**

# Validateurs angular

```
<style>
  input.ng-invalid {
    border-left: 5px red solid;
  }
  input.ng-valid {
    border-left: 5px green solid;
  }
</style>
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required maxlength="10" #username="ngModel">
    <div *ngIf="username.control.dirty && !username.control.valid">Invalid username</div>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required minlength="6" #password="ngModel">
    <div *ngIf="password.control.dirty && !password.control.valid">Invalid password</div>
  </div>
  <div>
    <label>Phone</label><input name="phone" ngModel required pattern="06([0-9]{8})" #phone="ngModel">
    <div *ngIf="phone.control.dirty && !phone.control.valid">Invalid phone number</div>
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```



# Merci !

