

# Angular 6



Formation Angular



# Programme - Jour 1

## **INTRODUCTION**

ES6, le nouveau JavaScript

ES6 et les modules

TypeScript, le typage du JavaScript

## **LES BASES DU FRAMEWORK**

Comprendre la philosophie du framework

Templating

Angular CLI, un outil pour tout générer

TP : Première application et outillage

## **COMPOSANTS**

Web Components

Décorateurs Angular

Property binding, envoyer des données au composant

Event binding, événements personnalisés

Cycle de vie

TP : Premier composant

## **DIRECTIVES**

Directive : fonctionnement et création

Les directives fournies par Angular

Attribute directives

Structural directives

Directives complexes

TP : Première directive

## **MODULES**

Déclarations d'un module: imports et exports

Les providers d'un module

Différents types de modules : bonnes et mauvaises pratiques

TP : Création d'un module et factorisation d'une librairie externe

## **PIPES**

Les transformateurs fournis

Formater une chaîne

Formater des collections

Utiliser un pipe comme un service

TP : Créer ses propres pipes

# INTRODUCTION

ES6, le nouveau JavaScript

ES6 et les modules

TypeScript, le typage du JavaScript

# ES6, le nouveau Javascript

*Note: ES6 = ES2015 = ECMAScript 6*



# Features principaux

## Let / const versus var :

- **const** : constantes
- **let** : variables

Ce qui équivalait auparavant à **var**.

## Différence :

- var : "top hoisted"

=> Une variable pouvait être utilisée avant d'être déclarée.

- let / const : "block scoped"

=> Ne peuvent être utilisées hors de leur scope

# Features principaux

```
function f() {  
  var x = 1  
  let y = 2  
  const z = 3  
  {  
    var x = 100  
    let y = 200  
    const z = 300  
    console.log('x in block scope is', x)  
    console.log('y in block scope is', y)  
    console.log('z in block scope is', z)  
  }  
  console.log('x outside of block scope is', x)  
  console.log('y outside of block scope is', y)  
  console.log('z outside of block scope is', z)  
}
```



# Features principaux

## Array helpers :

### - forEach :

```
var colors = ['red', 'green', 'blue']  
  
function print(val) {  
    console.log(val)  
}  
  
colors.forEach(print)
```

# Features principaux

## Array helpers :

### - map :

```
var colors = ['red', 'green', 'blue']

function capitalize(val) {
    return val.toUpperCase()
}

var capitalizedColors = colors.map(capitalize)

console.log(capitalizedColors)
```

# Features principaux

## Array helpers :

### - filter :

```
var values = [1, 60, 34, 30, 20, 5]

function lessThan20(val) {
    return val < 20
}

var valuesLessThan20 = values.filter(lessThan20)

console.log(valuesLessThan20)
```

# Features principaux

## Array helpers :

### - find :

```
var people = [  
  {name: 'Jack', age: 50},  
  {name: 'Michael', age: 9},  
  {name: 'John', age: 40}  
]  
  
function teenager(person) {  
  return person.age > 10 && person.age < 20  
}  
  
var firstTeenager = people.find(teenager)  
  
console.log('First found teenager:', firstTeenager.name)
```

# Features principaux

## Classes :

"Sucre syntaxique" ajouté aux héritages et chaînes de prototypes

```
class Point {  
    constructor(x, y) {  
        this.x = x  
        this.y = y  
    }  
  
    toString() {  
        return '[X=' + this.x + ', Y=' + this.y + ']'  
    }  
}
```

# Features principaux

## **Classes :**

Propriétés - éléments à retenir :

- variable de même nom
- propriétés dynamiques (calculées par la classe)
- méthodes

# Features principaux

## Classes :

```
const color = 'red'
const point = {
  x: 5,
  y: 10,
  color,
  toString() {
    return 'X=' + this.x + ', Y=' + this.y +
      ', color=' + this.color
  },
  [ 'prop_' + 42 ]: 42
}
```

# Features principaux

## Template strings :

```
function hello(firstName, lastName) {  
  return `Good morning ${firstName} ${lastName}!  
How are you?`  
}
```

```
console.log(hello('Jan', 'Kowalski'))
```



# Features principaux

## Arguments de fonctions par défaut :

```
function sort(arr = [], direction = 'ascending') {  
    console.log('I\'m going to sort the array',  
        arr, direction)  
}
```

```
sort([1, 2, 3])  
sort([1, 2, 3], 'descending')
```

# Features principaux

## Les promesses :

But : traiter des opérations **asynchrones**.

Elles peuvent être chaînées :

- then() : callback
- catch() : erreur

# Features principaux

## Les promesses :

```
function asyncFunc() {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            const result = "Un résultat !"  
        }, 1)  
    });  
}  
  
for (let i=0; i<10; i++) {  
    asyncFunc()  
        .then(result => console.log('Result is: ' + result))  
        .catch(result => console.log('Error: ' + result))  
}
```

# Typescript, le typage du Javascript



Typescript : '**superset**' de Javascript => apporter un **typage** à Javascript.

*Note: Typescript est une spécificité d'Angular*

# Types statiques vs dynamiques

JavaScript est **typé dynamiquement** :

```
let fooBar = 'Foo Bar'; fooBar = 25;      //
```

Le problème :

```
const fooCat = { name: 'Chat', color: 'brown' };  
const fooDog = { play: true, color: 'black' };  
const printColor = animal => console.log(animal.color); //  
const printColor = animal => console.log(animal.play); //
```

# Types statiques vs dynamiques

Sans typage, peu de ressources disponibles :

- lire la documentation
- lire le code de la fonction
- ..

=> complexifie code + utilisation API / librairies...

# Types statiques vs dynamiques

=> Le typage Typescript permet :

- un code plus efficace (IDE)
- debuggage et testabilité simplifiés

# Les bases de TypeScript :

Caractéristiques:

- syntaxe proche de Javascript
- extension .ts

```
const fooNumber: number = 0;  
const fooString: string = 'Blah !';
```

```
// Types génériques  
const fooArray: Array<string> = ['Blah !'];
```

```
// Custom types  
const fooType: FooType = new FooType()  
const fooTypeArray: Array<FooType> = [new FooType()];
```



# Les bases de Typescript :

Concrètement :

```
const fooBlah: Array<string>
```

```
this.fooBlah.push('Blah blah')  
// Tout fonctionne correctement
```

```
this.fooBlah.push({parler: true, contenu: 'Blah blah'})  
// error TS2345  
// Argument of type {} is not assignable to parameter of  
// type 'string'.
```

# Les bases de TypeScript :

Et si on ne connaît pas le type ?

=> type dynamique "**any**"

```
const undefinedType: any
```

On peut aussi utiliser l'union de types :

```
const undefinedType: string | number  
undefinedType = 'Blah'  
undefinedType = 25 // 🙌
```

# Que peut-on typer ?

Variables :

```
let n: number = 1
const s: string = 'Hello'
```

Paramètres de fonctions :

```
function f(i: number) { ... }
```

Retour de fonction :

```
function f(): number {
  return 42
}
```

# Types basiques :

Booleans:

```
let fini: boolean = false
```

Numbers:

Nb : En plus des décimales et hexadécimales, Typescript supporte aussi les types de littéraux binaires et octaux introduits par ES6.

```
let decimal: number = 6;  
let hex: number = 0xf00d;  
let binary: number = 0b1010;  
let octal: number = 0o744;
```

# Types basiques :

Strings:

```
let color: string = "blue";  
color = 'red';
```

Nb : Typescript supporte aussi les "template strings", introduites par ES6.

```
let fullName: string = `Bob Bobbington`;  
let age: number = 37;  
let sentence: string = `Hello, my name is ${ fullName }.  
  
I'll be ${ age + 1 } years old next month.`;
```

# Types basiques :

Arrays :

Deux façons de typer les arrays :

```
let list: number[] = [1, 2, 3]; //elemType[]  
let list: Array<number> = [1, 2, 3]; // Array<elemType>:
```

Tuple :

Permet de typer un tableau où le type de certains éléments est connu :

```
// Declare a tuple type  
let x: [string, number];  
// Initialize it  
x = ["hello", 10]; // OK  
// Initialize it incorrectly  
x = [10, "hello"]; // Error
```

# Types implicites :

Lorsqu'il s'agit de types basiques, le compilateur associe par défaut les types des variables non annotées :

```
let n = 1                                // let n: number = 1

let s = "Hello World"                   // let s: string = "Hel

n = s;                                  // COMPILATION ERROR
s = n;                                  // COMPILATION ERROR

function f() {                           // function f(): string
    return "hello"
}
```

# Les bases de Typescript :

## Déclarer un custom type :

=> créer une **classe** ou une **interface**

```
interface Animal {  
    nom: string,  
    bruit: string  
}
```

```
function anime(arg: Animal)  
    { return `${arg.name} fait ${arg.bruit}` }
```

```
const Canard: Animal = {nom: 'Canard', bruit: 'coin coin'}  
anime(this.Canard) // Ok
```



# Les bases de Typescript :

## Classes et interfaces :

Un type peut être déclaré dans Typescript par une classe ou une interface :

```
interface Poulet {  
    race: string,  
    pleinAir: boolean,  
    fermier: boolean  
}
```

```
class Canard {  
    constructor( public race: string,  
                 public fermier: boolean,  
                 public magret: boolean) {}  
}
```

# Les bases de Typescript :

## Classes et interfaces :

Différence --> une interface ne peut pas être instanciée :

```
const poulet : Poulet {  
  race: 'Gallus domesticus',  
  pleinAir: true,  
  fermier: true  
}
```

```
const canard: Canard = new Canard ('Canard de Barbarie',  
  true, true)
```

# Les bases de TypeScript :

## Nb - Le réel intérêt des interfaces :

Interface (classe 'allégée') => intérêt visible après compilation:

- l'interface sert à **vérifier les types** --> effacée de l'output final
- la classe, même non instanciée, est considérée comme déclarée --> présente dans l'output final.

# Les bases de Typescript :

## Les décorateurs :

Déclarations particulières :

- propres à Typescript
- attachées à des classes, méthodes, paramètres, etc.

Syntaxe: **@expression** (*'expression' --> une fonction appelée à l'instanciation de l'élément décoré*)

En Angular: attacher des "**métadonnées**" propres au framework

# Les bases de TypeScript :

## Les décorateurs :

Un exemple de décorateur très commun : le composant !

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

@Component({
  selector: 'composant-exemple',
  template: '<div>Woow un composant !</div>',
})
export class ExempleComposant {
  constructor() {
    console.log('Hello, je suis un composant');
  }
}
```

# Les bases de TypeScript :

## Les décorateurs :

Décorateur = fonction exécutée lors de l'instanciation de la classe

=> ici permet à Angular de :

- définir 'ExempleComposant' comme composant
- configurer 'ExempleComposant'

# Les bases de Typescript :

TP :

Installer le compilateur Typescript: **npm install -g typescript**

Créer un fichier en Ts :

- déclarer une **interface** + instancier un objet l'implémentant
- instancier une **classe** + instancier un objet de cette classe
- utiliser ces objets dans une méthode avec un **feature ES6** (helper functions, template literals...)

Puis compiler en Js: **tsc nomDuFichier.ts**

# LES BASES DU FRAMEWORK

Comprendre la philosophie du framework

Templating

Angular CLI, un outil pour tout générer

TP : Première application et outillage



# Comprendre la philosophie du framework

**Angular** ("Angular 2") est un framework :

- placé côté client
- fonctionnel sur navigateur, web workers, mobiles, serveurs (Angular Universal)

Release : développé par l'Angular Team et paru en septembre 2016.

*Nb: Il s'agit d'une refonte totale d'AngularJS (créé en 2009) avec lequel il ne doit pas être confondu.*

# Framework côté client

Différence framework front / back :

- back: navigateur construit le DOM en "parsant" un document HTML prêt à être rendu
- front: navigateur construit le DOM en interprétant un script

**Avantages :** limite les interactions serveurs

=> *navigation très fluide (en particulier en web mobile).*

**Inconvénient :** Un chargement au démarrage qui peut être long (conseillé < 250kb)

# Réponse serveur au lancement d'une application Angular

```
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>Angular Sample App</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>

<body>

// Le composant racine de notre application
<app-root></app-root>

// L'ensemble des scripts permettant la construction du DOM à partir du composant root
<script type="text/javascript" src="inline.bundle.js"></script>
<script type="text/javascript" src="polyfills.bundle.js"></script>
<script type="text/javascript" src="scripts.bundle.js"></script>
<script type="text/javascript" src="styles.bundle.js"></script>
<script type="text/javascript" src="vendor.bundle.js"></script>
<script type="text/javascript" src="main.bundle.js"></script>

</body>
</html>
```

# Le concept des SPA

Single Page Application :

une fois instanciée au chargement, l'application n'a plus besoin de reload auprès du server pour fonctionner.



# Questions

Est-ce qu'on peut avoir plusieurs pages sur une SPA ?

# Questions

Est-ce qu'on peut avoir plusieurs pages sur une SPA ?

Angular => module Router, permet de "simuler" différentes Urls.



# Exemple d'instanciation d'un Router

Le Router est instancié dans le fichier  
src/app/app.module.ts :

```
import { RouterModule, Routes } from '@angular/router';
// autres imports

const appRoutes: Routes = [
  { path: 'route1', component: Route1Component },
  // on peut aussi inclure des paramètres dans l'url
  { path: 'route2/:parametre2', component: Route2Component },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
    // autres imports
  ],
  ...
})
export class AppModule { }
```

# Angular CLI, un outil pour tout générer

L'Angular CLI :

- **ng new +nom** : boilerplate
- **ng generate** (ng g) : nouveaux éléments
- **ng serve** : serveur de développement
- **ng build** : fichiers distants
- etc...

Installation :

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



# Création de votre première application Angular

Extrêmement simple :

```
ng new applicationAngular  
cd applicationAngular  
ng serve
```

=> <http://localhost:4200> ^^/

# Architecture standard d'une application Angular

```
// Tout ce qui va concerner les tests end to end
|- e2e/
    |----- app.e2e-spec.ts
    |----- app.po.ts
    |----- tsconfig.e2e.json

// les dépendances avec npm
|- node_modules/

// l'endroit où les fichiers de build seront mis
|- dist/
```

```
// Le dossier où vous allez modifier vos fichiers de code
// Là où va se trouver vos composants, services, etc..
```

```
| - src/
|   |----- app/
|       |----- app.component.css | html | spec.ts | ts
|       |----- app.module.ts
|   |----- assets/
|   |----- environments/
|       |----- environment.prod.ts | ts
|   |----- favicon.ico
|   |----- index.html
|   |----- main.ts
|   |----- polyfills.ts
|   |----- styles.css
|   |----- test.ts
|   |----- tsconfig.app.json
|   |----- tsconfig.spec.json
|   |----- typings.d.ts
```

```
// la configuration globale de votre application
|- .angular-cli.json // fichier de configuration principal
|- .editorconfig      // peut être utilisé dans VS Code
|- .gitignore
|- karma.conf.js
|- package.json
|- protractor.conf.js
|- README.md
|- tsconfig.json
|- tslint.json
```

# Les principaux fichiers

## app.module.ts :

```
/* imports JavaScript */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';

import { AppComponent } from './app.component';

/* classe AppModule avec le décorateur @NgModule */
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

# Les principaux fichiers

## **app.module.ts :**

- référencer tous les imports
  - instancier une nouvelle classe NgModule
- => *métadata : configurent la compilation Angular des différents éléments*

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

# Les principaux fichiers

## **app.module.ts :**

Détail de l'objet paramètre de @NgModule :

- *declarations*— les composants de l'application
- *imports*—les modules importés  
([BrowserModule](#) permettant de rendre l'application dans un navigateur, etc)
- *providers*—les providers de services
- *bootstrap*—le composant *root* que Angular crée et insert dans l'`index.html` (point d'entrée) et permettant l'initialisation de l'application

Nb: AppComponent, composant root par défaut, est présent dans *declarations* et *bootstrap*.

# Les principaux fichiers

**app.component.ts - notre premier composant !**

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

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



# Les principaux fichiers

**app.component.ts** - notre premier composant !

AppComponent:

- seul composant présent dans index.html
- instanciera '**en cascade**' l'ensemble de l'app

```
<!doctype html>
<html lang="en">
<head><title>Une app Angular</title>...</head>
<body>
  <!-- appComponent instancié ici ! -->
  <app-root></app-root>
</body>
</html>
```

# Les principaux fichiers

**app.component.ts - notre premier composant !**

Une cascade de composants :

```
<app-composant1></app-composant1>  
<app-composant2></app-composant2>  
<app-composant3></app-composant3>
```

*Nb: n'importe quel composant peut instancier un composant 'enfant' - pas réservé au composant racine.*

# COMPOSANTS

Web Components

Décorateurs Angular

Property binding, envoyer des données au composant

Event binding, évènements personnalisés

Cycle de vie

TP : Premier composant

# Web components

## Principe:

permettent la création de **nouveaux tags HTML** personnalisés

## API:

**CustomElementRegistry.define()**, avec en arguments :

- nom de l'élément
- un objet de classe définissant le comportement de l'élément  
(- options facultatives)

# Web components

Nb: fonctionnalité implémentée nativement dans certains navigateurs

- pas de librairies, framework...
- pas entièrement supportée (polyfills)

# Web components

```
customElements.define('word-count', WordCount,  
  { extends: 'p' });
```

```
class PopUpInfo extends HTMLElement {  
  constructor() {  
    // Toujours appeler "super" d'abord dans le constr  
    super();  
  
    // Ecrire la fonctionnalité de l'élément ici  
    ...  
  }  
}
```

# Composants Angular :

Reprennent le **même principe** (custom tags) + ajout de **nombreuses API** (services, routing, communication server, etc).

Nb: En réalité, composants webs et angular **diffèrent** car:

- composants webs => l'API des custom elements (fonction **define()**, ..)
- composants Angular => propre système de création d'éléments custom (**ngFactories**).

# Composants Angular :

```
@Component({  
  selector: 'greet',  
  template: 'Hello {{name}}!'  
})  
class Greet {  
  name: string = 'World';  
}
```

```
                                // Rendu HTML  
<app-great>Hello World!</app-great>
```



# Un nouveau composant :

Création d'un nouveau composant :

```
ng g c exemple
```

La CLI se charge de :

- création de 4 fichiers :  
*exemple.component.ts | html | css | spec.ts (fichier de test)*
- configuration du composant dans NgModule

# Un nouveau composant :

Fichier app.module.ts actualisé :

```
...  
import { ExempleComponent } from './exemple/exemple.compon  
  
@NgModule({  
  declarations: [..., ExempleComponent],  
  imports: [...],  
  providers: [...],  
  bootstrap: [...]  
})  
export class AppModule { }
```

# Décorateurs composants

Un composant est une classe pourvue d'un décorateur `@Component({})` :

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

@Component({
  selector: 'un-composant',
  template: '<h1>Wow - un composant !</h1>'
})
class UnComposant {}
```

`<un-composant></un-composan`

# Décorateurs composants

Le décorateur `@Component({})` permet de configurer le composant en passant des métadonnées en paramètre au décorateur :

```
@Component({  
  changeDetection?: ChangeDetectionStrategy  
  viewProviders?: Provider[]  
  moduleId?: string  
  templateUrl?: string  
  template?: string  
  styleUrls?: string[]  
  styles?: string[]  
  animations?: any[]  
  encapsulation?: ViewEncapsulation  
  interpolation?: [string, string]  
})
```

# Décorateurs composants

En pratique, on se sert principalement des paramètres `selector`, `template` (ou `templateUrl`) et `styles` (ou `styleUrls`) :

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

@Component({
  selector: 'un-composant',
  template: `<h1>Wow - un composant !</h1>
             <div>Je suis un composant</div>`,
  styles: [`h1 { color: red}
           div {color: blue}`]
})
class UnComposant {}
```

# Décorateurs composants

Et instantiation de notre nouveau composant, dans app.component.html par exemple :

Wow – un appel à notre nouveau composant :  
`<app-exemple></app-exemple>`

# Décorateurs composants

Question : Le style défini dans un composant parent sera-t-il appliqué à un composant enfant ?

# Décorateurs composants

Question : Le style défini dans un composant parent sera-t-il appliqué à un composant enfant ?

Non => principe de **View Encapsulation**, ( composant = vue isolée)

Pour **forcer l'héritage** :

```
@Component({
  selector: 'no-encapsulation',
  templateUrl: './no-encapsulation.html',
  styleUrls: ['./no-encapsulation.css'],
  encapsulation: ViewEncapsulation.None // Supprime le principe
})
class NoEncapsulation {}
```



# Data binding

## échanger des données

Angular nous permet d'échanger des données entre :

- le composant et la vue
  - one way (vue => composant // composant => vue)
  - two ways (vue <=> composant)
- entre les différents composants
  - par input (parent => enfant)
  - par output (enfant => parents & siblings)

# Data binding

**composant  $\Leftrightarrow$  vue**

Un grand avantage des composants est de nous permettre de manipuler des données dans le DOM - pour cela, nous avons besoin de pouvoir lier les datas entre le composant (plus exactement, son instance, déclarée dans le fichier .ts - que l'on peut assimiler au 'Model') et son template (la 'Vue').

Pour cela, Angular met à notre disposition plusieurs façons de 'bind' les datas.

# composant / vue: one-way

## 1. Composant => vue :

### A. Interpolation : {{ .. }}

La syntaxe `{{ var }}` permet de lier d'inclure une variable déclarée dans le composant dans le template

```
@Component({  
  selector: 'app-exemple',  
  template: 'Je suis une {{ foo }}'  
})  
export class ExempleComponent {  
  foo : string = 'interpolation'  
}
```

# composant <=> vue: one way

## B. Property binding : [ .. ]

La syntaxe `[prop] = "var"` permet de lier la propriété d'un attribut d'un élément du DOM à une variable.

```
@Component({
  selector: 'app-exemple',
  template: '<div [property]="binding">
    Une div avec un attribut bindé
  </div>'
})
export class ExempleComponent {
  binding : string = 'nouvelle valeur'
}
```

# composant <=> vue: one way

## 2. Vue => composant :

### Event binding : ( .. )

La syntaxe `(event) = "methode()"` permet de lier un événement provenant de la vue à une variable ou méthode déclarée dans le composant.

```
(focus)="myMethod( )" // An element has received focus  
(blur)="myMethod( )" // An element has lost focus
```

```
(submit)="myMethod( )" // A submit button has been pre
```

```
(scroll)="myMethod( )" // A scroll event has been triggered
```

# composant $\Leftrightarrow$ vue: two ways

## 2. Vue $\Rightarrow$ composant $\Rightarrow$ vue :

### La directive ngModel : [( ngModel )]

La syntaxe [(ngModel)] = "variable" permet de lier la valeur d'un input à une variable d'un composant - qui peut ensuite être renvoyée actualisée à la vue.

*En réalité, ngModel est une combinaison de l'event binding et de la property binding.*

# composant <=> vue: two ways

## La directive ngModel : [( ngModel )]

```
<div>
  <!-- la variable 'model' est updatée à chaque fois que
        l'utilisateur modifie la valeur de l'input, et
        updatée dans le composant
  -->
  <input [(ngModel)]="model">
  <!-- le composant 'renvoie' la valeur actualisée
        à la vue
  -->
  <p>Valeur actualisée : {{model}}</p>
</div>
```

# composant <=> vue: two ways

## La directive ngModel : Implémentation

Nb: ngModel nécessite l'import préalable de FormsModule :

```
// app.module.ts
```

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

```
@NgModule({  
  imports: [ FormsModule, ... ],  
  ... })
```



# composant <=> vue: two ways

## La directive ngModel : Implémentation

```
// app.component.ts

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

@Component({
  selector: 'app-root',
  template: `<input [(ngModel)]="model">
               <p>Notre data-binding : {{model}}</p>`
})
export class AppComponent {
  model: string; // Déclaration de la variable de bir
}
```

# Data binding

composant  $\Leftrightarrow$  vue

**TP 1:**

Créer un composant avec:

- une interpolation (controller  $\Rightarrow$  vue)
- un property-binding (controller  $\Rightarrow$  vue)
- un event-binding (vue  $\Rightarrow$  controller)
- un ng-model (controller  $\Leftrightarrow$  vue)

# Data binding

parent  $\Leftrightarrow$  enfant

Parent (controller / vue)  $\Rightarrow$  enfant (vue)

- principe de transclusion

Parent (controller / vue)  $\Rightarrow$  enfant (controller / vue)

- property / event bindings (parent)

- @Input / @Output (enfant)

*Nb: Parents / enfants **directs** uniquement.*

# Inter-comp :

**Transclusion : Parent (controller / vue) => enfant (vue)**

=> 'slot' pour insertion dynamique de contenu

```
@Component({  
  selector: 'app-transclusion',  
  templateUrl: './transclusion.component.html'  
})  
export class TransclusionComponent {}
```

```
<!-- transclusion.component.html -->  
<div class="transclusion-header">Je suis un header fixé<  
<ng-content></ng-content>    <!-- contenu dynamique ici -  
<div class="transclusion-header">Je suis un footer fixé<
```

# Inter-comp :

**Transclusion : Parent (controller / vue) => enfant (vue)**

Instanciation dans le composant parent :

```
<!-- parent.component.html -->  
<p>Intégration de contenu dynamique :</p>  
<transclusionComponent>  
    Wow - du contenu intégré dynamiquement !  
</transclusionComponent>
```

# Inter-comp :

**Transclusion : Parent (controller / vue) => enfant (vue)**

Output :

```
<p _ngcontent-c0="">Intégration de contenu dynamique :</p>
<app-transclusion _ngcontent-c0="" _nghost-c1="">
  <div _ngcontent-c1="" class="transclusion-header">
    Je suis un header fixé</div>
    Wow - du contenu intégré dynamiquement !
    <div _ngcontent-c1="" class="transclusion-header">
      Je suis un footer fixé</div>
</app-transclusion>
```

# Inter-comp :

**Input : Parent (controller / vue) => enfant (controller)**

Principe:

- instanciation enfant avec donnée à transmettre en attribut
- 'reception' de l'attribut par controller enfant via @Input

# Inter-comp :

**Input : Parent (controller / vue) => enfant (controller)**

Instanciación enfant :

```
// Data non liée au controller parent  
<child data="dataFromParent"></child>
```

```
// Data liée au controller parent  
<child [data]="dataFromParent"></child>
```



# Inter-comp :

**Input : Parent (controller / vue) => enfant (controller)**

Controller enfant :

```
// Ne pas oublier d'importer le décorateur Input
import { Component, Input } from '@angular/core';

@Component({
  selector: 'child',
  ...
})
export class Child {

  @Input() item: WineItem;

}
```

# Inter-comp :

**Output : Enfant (controller) => parent (vue puis controller)**

Principe :

- à l'instanciation, lier un 'event' enfant à un callback parent

=> envoi de données:

- l'enfant émet un 'event'
- l'event déclenche un callback parent
- le parent est notifié de la donnée

# Inter-comp :

**Output : Enfant (controller) => parent (vue puis controller)**

Composant parent :

```
// ParentComponent.html  
<child (receivedData)="newDataCallback($event)"></child>
```

```
// ParentComponent.ts  
@Component({ ... })  
export class ParentComponent {  
  data: any;  
  newDataCallback(event: any): void {  
    // update d'une variable locale  
    this.data = event  
  }  
}
```

# Inter-comp :

**Output : Enfant (controller) => parent (vue puis controller)**

Controller enfant :

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

@Component({ selector: 'child', ... })
export class ItemDetailsComponent {

    // Lier la propriété 'dataToSend' à l'attribut 'receivedData'
    @Output('receivedData') dataToSend: EventEmitter<any>;

    constructor() {
        this.dataToSend = new EventEmitter<ISelectEvent>()
    }

    sendData(data): void { this.itemSel.emit({ data }) }
}
```

# Data binding

Parent  $\Leftrightarrow$  Enfant(s)

TP 2:

Créer un ensemble composant parent / enfant avec :

- une **transclusion** (vue parent  $\Rightarrow$  vue enfant)
- un **input** statique et dynamique (vue/controller parent  $\Rightarrow$  controller enfant)
- un **output** (controller enfant  $\Rightarrow$  controller parent) avec un objet de données (ex: click + origine)

# Cycles de vie

## Lifecycle hooks :

Permettent d'appeler des callbacks à différents moments du cycle.

```
@Component(...)  
export class MyComponent {  
  constructor() { }  
  ngOnInit() {}  
  ngOnDestroy() {}  
  ngDoCheck() {}  
  ngOnChanges(records) {}  
  ngAfterContentInit() {}  
  ngAfterContentChecked() {}  
  ngAfterViewInit() {}  
  ngAfterViewChecked() {}  
}
```

# Caractéristiques composant

## Lifecycle hooks :

En pratique, on utilise principalement `ngOnInit()` en addition au constructeur :

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

@Component()
export class myComponent implements OnInit{
  constructor() {}
  ngOnInit() { /* code à executer à l'initialisation */
}
```

**Nb : ne pas oublier d'importer et d'implémenter le hook.**

# DIRECTIVES

Fonctionnement et création

Les directives fournies par Angular

Attribute directives

Structural directives

Directives complexes

TP : Première directive



# Les directives :

## Le problème :

Composants => créer des vues dans le DOM

Mais ne permettent pas de **modifier des éléments existants** du DOM, comme :

- structure des noeuds
- comportement des éléments

# Les directives :

## Définition :

Classe associée à une balise (idem composant), mais sans vue.

*Nb : composants = directives particulières jusqu'à Angular 2.*

## Intérêt :

Faire exécuter du code au navigateur modifiant des éléments présents dans le DOM.

# Les directives :

## Déclaration :

```
import { Directive, Renderer2, ElementRef, HostListener } from '@angular/core';

@Directive({selector: '[appHoverEffect]'})
export class HoverEffectDirective {

    constructor( private renderer : Renderer2, private elementRef: ElementRef ) {}

    @HostListener( 'mouseover' )
    applyHover() {
        this.renderer.setStyle(this.elementRef.nativeElement, 'color', 'red')
    }
}
```

# Les directives :

## Instanciation :

```
// Element standard du DOM
```

```
<div appHoverEffect> App hover ! </div>
```

```
// Autre
```

```
<directedComponent appHoverEffect><directedComponent>
```

***Nb:*** nouvelle instance de directive créée à chaque rencontre de la balise.

# Les directives :

## Exemples d'utilisation:

- itération d'un élément DOM existant sur une liste - *structure*
- ajout / retrait dynamique d'un nœud - *structure*
- règles CSS à des éléments - *comportement*
- > pourquoi une directive plutôt qu'une classe ?
- réponse à un événement - *comportement*
- etc...

# Les directives :

## Types de directives :

- **les directives structurelles** : modifient la structure du DOM
- **les directives d'attributs** : modifient l'apparence ou le comportement de certains éléments

# Directives structurelles :

**Syntaxe : \*directive = "template expression"**

- ' \* ' : obligatoire pour directives structurelles - Nb :  
contrairement aux directives d'attribut, pas de [...] ou de (..)
- 'template expression' : expression retournant une valeur évaluée par Angular (contexte : leur composant)

```
<div *ngIf="Angular"> Ok // </div>  
<div *ngIf="AngularVersion > 4"> 🔥🔥 </div>
```

Nb : les 'template expressions' sont soumises à certaines limitations par rapport au javascript standard (chaînage d'expressions, etc)

# Dir. structurelles : NgIf

Permet d'afficher un élément selon la valeur d'une expression booléenne.

```
<div *ngIf="false"></div>  
<div *ngIf="a > b"></div>  
<div *ngIf="str == 'yes'"></div>  
<div *ngIf="myFunc()"></div>
```

Attention, NgIf fait apparaître ou disparaître l'élément auquel elle est attribuée du DOM - c'est donc par exemple différent d'un 'display: none'.

Nb : appliquée à un composant, ngIf induit l'initialisation ou la destruction de celui-ci - cf la notion de 'cycle de vie' d'un composant.



# Dir. structurelles : NgSwitch

Fonctionne comme un 'switch case' traditionnel :

```
<div *ngIf="myVar == 'A'">Var is A</div>  
<div *ngIf="myVar == 'B'">Var is B</div>  
<div *ngIf="myVar == 'C'">Var is C</div>  
<div *ngIf="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var is something else</div>
```

équivalent à :

```
<div [ngSwitch]="myVar">  
  <div *ngSwitchCase="A">Var is A</div>  
  <div *ngSwitchCase="B">Var is B</div>  
  <div *ngSwitchCase="C">Var is C</div>  
  <div *ngSwitchDefault>Var is something else</div>  
</div>
```

# Dir. structurelles : NgFor

Permet d'itérer sur un array :

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

@Component({
  selector: 'todo-list',
  template: `
    <h2>Todos</h2>
    <ul>
      <li *ngFor="let todo of todos">{{todo}}</li>
    </ul>
  `
})
export class TodoList {
  todos = ['Walk the dog', 'Stay in bed', 'Code more']
}
```

# Directives d'attributs

Modifient le comportement ou l'apparence d'un élément.

=> Doit avoir un sélecteur CSS

```
@Directive({  
  selector: '[doNothing]' })  
export class DoNothingDirective {  
  constructor() {  
    console.log('Do nothing directive');  
  }  
}
```

# Directives d'attributs

Les sélecteurs :

- un élément : footer.
- une classe (rare) : .alert.
- un attribut (le plus fréquent) : [color].
- un attribut avec une valeur : [color=red].
- une combinaison : footer[color=red].

# Directives d'attributs

Un exemple de sélecteur :

```
@Directive({
  selector: 'div.loggable[logText]:not([notLoggable=true])'
})
export class ComplexSelectorDirective {
  constructor() {
    console.log('Complex selector directive');
  }
}
```

# Dir. d'attributs 'built-in' :

Angular nous fournit 3 directives d'attributs principale :

- ngStyle
- ngClass
- ngModel

+ d'autres fournies par modules supplémentaires: FormsModule, RouterModule, Angular Material...

# Dir. d'attributs - NgStyle

Attribuer un objet de style CSS dynamique:

```
currentStyle: {};  
setCurrentStyle() {  
  // Propriétés CSS  
  this.currentStyle = {  
    'color': this.color,  
    'font-size': this.fontSize  
  };  
}
```

```
<!-- Avec ngStyle -->
```

```
<div [ngStyle]="currentStyle">Un style spécial !</div>
```

```
<!-- Traditionnel -->
```

```
<div [style.color]="color">Un style moins spécial</div>
```

# Dir. d'attributs - NgClass

Attribuer des classes CSS par :

- string
- array
- objet

```
currentClasses: {};  
setCurrentClasses() {  
    // valeurs booléennes attribuées à des noms de classes  
    this.currentClasses = {  
        'saveable': this.canSave,  
        'modified': !this.isUnchanged,  
        'special': this.isSpecial  
    };  
}
```



# Dir. d'attributs - NgClass

Objet attribué à ngClass :

```
<div [ngClass]="currentClasses">This div is initially saved  
unchanged, and special</div>
```

Binding traditionnel :


```
<!-- toggle the "special" class on/off with a property -->  
<div [class.special]="isSpecial">The class binding is special</div>
```

# Rmq NgStyle - NgClass

Modification dynamique de la valeur attribuée à la directive par modification d'une variable contrôleur :

src/app/app.component.ts

```
currentClasses: {};  
setCurrentClasses() {  
    // CSS classes: added/removed per current state of component  
    properties  
    this.currentClasses = {  
        'saveable': this.canSave,  
        'modified': !this.isUnchanged,  
        'special': this.isSpecial  
    };  
}
```



# Rmq NgStyle - NgClass

Note :

It's up to you to call `setCurrentClasses()`, both initially and when the dependent properties change.

=> recréer l'objet style en callback

# Rmq NgStyle - NgClass

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

  this.color;
  this.fontSize;
  currentStyle: {};
  setCurrentStyle() {
    this.currentStyle = {
      'color': this.color,
      'font-size': this.fontSize
    };
  }

  changeAStyle(color, fontSize) {
    this.color = color;
    this.fontSize = fontSize;
    // Recréer l'objet style en callback
    this.setCurrentStyle()
  }
}
```

# Rmq NgStyle - NgClass

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

  currentStyle = {}
  currentStyle = {}
  set color(value) {
    this.currentStyle['color'] = value
  }
  set fontSize(value) {
    this.currentStyle['font-size'] = value
  }

  changeAStyle(color, fontSize) {
    this.color = color;
    this.fontSize = fontSize;
    console.log(this.currentStyle) // actualisé
  }

}
```

# (Additionnel) Getters & setters

Fonctions permettant de lier dynamiquement, dans un objet, **une propriété A à une propriété B** en exécutant un callback :

- quand on accède à B = **getter**
- quand la valeur de A est modifiée = **setter**

# (Additionnel) Getters & setters

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

  prop1 = 1
  get prop2() {
    return this.prop1 + 3;
  }
  prop3 = this.prop1 + 3;

  testProp() {
    this.prop1 = 3;
    console.log(this.prop2); // 6 (actualisée)
    console.log(this.prop3); // 4 (non-actualisée)
  }
}
```

# (Additionnel) Getters & setters

```
@Component({})  
export class testSetter {  
  
  set prop1(value) {  
    this.prop2 = value + 3  
  };  
  prop2: any;  
  prop3: any = this.prop1 + 3;  
  
  constructor() { this.prop1 = 1; }  
  
  testProp() {  
    this.prop1 = 3;  
    console.log(this.prop2); // 6 (actualisée)  
    console.log(this.prop3); // NaN  
  }  
}
```



# Dir. d'attributs - Custom

On peut facilement créer nos propres directives d'attributs :

```
@Directive({
  selector: '[loggable]'
})
export class InputDecoratorOnSetterDirective {
  @Input('logText')
  set text(value) {
    console.log(value);
  }
}
```

```
<div loggable logText="Hello">Hello</div>
// notre directive console.log "Hello"
```

# Directives & décorateurs

Quelques décorateurs utiles :

- @Input() : accéder aux valeurs des attributs (attribuées dans le template du composant parent)
- @HostListener() : accéder aux événements se produisant se l'élément hôte
- @HostBinding() : accéder aux valeurs des propriétés

```
@HostListener('mouseover') onMouseOver() {  
    console.log('Host listener ...')  
}
```

# Directives

## TP:

- Structurelles : implémenter un **\*ngIf** (ex: toggle button) et un **\*ngFor**
- Attribut : **ngStyle** et **ngClass** (exercice des couleurs)
- Custom : créer une animation au hover + au scroll (avancé)

# Composants / Binding / Directives

**TP de synthèse :**

**Base d'une application de display d'articles**

- **Architecturer** l'app + pose premiers composants
- **Composant display** à partir d'articles à partir d'un mock
- **Interaction composants** à partir d'events du DOM  
(*avancé*) : réception d'events écoutées par un composant A dans un composant B

# MODULES

Déclarations d'un module: imports et exports

Les providers d'un module

Différents types de modules : bonnes et mauvaises pratiques

TP : Création d'un module et factorisation d'une librairie externe

# @NgModule

Nous allons reparler un instant de la classe NgModule :

```
import {NgModule} from '@angular/core'
import {CommonModule} from '@angular/common'
import {GreeterComponent} from './greeter.component'
```

```
@NgModule({
  imports: [CommonModule],
  declarations: [GreeterComponent],
  exports: [GreeterComponent]
})
export class GreeterModule {
}
```

# @NgModule

NgModule nous permet -comme son nom l'indique- d'importer des modules Angular, et donc d'étendre largement les fonctionnalités de notre application (composants, directives, services, etc...).

Deux types principaux de modules :

- les modules déjà installés dans le dossier Node par la CLI ('@angular/..')
- les modules à importer soi-même (packages)

**Attention, une fois installés, les modules doivent être importés par une instance NgModule.**

# @NgModule

Ne pas confondre modules Angular et modules javascript:

- Angular : seront importés et gérés par un NgModule
- Js: gérés par Webpack, ou autre module loader



# @NgModule

Configuration de @NgModule :

```
import {CommonModule} from '@angular/common'  
import {PackageModule} from 'package/module'  
import {ServiceModule} from 'service/module'  
import {MyComponent} from './my/my.component'
```

```
@NgModule({  
  // La plupart des modules importés seront déclarés ici  
  imports: [CommonModule,  
            PackageModule],  
  // Les composants que nous créons sont déclarés ici  
  declarations: [MyComponent],  
  // Lorsque les modules importés contiennent des services  
  // ils sont déclarés ici - nous y reviendrons  
  providers: [GreeterComponent]  
})  
export class AppModule {}
```

# Modules

## TP:

- Importer un module Angular (ex: Angular Material) et organiser les imports sous forme d'un feature Module
- Importer une librairie Javascript (ex: LocalForage)

# PIPES

Les transformateurs fournis

Formater une chaîne

Formater des collections

Utiliser un pipe comme un service

TP : Créer ses propres pipes

# Introduction

Les pipes sont des opérateurs permettant d'appliquer une transformation à un input :

```
<p>{{ 10.6 | currency:'CAD': 'symbol-narrow' }}</p>  
<!-- retournera '$10.60' -->
```

Les pipes peuvent être utilisés :

- directement dans le template (ci-dessus)
- dans le composant => *necessite l'import du pipe souhaité, ainsi que 'l'injection' du pipe via le constructeur (cf chapitre sur l'Injection de Dépendances)*

# Introduction

Utilisation dans un composant :

```
import { Component } from '@angular/core';
// Importer le pipe
import { CurrencyPipe } from '@angular/common';
@Component({
  selector: 'app-money',
  template: `<p>{{ stringAsCurrency }}</p>`
})
export class MoneyComponent {
  money: number = 1;
  stringAsCurrency: string;
  // injection du pipe
  constructor(currencyPipe: CurrencyPipe) {
    // appel sur le pipe de la méthode transform
    this.stringAsCurrency = currencyPipe.transform(this.mon
  }
}
```

# Introduction

Paramétrisation des pipes :

Un pipe peut accepter des paramètres optionnels, pour affiner l'output.

=> {{ ... | nomDuPipe: paramètre1 : paramètre2 : ... }}

<p>Une date: {{ myDate | date:"MM/dd/yy" }} </p>

# Introduction

Les paramètres passés aux pipes peuvent être des 'template expressions' :

```
<!-- Template -->
<p>Une date : {{ myDate | date:format }}</p>
<button (click)="changeFormat()">Changer le format</button>

// composant
export class ChangeDateFormatComponent {
  myDate = new Date(2018, 1, 31); // 31 janvier 2018
  changeFormat = true; // true == shortDate

  get format() { return this.toggle ?
                  'shortDate' : 'fullDate'; }
  changeFormat() { this.toggle = !this.toggle; }
}
```

# Introduction

Enchaîner les pipes :

Une date enchaînée :

```
{{ myDate | date | uppercase }}
```

Une date enchaînée :

```
{{ myDate | date: format | uppercase }}
```



# Pipes fournis : json

Applique simplement **JSON.stringify()** :

*Nb: json est n'est pas très utilisé en production, mais bien pratique pour le debug.*

Utilisation dans un template :

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

```
// affichera :
```

```
<p>[ { "name": "Margarita" }, { "name": "Quatre fromages" } ]</p>
```

## Utilisation dans un composant :

```
import { Component } from '@angular/core';
import { JsonPipe } from '@angular/common';
@Component({
  selector: 'app-pizza',
  template: `<p>{{ pizzaAsJson }}</p>`
})
export class PizzaComponent {
  pizzas: Array<any> = [{ name: 'Margarita' }, { name:
  pizzasAsJson: string;
  constructor(jsonPipe: JsonPipe) {
    this.pizzasAsJson = jsonPipe.transform(this.pizzas);
  }
}
```

Nb: l'utilisation de pipes dans un composant présentant toujours une architecture similaire, nous ne donnerons en exemple que l'utilisation en template pour les pipes suivants.

# Pipes fournis : slice

Applique **slice()** au sous-ensemble d'une collection (pour en afficher qu'une partie).

=> 2 paramètres : un indice de départ et, éventuellement, un indice de fin.

```
<!-- Sur un array -->  
<p>{{ pizzas | slice:1:3 }}</p>
```

```
<!-- Sur une chaîne de caractères -->  
<p>{{ 'Margarita' | slice:0:5 }}</p>
```

# Pipes fournis : text format

Les pipes **uppercase**, **lowercase** et **titlecase** appliquent différentes transformations à des chaînes de caractères :

```
<p>{{ 'Quatre fromages' | uppercase }}</p>  
<!-- affichera 'QUATRE FROMAGES' -->
```

```
<p>{{ 'Quatre fromages' | lowercase }}</p>  
<!-- affichera 'quatre fromages' -->
```

```
<p>{{ 'Quatre fromages' | titlecase }}</p>  
<!-- affichera 'Quatre Fromages' -->
```

# Pipes fournis : number

```
<p>{{ 12345 }}</p>  
<!-- affichera '12345' -->
```

```
<p>{{ 12345 | number }}</p>  
<!-- affichera '12,345' -->
```

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

```
<p>{{ 12345 | number:'.2' }}</p>  
<!-- affichera '12,345.00' -->
```

```
<p>{{ 12345.13 | number:'.1-1' }}</p>  
<!-- affichera '12,345.1' -->
```

# Pipes fournis : percent

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

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

# Pipes fournis : currency

```
<p>{{ 10.6 | currency:'CAD' }}</p>  
<!-- affichera 'CA$10.60' -->
```

```
<p>{{ 10.6 | currency:'CAD':'symbol-narrow' }}</p>  
<!-- affichera '$10.60' -->
```

```
<p>{{ 10.6 | currency:'EUR':'code':'.3' }}</p>  
<!-- affichera 'EUR10.600' -->
```

# Pipes fournis : date

Nb: ce pipe est très similaire à la librairie Moment.js

```
<p>{{ myDate | date:'dd/MM/yyyy' }}</p>  
<!-- affichera '16/07/1986' -->
```

```
<p>{{ myDate | date:'longDate' }}</p>  
<!-- affichera 'July 16, 1986' -->
```

```
<p>{{ myDate | date:'HH:mm' }}</p>  
<!-- affichera '15:30' -->
```

```
<p>{{ myDate | date:'shortTime' }}</p>  
<!-- affichera '3:30 PM' -->
```

# Pipes fournis : async

Permet d'afficher des données obtenues de manière asynchrone en utilisant **PromisePipe** ou **ObservablePipe**.

*=> Nous reviendrons sur ce pipe dans le chapitre concernant les Observables dans RxJS.*

Un pipe async retourne une chaîne de caractères vide jusqu'à ce que les données deviennent disponibles (ex: promise résolue, dans le cas d'une promise), puis :

- **retourne** la valeur obtenue
- déclenche un cycle de **détection de changement** une fois la donnée obtenue (cf Observables)



# Pipes fournis : async

Exemple utilisant une promesse :

```
import { Component } from '@angular/core';
@Component({
  selector: 'ns-greeting',
  template: `<div>{{ asyncGreeting | async }}</div>`
})
export class GreetingComponent {
  asyncGreeting = new Promise(resolve => {
    // promise resolve après 1 seconde
    window.setTimeout(() => resolve('hello'), 1000);
  });
}
```

# Custom pipes

Pour créer un custom pipe, il suffit de :

- créer une nouvelle classe
- y ajouter le décorateur **@Pipe({name: "..", ..})**
- y implémenter l'interface **PipeTransform**, ce qui nous amène à écrire une méthode **transform()**

```
import { PipeTransform, Pipe } from '@angular/core';
@Pipe({ name: 'newPipe' })
export class NewPipe implements PipeTransform {
  transform(value, args) {
    return ...;
  }
}
```

# Custom pipes

Il faut ensuite rendre ce pipe disponible dans l'application en le déclarant dans **ngModule** :

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

# Custom pipes - exemple

Pipe affichant le temps écoulé depuis une date à l'aide de la librairie Moment.js :

=> *Nous utiliserons la fonction **fromNow()** de Moment.js*

1. installer Moment.js avec NPM :

```
npm install moment
```

Nb: Les types nécessaires pour TypeScript sont déjà inclus dans la dépendance NPM, nous pouvons donc déjà profiter d'un typage prédéfini.

# Custom pipes - exemple

## 2. Créer le pipe :

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

export class FromNowPipe implements PipeTransform {
  transform(value, args) {
    return moment(value).fromNow();
  }
}
```

## 3. Le rendre disponible (en le déclarant dans NgModule)

# Pipes

## TP:

- Utiliser un Pipe fourni
- Pipe Async avec une Promesse
- Custom Pipe (ex: limiter le nombre de caractères d'un texte)

# Programme - Jour 2

## **SERVICES**

Les services fournis

Injection de service

TP : Injecter les services fournis par Angular

## **INJECTION DE DÉPENDANCES (IOC)**

Principes

Configurer son application

L'injection de dépendances : type-based et hiérarchique

Différents types de providers

TP : Créer ses propres services

# **ROUTER**

RouterModule: Configuration des routes et URLs

Définitions des routes, liens et redirection, paramètres

Hiérarchies de routes

Vues imbriquées

Cycle de vie (Routing lifecycle)

TP : Transformer une application Web en Single Page  
Application



# Dépendances & Services

# Injection de dépendances

## Dépendance :

- un composant C consomme une fonction F
  - F est déclarée dans un service S
- C dépend de S => S est une **dépendance** de C

## 2 possibilités :

- C crée une instance de S
- le framework crée une instance de S, qu'il 'injecte' dans C

= **injection de dépendance**

# Exemple

Une dépendance est simplement une classe que l'on va 'injecter' dans une autre classe (généralement service --> composant) :

```
export class ApiService {  
  get(path) {  
    // todo: appeler le backend  
  }  
}
```

# Exemple

Signaler l'injection : le décorateur @Injectable

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

@Injectable()
export class ApiService {
  get(path) {
    // todo: appeler le backend
    // en utilisant la dépendance Http
  }
}
```

# Injection de dépendances

Pour injecter notre dépendance, on a ensuite besoin :

- d'une façon d'enregistrer la dépendance, pour la rendre disponible à l'injection dans d'autres composants/services.
- d'une façon de la déclarer dans nos composants ou services.

# Injection de dépendances

## Enregistrer une dépendance :

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

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

  constructor() { }
}
```

# Injection de dépendances

## Déclarer la dépendance :

```
...  
import { ApiService } from '../api-service';  
  
@Component({ ... })  
export class Component {  
  
    constructor( public apiService : ApiService ) {}  
  
    useService() {  
        this.apiService.....subscribe(  
            data => // use data  
        )  
    }  
}
```

# Les services

Les services correspondent à la façon la plus utilitaire d'injecter des dépendances.

## Principe:

Quand plusieurs composants ont besoin de faire la même chose :

- **factoriser le code** correspondant dans un service
- **injecter** dans les composants



# Les services

**Créer un service :**

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

@Injectable()
export class LoginService {

    constructor() {}

    doSomething() {}
}
```

# Les services

## Injecter un service :

```
import {Component} from '@angular/core';
import {LoginService} from 'login.service';

@Component({
  selector: 'my-component',
  providers: [LoginService],
  template: require('./my.component.html')
})
export class MyComponent {

  constructor(private loginService: LoginService) { }

  ngOnInit() {
```

# Routeage

# Introduction

But: associer une URL à un état de l'application (meilleure UX)

=> **routeur** : chaque framework a le sien

En Angular, il s'agit du module **RouterModule**.

# RouterModule

*Nb: optionnel => à inclure dans NgModule*

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
// Configuration, cf prochaine slide
import { ROUTES } from './app.routes';
...
```

```
@NgModule({
  imports: [BrowserModule, RouterModule.forRoot(ROUTES)],
  declarations: [...],
```

# RouterModule

Configuration du module :

*Nb: cela peut se faire dans un fichier dédié, généralement nommé app.routes.ts.*

```
import { Routes } from '@angular/router';
import { HomeComponent } from '../home/home.component';
import { OtherComponent } from '../other/other.component';
export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  { path: 'other', component: OtherComponent }
]
```

# RouterModule

Inclure le composant 'routé' dans le template :

```
<router-outlet></router-outlet>.
```

# RouterModule

Exemple :

```
<header>
  <nav>...</nav>
</header>
<main>
  <router-outlet></router-outlet>
  <!-- le template du composant 'routé' sera
        inclu ici -->
</main>
<footer>fait avec &lt;3 par ...</footer>
```



# Navigation :

Naviguer entre différents composants ?

En effet, avec des liens "classiques" :

click --> reload page --> relance toute l'app (SPA)

=> utiliser une directive particulière : **routerLink**.

# RouterLink :

RouterLink - argument :

- le chemin (string)
- le chemin + paramètres (array<string>)

Nb: importer cette directive ?

# RouterLink :

## Exemples :

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

*RouterLinkActive - ajouter une classe CSS lorsque le lien pointe sur la route courante :*

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

# navigate() :

Naviguer depuis le composant :

- injecter le **service Router** (*cf partie sur la DI*)
- utiliser sa méthode **navigate()**

```
export class navigationComponent {  
    // Injection d'une instance du Router  
    constructor(private router: Router) {  
    }  
    saveAndMoveBackToHome() {  
        // Route :  
        this.router.navigate([ ' ' ]);  
    }  
}
```

# Urls dynamiques :

- définir une route dans la configuration avec des paramètres dynamiques ( " ../:paramDyn/.. ")

```
export const routes: Routes = [  
  { path: '', component: HomeComponent },  
  { path: 'newsFeed', component: newsFeedComponent },  
  { path: 'profiles/:profileId', component: ProfileComponent },  
];
```

- définir des liens dynamiques ("[routerLink] = "[paramStatic', paramDyn, ...]"

```
<a href="" [routerLink]="['/profiles', profile.id]">  
  Voir profil</a>
```

# Introduction à RxJS

# Design pattern

Observable :

Fonction qui associe une source de données à un observateur + retourne un moyen d'annuler cette liaison.

Observeur:

Objet ayant une méthode `next()`, et optionnellement `complete()` et `error()`.

# Design pattern

```
function monObservable(observer) {  
  let array = [1, 2 , 3 , 4]  
  array.forEach(  
    (el) => observer.next(el)  
  )  
  observer.complete()  
}
```

```
let observer = {  
  next : (value) => console.log('Nouvelle valeur : ' + val  
  complete : () => console.log('Terminé.')  
}
```



# La librairie Rxjs

**Rxjs** permet de créer des observables :

- "safe"
- grâce à des outils

```
const my_observable = new Observable(  
  (observable) => observable.next(42))
```

```
const my_observable2 = Observable.of(42);
```

***of** est la méthode la plus simple et permet de créer un observable n'envoyant qu'une seule valeur.*

# Rxjs - Subscribe

Méthode subscribe() permet d'exécuter l'observable :

```
const my_observable = Observable.of(42);
```

```
my_observable.subscribe(  
  (value) => console.log(value),  
  (error) => console.log(error),  
  () => console.log('terminé')  
);
```

```
my_observable.complete();
```

# Rxjs - Subscribe

Méthode subscribe() permet d'exécuter l'observable :

```
const my_observable = Observable.of(42);
```

```
my_observable.subscribe(  
  (value) => console.log(value),  
  (error) => console.log(error),  
  () => console.log('terminé')  
);
```

```
my_observable.complete();
```

# Rxjs - Pipe

Méthode pipe() (**Angular 6+**) permet de transformer l'output, en revoyant un nouvel observable:

```
const my_observable = Observable.of(42);
```

```
my_observable  
  .pipe(  
    map( val => val - 10))  
  .subscribe(  
    (value) => console.log(value));
```

# Un cas concret

Angular nous propose justement de nombreux services exploitant à fond la programmation réactive, tel que [HTTP](#).

Les différentes méthodes du service **http** retournent des **Observable<Response>**. Notez que le type des variables émises par l'observable est précisé entre chevrons, les observables sont en effet [génériques](#).

**Echanger avec le  
serveur**

# Le module HttpClient

## Attention :

On utilise le nouveau **HttpClientModule** introduit avec Angular 4.3 dans le package **@angular/common/http**, qui est une réécriture complète du **HttpModule** qui existait jusqu'à alors. Ce chapitre ne parle pas de l'ancien **HttpModule** du package **@angular/http** qui était utilisé précédemment.

# Le module HttpClient

**Nb :**

Traditionnellement, on utilise HTTP, mais il y a des alternatives :

- WebSockets
- bibliothèques HTTP, comme l'API fetch, pour le moment disponible sous forme de polyfill, mais qui devrait devenir standard dans les navigateurs.



# HttpClient - implémentation

L'implémentation du module est assez simple :

1. Le déclarer dans app.module.ts
2. "L'injecter" partout où on en a besoin

```
@Component({  
  selector: 'xxx',  
  template: '<h1>xxx</h1>'  
})  
export class RacesComponent {  
  constructor(private http: HttpClient) {  
  }  
}
```

# HttpClient - implémentation

Il propose plusieurs méthodes, correspondant aux verbes HTTP communs :

get • post • put • delete • patch • head • jsonp

Une requête est effectuée de la façon suivante :

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

Cela retourne un Observable => on doit donc s'y abonner pour obtenir la réponse.

# HttpClient - implémentation

Abonnement :

```
http.get(`${baseUrl}/api/foo/bar`)  
    .subscribe((response: Foobar) => { console.log(respo
```

Nb : Le corps de la réponse, qui est la partie la plus intéressante, est directement émis par l'Observable. On peut néanmoins accéder à la réponse HTTP complète :

```
http.get(`${baseUrl}/api/foo/bar`, { observe: 'response' })  
    .subscribe((response: HttpResponse<Foobar>) => {  
        console.log(response.status); // logs 200  
        console.log(response.headers.keys()); // logs []  
    });
```

# HttpClient - implémentation

Envoyer des données est aussi trivial. Il suffit d'appeler la méthode `post()`, avec l'URL et l'objet à poster :

```
http.post(`${baseUrl}/api/foo/bar`, FooBar)  
    .subscribe((response: Foobar) => { console.log(respo
```

# Les formulaires

# Les formulaires en Angular

Angular propose 2 types de formulaires:

- "template driven": formulaires simples, peu de validation.
- "reactive forms": représentation du formulaire dans le contrôleur

=> plus verbeux, mais aussi plus puissant (validation custom)

# Les formulaires en Angular

Fonctionnement :

Dans les 2 cas, Angular crée une représentation de notre champ sous la forme d'un arbre d'objets de la classe **FormControl**.

- "template driven" : objet créé en interne
- "reactive form" : objet créé manuellement dans le composant

# Template driven

Il suffit d'ajouter les directives ngModel, qui crée le FormControl, et le <form> crée automatiquement le FormGroup.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <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>
```



# Reactive forms :

On crée 'manuellement' le formulaire :

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

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

export class RegisterFormComponent {

  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('');
    this.passwordCtrl = fb.control('');
```

# Reactive forms :

Que l'on lie ensuite au template :

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label>
    <input formControlName="username">
  </div>
  <div>
    <label>Password</label>
    <input type="password" formControlName="password">
  </div>
  <button type="submit">Register</button>
</form>
```

# Du style

=> ajout / retrait automatique classes CSS selon l'état du formula

*Exemple: ng-invalid si un de ses validateurs échoue*

```
input.ng-invalid {  
  border: 3px red solid;  
}
```

# Attributs

Un FormControl a plusieurs attributs :

- valid : si le champ est valide, au regard des contraintes et des validations qui lui sont appliquées.
- invalid : si le champ est invalide, au regard des contraintes et des validations qui lui sont appliquées.
- errors : un objet contenant les erreurs du champ.

etc....

+ quelques méthodes comme `hasError()` pour savoir si le contrôle a une erreur donnée.

# Les formulaires en Angular

On peut donc écrire :

```
const password = new FormControl( 'abc' );  
console.log(password.dirty);  
console.log(password.value);  
console.log(password.hasError( 'required' ));
```

Ces contrôles peuvent être regroupés dans un FormGroup ("groupe de formulaire") pour constituer une partie du formulaire qui a des règles de validation communes. Un formulaire lui-même est un groupe de contrôle.

# Les formulaires en Angular

```
const form = new FormGroup({  
  username: new FormControl('Cédric'),  
  password: new FormControl() });  
console.log(form.dirty);
```

Un FormGroup a les mêmes propriétés qu'un FormControl, avec quelques différences :

- valid : si tous les champs sont valides, alors le groupe est valide.
- invalid : si l'un des champs est invalide, alors le groupe est invalide.
- Etc ...

# Validation

Angular nous permet de rajouter des paramètres de validation facilement.

Reactive form :

```
constructor(fb: FormBuilder) {  
  this.userForm = fb.group({  
    username: fb.control('', [Validators.required,  
                               Validators.minLength(3)]),  
    password: fb.control('', Validators.required)  
  });  
};
```

# Validation

Template driven :

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)"
  #userForm="ngForm">
  <div>
    <label>Username</label><input name="username"
      ngModel required minlength="3">
  </div>
  <div>
    <label>Password</label><input type="password"
      name="password" ngModel required>
  </div>
  <button type="submit">Register</button>
</form>
```



# Validation

Quelques validateurs sont fournis par le framework :

- `Validators.required` pour vérifier qu'un contrôle n'est pas vide ;
- `Validators.minLength(n)` pour s'assurer que la valeur entrée a au moins `n` caractères ;
- `Validators.maxLength(n)` pour s'assurer que la valeur entrée a au plus `n` caractères ;
- `Validators.email()` (disponible depuis la version 4.0) pour s'assurer que la valeur entrée est une adresse en
- `Validators.min(n)` (disponible depuis la version 4.2) pour s'assurer que la valeur entrée vaut au moins `n` ;
- `Validators.max(n)` (disponible depuis la version 4.2) pour s'assurer que la valeur entrée vaut au plus `n` ;
- `Validators.pattern(p)` pour s'assurer que la valeur entrée correspond à l'expression régulière `p` définie.

# Erreurs et soumission

Reactive form:

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label>
    <input formControlName="username">
  </div>
  <div>
    <label>Password</label>
    <input type="password" formControlName="password">
  </div>
  <button type="submit" [disabled]="userForm.invalid">
    Register
  </button>
</form>
```