



# Angular





# Présentation du cours



4 jours pour devenir ceinture noire sur Angular

- Concepts théoriques avancés
- Cas d'étude & démonstration
- Bonnes pratiques et astuces

- 40 % de théorie (slides)
- 60 % de pratique (TPs)



# Objectifs pédagogiques



- Organiser, modulariser et tester ses développements JavaScript
- Maîtriser les fondamentaux du Framework Angular 2
- Créer rapidement des applications Web complexes
- Savoir intégrer les tests unitaires au développement





# Pré requis et outils



- **Pré-requis :**
  - Connaissance en JavaScript, HTML.
  - Connaissance en développement avec un langage orienté objet.
  - Bonnes connaissances des technologies du Web et des outils modernes de développement Front-End.
- **Outils :**
  - Npm (Node.js)
  - Eclipse Neon+ Angular IDE/IntelliJ Idea ...
  - Tomcat+7 , JDK+7





Qui suis-je ?



- TODO





**Qui êtes vous ?  
Vos attentes ?**





# Plan du cours

## INTRODUCTION

Historique d'Angular

Développement JavaScript : rappels

ES6

TypeScript

Travaux pratiques

## La philosophie d'Angular : Les composants

### L'utilitaire ng ou @angular/cli

## Classifications des composants applicatifs

Templates

Définition de composants

Les hooks

Les pipes

Les directives



## Gestion des formulaires, "Routing" et requête HTTP

L'injection des dépendances

Les services

Les observables

Le service HTTP

FormControl et FormGroup.

TDF versus DDF : Template Driven Form et Data Driven Form

Le routage

## Migrer d'AngularJS 1.x à Angular

## Tests unitaires, Build et Bonnes pratiques et outils





# Organisation & suivi du cours



## Déroulement du cours

N'hésitez pas à interrompre ou à intervenir

Si un chapitre ne vous intéresse pas, on peut le sauter







# Introduction



- Angular est un Framework JavaScript open source sous la licence MIT.
- Il assure la création des applications web dynamiques monopages, permettant de développer ses propres balises et attributs HTML.





# Historique



## Origines du projet

- Angular a été créé par **Misko Hevery** et **Adam Abrons** en 2009 dans les locaux de Google par **Brat Tech LLC**.
- Initialement appelé **GetAngular**.
- Le framework a ensuite été rendu disponible en Open Source.
- Google a par la suite repris le code source pour le développer.
- Sept 2014. Annonce de la sortie d'Angular 2 (un peu prématurée).
- Mai 2016. Sortie de la première Release Candidate : **Angular 2**.
- Mars 2017 : **Angular 4**





# Qu'est-ce qu'Angular ?

Angular est un framework JavaScript pour créer des applications monopages (SPA), web et mobiles.

Quels types d'applications peut-on développer ?

- De petits widgets interactifs pour un site web existant (moteur de recherche, module de réservation). Exemple :

<https://www.virginamerica.com/>

- Site web complet. Exemple : <https://weather.com/>

- Application mobile. Exemple : <https://posse.com/>

- Même un **pokedex** : <https://ng-pokedex.firebaseio.com/>



Plus de références : <http://builtwithangular2.com/>





# La communauté



## La communauté d'Angular est très active.

- Programmeurs
- Présente sur Github et Stackoverflow <https://github.com/angular/>
- Développeurs web (professionnels ou non)
- Conférences organisées partout dans le monde
  - »AngularU à San Francisco
  - »AngularConnect à Londres au Royaume-Uni
  - »Ng-conf à Salt Lake City
  - »Ng-Vegas à Las Vegas
  - »Ng-Europe à Paris <https://ngeurope.org/>
  - »Et plusieurs autres
  - »... ngMorocco





# Sites web utilisant AngularJS



## Question :D

Parmi les sites suivants quels sont les sites qui n'utilisent pas Angular ?

- <https://genesui.com/demo/real/bootstrap4-static/>
- <https://ng2snake.herokuapp.com/>
- <https://www.nasa.gov/>
- <https://goodfil.ms/>
- <http://ng2piano.azurewebsites.net/>





# Introduction



- Précision préliminaire pour les développeurs JavaScript "old school": avec Angular, il n'y a, en général, pas de manipulation directe du DOM, si, si ...!!
- Avec jQuery, Prototype et autres librairies JavaScript, on doit presque toujours sélectionner un élément (via l'API DOM) pour pouvoir l'utiliser.
- Avec AngularJS on peut ajouter, supprimer et modifier la page HTML sans faire aucun appel au DOM: plus besoin de **\$()**, **getElementById()**, ...





# Les applications monopages (SPA)

- **Angular** permet de développer des applications Web de type SPA.
- Une **SPA** (Single Page Application) est une application web accessible via une page web unique.
- Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée et d'améliorer ainsi l'expérience utilisateur (meilleure fluidité).





# Les applications monopages (SPA)

- La différence entre une **SPA** et un site web classique réside dans leur structure et dans la relation qu'ils établissent entre le navigateur et le serveur:
  - Une SPA est donc composée d'une seule page.
  - Le rôle du browser (front-end) est beaucoup plus important : toute la logique applicative y est déportée.
  - Le serveur (back-end) est "seulement" responsable de la fourniture des ressources à l'application et surtout de l'exposition des données.







# SPA: Pourquoi on en parle ?

- Les frameworks **JS/TS** comme **Angular** participent à la popularité des **SPA**.
- Les SPA s'appuyant sur de tels frameworks ont en général comme avantage d'être:
  - Testables (unitairement et fonctionnellement)
  - Fluides (pas de rechargement d'url etc)
  - Bien organisées
  - Maintenables et évolutives
  - ...





## Pourquoi Angular ?



- Angular 2/4 est plus facile à apprendre que AngularJS, enfin si on connaît TypeScript
- Angular 2 a été réécrit en TypeScript
- Performance et mobile : Angular 2 a été conçu initialement pour le mobile
- Maintenabilité
- Testabilité
- Google ...





# Principales caractéristiques d'Angular 2

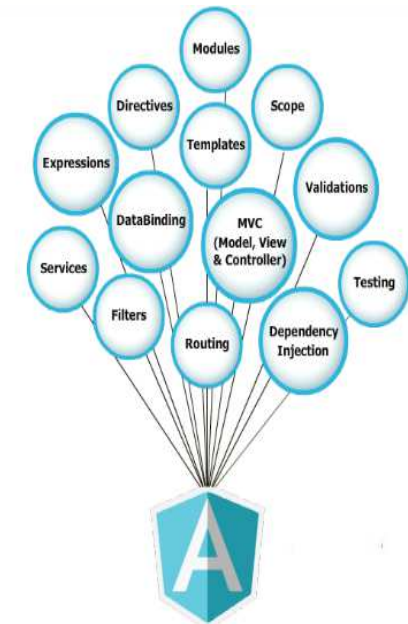
Plusieurs langages supportés(1). ES5, ES6, TypeScript...

**Complet.** Inclut toutes les briques nécessaires à la création d'une appli professionnelle. Routeur, requêtage HTTP, gestion des formulaires, internationalisation...

**Modulaire.** Le framework lui-même est découpé en sous-paquets correspondant aux grandes aires fonctionnelles (core, router, http...). Vos applis doivent être organisées en composants et en modules (1 module = 1 fichier).

**Rapide.** D'après les benchmarks, Angular 2 est aujourd'hui 5 fois plus rapide que la version 1.

**Tout est composant.** Composant = brique de base de toute appli Angular 2.





## ES6 et TypeScript



Le langage appelé « **JavaScript** » est formellement reconnu avec le nom « **EcmaScript** ».

La nouvelle version est appelée **ES6** (**EcmaScript 6** ou **EcmaScript 2015**) et étend JavaScript pour fournir des supers fonctionnalités .

ES6 n'est pas supporté aujourd'hui par la majorité des navigateurs, il a besoin d'être « **Transpilé** » vers la version **ES5**.

On peut choisir parmi plusieurs transpileurs, on utilisera celui du TypeScript , le langage choisi par l'équipe Angular pour écrire Angular.





# ES6 et TypeScript : Historique



1995 : Netscape crée le langage dynamique JavaScript pour faciliter le développement côté navigateur.

1995 : Netscape rend possible l'implémentation d'applications côté serveur en JavaScript avec "Netscape Enterprise Server".

1997 : Création du standard "cross-browser" ECMAScript.

2009 : Sortie de NodeJS par **Ryan Lienhart Dahl**

2015 : Finalisation du standard ECMAScript 6.

## **En parallèle Microsoft travaillait sur TypeScript :**

Octobre 2012 : Version non officielle de TypeScript

Avril 2014 : Release 1.0

Mars 2015 : 1.5-alpha

23/09/2016 : Release 2.0.5





## ES6



ES6 apportent plusieurs fonctionnalités manquantes à savoir :

- Les classes
- Les fonctions fléchées
- Template chaîne de caractère
- L'Héritage
- Les constantes et les variables avec des portées bloqué.
- La destructuration
- Les modules





# Les classes



- **Les classes** sont des nouvelles fonctionnalités dans ES6 pour décrire le comportement des objets et fait en sorte qu'ES6 soit semblable à un langage Orienté objet traditionnel.

```
class Hamburger {  
  constructor() {  
    // This is the constructor.  
  }  
  listToppings() {  
    // This is a method.  
  }  
}
```

- En JavaScript, le mot clé « **this** » peut être utilisé pour référencer l'instance de l'objet courant. Il est tout de même possible de modifier cette référence en fonction du contexte appelant.

```
Function.prototype.call(object [,arg, ...])  
Function.prototype.bind(object [,arg, ...])  
Function.prototype.apply(object [,argsArray])
```





# L'objet



- Un objet est une instance de classe créée en utilisant le mot clé **new**.

```
let burger = new Hamburger();  
burger.listToppings();
```

- Dans cet exemple, si le mot clé **this** est utilisé à l'intérieur de la classe **Hamburger** il référence l'objet **burger**.







# this dans les classes



- En général, si le mot clé `this` est utilisé dans une classe, il référence l'instance de cette classe.

```
class Toppings {  
  ...  
  
  formatToppings() { /* implementation details */ }  
  
  list() {  
    return this.formatToppings(this.toppings);  
  }  
}
```

- Il existe deux types d'invocation

- Invocation de méthode.
- Invocation de fonction.

```
someObject.someMethod();
```

```
someFunction();
```

**"use strict";**

non

oui

**this** référence le contexte souhaité

**this**=undefined





## this : exemple 1

- Considérons l'exemple suivant :
- La plupart des navigateurs généreront une exception parce que la méthode log utilise le mot clé this et s'attends à ce que this référence console. Cette référence est perdue dès que la méthode a été détaché de la console.
- Il est possible de fixer explicitement ce problème en utilisant par exemple la méthode **bind**.

```
var log = console.log;  
log('Hello');
```

```
var log = console.log.bind(console);  
log('Hello');
```





# Les classes

➤ Exercice :

Créer un fichier HTML , **fonctionArrow.html**

Créer un fichier JavaScript **farrow.js** et

importer le dans le fichier html.

Exécuter le code ci-contre.

Ecrire la classe équivalente au code

Javascript ci-contre.



```
/* Avant */
function Animal(friends) {
  this.friends = friends;
  this.hello = function(friend) {
    console.log("hello " + friend);
  }
  this.helloAll = function() {
    this.friends.forEach(function(friend) {
      this.hello(friend); /* error */
    });
  }
}

var wolf = new Animal(["rox", "rookie"]);
wolf.helloAll();
```





## this : exemple 2



- Considérons ce deuxième exemple:

```
class ServerRequest {  
  notify() {  
    ...  
  }  
  fetch() {  
    getFromServer(function callback(err, data) {  
      this.notify(); // this is not going to work  
    });  
  }  
}
```

- Il s'agit d'un autre cas de confusion d'utilisation du mot clé **this**.
- **this** est appelé dans une fonction déclarée à l'intérieur d'une autre fonction.
- Dans ce cas, **this** ne pointe pas sur l'instance de la classe et la méthode **notify** ne peut être appelé.
- Solution ES6 : **Les fonctions fléchées**





# Les fonctions fléchées



- ES6 propose une nouvelle syntaxe qui permet de gérer le mot clé **this** convenablement.
- Cette syntaxe permet de définir des fonctions appelées «**Fléchées**» et facilite aussi l'écriture des fonctions.
- Considérons l'exemple suivant :

```
items.forEach(function(x) {  
  console.log(x);  
  incrementedItems.push(x+1);  
});
```

- Avec l'écriture fléchées:

```
items.forEach((x) => {  
  console.log(x);  
  incrementedItems.push(x+1);  
});
```





# Les fonctions fléchées



➤ Plus simple encore :

```
incrementedItems = items.map(function (x) {  
  return x+1;  
});
```



```
incrementedItems = items.map((x) => x+1);
```

!! Une seule grande différence ici à retenir : les fonctions fléchées n'utilise pas une variable

**this** local. La portée du mot clé est celle du contexte parent externe,,,





# Les fonctions fléchées : Exemple

- Considérons l'exemple suivant :

```
class Toppings {  
  constructor(toppings) {  
    this.toppings = Array.isArray(toppings) ? toppings : [];  
  }  
  outputList() {  
    this.toppings.forEach(function(topping, i) {  
      console.log(topping, i + '/' + this.toppings.length); // `this` will be undefined  
    });  
  }  
}  
  
var myToppings = new Toppings(['cheese', 'lettuce']);
```

- <http://jsbin.com/qakigoqulo/edit?js,console>
- Une erreur est générée parce que le mot clé `this` est utilisé dans la fonction anonymes,





# Les fonctions fléchées : Exemple

- Avec la syntaxe des fonctions fléchées :

```
class Toppings {  
  constructor(toppings) {  
    this.toppings = Array.isArray(toppings) ? toppings : [];  
  }  
  outputList() {  
    this.toppings.forEach((topping, i) => {  
      console.log(topping, i + '/' + this.toppings.length) // `this` works!  
    });  
  }  
}  
  
var myToppings = new Toppings(['cheese', 'lettuce']);  
  
myToppings.outputList();
```

- <http://jsbin.com/tulikutife/edit?js,console>
- Ça marche !!







# Les fonctions fléchées



## ➤ Exercice :

Si ce n'est pas encore fait, modifier la classe Animal précédente en utilisant les fonction fléchées et résoudre le problème constaté.

Créer un objet littéral contenant deux fonctions :

Une fonction traditional

Une fonction arrow

Chaque fonction doit afficher le contenu suivant :

➤ **`console.log('traditionalFunc this === o?', this === o);`**

avec o: l'objet littéral créé.

Quel constat faire ?





# ES6 et les chaines de caractères

- En JavaScript : Si une expression est entourée par " ou ' , il s'agit d'une chaîne de caractères et ne peut être écrite que dans une seule ligne de code.
- Il n'est pas possible d'insérer des données dans des chaînes de caractères sans passer par le mécanisme de concaténation.

```
var name = 'Sam';  
var age = 42;  
  
console.log('hello my name is ' + name + ' I am ' + age + ' years old');
```

- ES6 introduit une nouvelle syntaxe pour insérer des données dans des chaînes de caractères,
- Il s'agit de l'utilisation des backticks « ` » (AltGr+7 )

```
var name = 'Sam';  
var age = 42;  
  
console.log(`hello my name is ${name}, and I am ${age} years old`);
```





# ES6 et l'héritage par l'exemple

```
// Base Class : ES6
class Bird {
  constructor(weight, height) {
    this.weight = weight;
    this.height = height;
  }

  walk() {
    console.log('walk!');
  }
}

// Subclass
class Penguin extends Bird {
  constructor(weight, height) {
    super(weight, height);
  }

  swim() {
    console.log('swim!');
  }
}

// Penguin object
let penguin = new Penguin(...);
penguin.walk(); //walk!
penguin.swim(); //swim!
```

```
// JavaScript classical inheritance.

// Bird constructor
function Bird(weight, height) {
  this.weight = weight;
  this.height = height;
}

// Add method to Bird prototype.
Bird.prototype.walk = function() {
  console.log("walk!");
};

// Penguin constructor.
function Penguin(weight, height) {
  Bird.call(this, weight, height);
}

// Prototypal inheritance (Penguin is-a Bird).
Penguin.prototype = Object.create( Bird.prototype );
Penguin.prototype.constructor = Penguin;

// Add method to Penguin prototype.
Penguin.prototype.swim = function() {
  console.log("swim!");
};

// Create a Penguin object.
let penguin = new Penguin(50,10);

// Calls method on Bird, since it's not defined by Penguin.
penguin.walk(); // walk!

// Calls method on Penguin.
penguin.swim(); // swim!
```





# ES6 et la délégation



```
// ES6
class Bird {
  constructor(weight, height) {
    this.weight = weight;
    this.height = height;
  }
  walk() {
    console.log('walk!');
  }
}

class Penguin {
  constructor(bird) {
    this.bird = bird;
  }
  walk() {
    this.bird.walk();
  }
  swim() {
    console.log('swim!');
  }
}

const bird = new Bird(...);
const penguin = new Penguin(bird);
penguin.walk(); //walk!
penguin.swim(); //swim!
```





# ES6 et la portée des variables

- ES6 apporte la notion de variable avec **portée bloquée**.
- Ceci est plus familier avec les autres langages de programmation connus,
- Analysons cet exemple :

```
var five = 5;
var threeAlso = three; // error

function scope1() {
  var three = 3;
  var fiveAlso = five; // == 5
  var sevenAlso = seven; // error
}

function scope2() {
  var seven = 7;
  var fiveAlso = five; // == 5
  var threeAlso = three; // error
}
```

- Les variables définies avec le mot clé **var** ont une portée limitée au niveau de la fonction





# ES6 et la portée des variables

- ES6 apporte la notion de variable avec **portée bloque**.
- En plus du mot clé var, ES6 introduit deux autres mots clés : **let** et **const**.
- « **let** » agit comme le mot clé var (écriture et lecture) avec une portée bloque délimité par {},
- « **const** » agit comme le mot clé let **sans** la possibilité **d'écrire** dans la variable utilisée.

## let

```
var i;
for (i = 0; i < 10; i += 1) {
  var j = i;
  let k = i;
}
console.log(j); // 9
console.log(k); // undefined
```

## const

```
const myName = 'pat';
let yourName = 'jo';

yourName = 'sam'; // assigns
myName = 'jan';   // error
```





# ES6 et l'opérateur de décomposition

- La syntaxe de décomposition permet de développer une expression lorsque plusieurs arguments ou plusieurs éléments sont nécessaires (respectivement pour les appels de fonctions et les littéraux de tableaux)

## Quelques exemples :

```
const add = (a, b) => a + b;  
let args = [3, 5];  
add(...args); // same as `add(args[0], args[1])`, or `add.apply(null, args)`
```

```
let cde = ['c', 'd', 'e'];  
let scale = ['a', 'b', ...cde, 'f', 'g']; // ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
let mapABC = { a: 5, b: 6, c: 3};  
let mapABCD = { ...mapABC, d: 7}; // { a: 5, b: 6, c: 3, d: 7 }
```







# ES6 et l'opérateur de décomposition

- La syntaxe de décomposition permet de développer une expression lorsque plusieurs arguments ou plusieurs éléments sont nécessaires (respectivement pour les appels de fonctions et les littéraux de tableaux)

## Quelque exemples :

```
function addSimple(a, b) {  
  return a + b;  
}  
  
function add(...numbers) {  
  return numbers[0] + numbers[1];  
}  
  
addSimple(3, 2); // 5  
add(3, 2);      // 5
```

```
function print(a, b, c, ...more) {  
  console.log(more[0]);  
  console.log(arguments[0]);  
}  
  
print(1, 2, 3, 4, 5);  
// 4  
// 1
```







# ES6 et l'opérateur de décomposition

- La syntaxe de décomposition permet de développer une expression lorsque plusieurs arguments ou plusieurs éléments sont nécessaires (respectivement pour les appels de fonctions et les littéraux de tableaux)

## Quelques exemples : Affectation par décomposition

```
let foo = ['one', 'two', 'three'];  
  
let one   = foo[0];  
let two   = foo[1];  
let three = foo[2];
```



```
let foo = ['one', 'two', 'three'];  
let [one, two, three] = foo;  
console.log(one); // 'one'
```

```
let myModule = {  
  drawSquare: function drawSquare(length) { /* implementation */ },  
  drawCircle: function drawCircle(radius) { /* implementation */ },  
  drawText: function drawText(text) { /* implementation */ },  
};  
  
let {drawSquare, drawText} = myModule;  
  
drawSquare(5);  
drawText('hello');
```





# ES6 et les modules



- ES6 introduit la notion de module,
- Un module correspond à un **seul** fichier qui permet d'isoler du code source.
- La portée de du code et des données du module est limitée au module . C'est à dire qu'ils ne sont pas accessibles à l'extérieur du fichier.
- Pour rendre visible le code source et les données du module , il suffit de l'exporter en utilisant le mot clé « **export** »

```
// File: circle.js  
  
export const pi = 3.141592;  
  
export const circumference = diameter => diameter * pi;
```





# ES6 les modules



## ❑ Utilisation d'un module coté serveur :

- Pour utiliser un module dans un serveur JavaScript il suffit de l'importer avec le mot clé **import**.

```
import {toto, truc} from "mon-module";
```

## ❑ Utilisation d'un module coté navigateur :

- Les navigateur ne reconnaissent pas les module ES6 et ne sont pas compatible avec la nouvelle syntaxe apportée. Pour utiliser un module coté navigateur il faut impérativement un **chargeur de module**.
- Il existe plusieurs chargeurs de module. On cite par exemple :
  - ✓ RequireJS
  - ✓ SystemJS
  - ✓ Webpack





# ES6 les modules



## ❑ Exemple avec SystemJS :

- Pour charger un module avec SystemJS il faut
  - ✓ Charger le code system.js
  - ✓ Appeler la fonction **System.import** pour charger le module.

```
<script src="/node_module/systemjs/dist/system.js"></script>
<script>
  var promise = System.import('app')
    .then(function() {
      console.log('Loaded!');
    })
    .then(null, function(error) {
      console.error('Failed to load:', error);
    });
</script>
```





# TypeScript



- TypeScript est un langage gratuit et open-source développé et maintenu par Microsoft depuis Octobre 2012.
- Si ES6 est la dernière version de JavaScript, TypeScript est alors une sur-couche (superset) de ES6, c-a-d que toutes les fonctionnalités de ES6 sont encapsulées dans TypeScript, mais pas toutes les fonctionnalités de TypeScript sont encapsulées dans ES6.
- Par conséquent TypeScript doit être « **transpilé** » pour fonctionner dans les navigateurs.

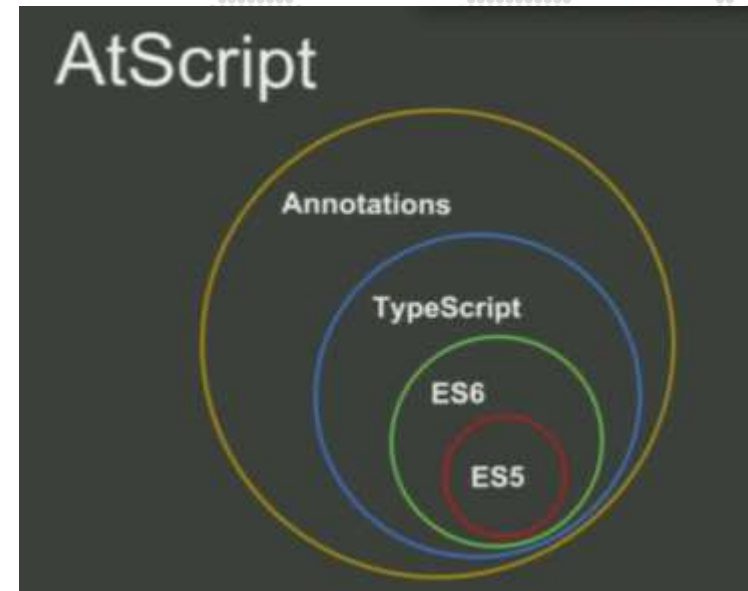
→ **Besoin d'un transpileur.**





# TypeScript et Angular 2

- Initialement, Angular 2 devait utiliser le langage AtScript prévu comme surcouche du TypeScript mais en Mars 2015, Microsoft annonce le support des fonctionnalités AtScript dans la prochaine version de TypeScript (1.5). Depuis, AtScript a été abandonné et Google et Microsoft collaborent au développement de TypeScript.





# Les fonctionnalités de TypeScript

## Le typage statique

➤ TypeScript reprend les types déjà définies par JavaScript :

- Boolean (true/false)
- Number (Integers, floats), infinity et NaN
- Caractères et chaîne de caractères
- [] tableau : number[], boolean []
- {} littéral Object
- Undefined (aucune affectation)

➤ TypeScript ajoute d'autres types à la liste :

- enum : énumérations comme {rouge,bleu,vert}
- any n'importe quel type
- void

```
let isDone: boolean = false;
let height: number = 6;
let name: string = "bob";
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

function showMessage(data: string): void {
    alert(data);
}
showMessage('hello');
```







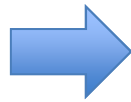
# Les fonctionnalités de TypeScript

## Le typage statique

En plus de ES6, TypeScript propose que toute variable soit **typée statiquement**.

ES6

```
function add(a, b) {  
  return a + b;  
}  
  
add(1, 3);    // 4  
add(1, '3'); // '13'
```



TS

```
function add(a: number, b: number) {  
  return a + b;  
}  
  
add(1, 3);    // 4  
// compiler error before JS is even produced  
add(1, '3'); // '13'
```







# Les fonctionnalités de TypeScript

## Le typage statique

### ➤ Problématique :

- TypeScript exige que le typage est nécessaire. Cependant, **comment** TypeScript s'interface avec les différentes bibliothèques JavaScript qui existent sachant que la plupart ne type pas forcément les variables.

### ➤ Solution :

- TypeScript reconnaît les fichiers dont le nom ressemble à **\*.d.ts**
- Ces fichiers contiennent une liste de définitions des types des bibliothèques.
- Plusieurs communautés travaillent sur ces interfaces de description de typage.
- **DefinitelyTyped** est l'outil de référence pour récupérer ces fichiers.
- <https://github.com/DefinitelyTyped/DefinitelyTyped> (+1500 libs)





# Les fonctionnalités de TypeScript

## Le typage statique

```
class User {  
    _firstName: string;  
  
    constructor(firstName: string) {  
        this._firstName = firstName;  
    }  
}  
  
new User(123);  
  
// error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.
```

- Le typage statique impose plus de rigueur et force le respect des conventions.





# Les fonctionnalités de TypeScript

## Le typage statique

```
class User {  
  
    constructor(firstName: string) {  
        this._firstName = firstName;  
    }  
  
}  
  
// error TS2339: Property '_firstName' does not exist on type 'User'.
```

- Le typage statique fournit une aide précieuse aux IDEs.





# Les fonctionnalités de TypeScript

## Le typage statique

- Question : A quoi sert le ? Dans cette exemple :

```
function logMessage(message: string, isDebug?: boolean) {  
  if (isDebug) {  
    console.log('Debug: ' + message);  
  } else {  
    console.log(message);  
  }  
}  
logMessage('hi'); // 'hi'  
logMessage('test', true); // 'Debug: test'
```





# Les fonctionnalités de TypeScript

## Le mot clé type

- Le mot clé type définit des alias aux types existants

```
type str = string;
let cheese: str = 'gorgonzola';
let cake: str = 10; // Type 'number' is not assignable to type 'string'
```

### type et l'union

```
function admitAge (age: number|string): string {
  return `I am ${age}, alright?!`;
}

admitAge(30); // 'I am 30, alright?!'
admitAge('Forty'); // 'I am Forty, alright?!'
```



```
type Age = number | string;

function admitAge (age: Age): string {
  return `I am ${age}, alright?!`;
}

let myAge: Age = 50;
let yourAge: Age = 'One Hundred';
admitAge(yourAge); // 'I am One Hundred, alright?!'
```





# Les fonctionnalités de TypeScript

## Le mot clé type

type et l'intersection

```
interface Kicker {  
  kick(speed: number): number;  
}  
  
interface Puncher {  
  punch(power: number): number;  
}  
  
// assign intersection type definition to alias KickPuncher  
type KickPuncher = Kicker & Puncher;  
  
function attack (warrior: KickPuncher) {  
  warrior.kick(102);  
  warrior.punch(412);  
  warrior.judoChop(); // Property 'judoChop' does not exist on type 'KickPuncher'  
}
```





# Les fonctionnalités de TypeScript

## Les interfaces

- Une interface peut être vue tout d'abord comme une sorte de contrat minimum que doit respecter une structure de données en termes d'attributs et de méthodes. Cette structure de données peut être un objet {...} ou une classe.

```
interface I1 {  
    a: number;  
}  
  
interface I2 {  
    b: string;  
}
```

- L'héritage multiple est permis dans **TypeScript**

```
class C implements I1, I2 {  
    a: number;  
    b: string;  
  
    constructor(a: number, b: string) {  
        this.a = a;  
        this.b = b;  
    }  
}
```

```
var c = new C(15, "bonjour");
```

- Il est aussi possible de faire l'héritage entre interface.

```
interface I3 extends I2 {  
    c: boolean;  
}
```





# Les fonctionnalités de TypeScript

## Le typage générique

- Le typage générique ajoute un niveau d'abstraction en rendant les types paramétrables, que ce soit dans une fonction, une classe ou une interface, sachant qu'au moment de l'appel effectif à cette fonction ou à cette classe, le type devra être explicitement défini.

```
function concatenate<T>(a1: T[], a2: T[]): T[] {  
    return a1.concat(a2);  
}  
  
resultNumbers = concatenate<number>([1, 2], [3, 4]); // [1, 2, 3, 4]  
resultStrings = concatenate<string>(["a", "b"], ["c", "d"]); // ["a", "b", "c", "d"]  
resultError1 = concatenate<number>([1, 2], ["a", "b"]); // erreur  
resultError2 = concatenate<string>([1, 2], ["a", "b"]); // erreur  
resultAny = concatenate<any>([1, 2], ["a", "b"]); // [1, 2, "a", "b"]
```







# Les fonctionnalités de TypeScript

## L'importation de modules externes

- Considérons l'exemple ci-dessous :

### module2.ts

```
export var id = 0;
export function fct() { ... }
export interface I {
  a: number;
}
export class C implements I {
  a: number;
}
```

ce fichier module2.ts peut être importé dans un autre fichier, qu'on supposera pour cette l'exemple présent dans le **même** répertoire que module2.ts, de la manière suivante :



```
import m = require("./module2");

m.id++;
```

le symbole m ainsi défini via le mot-clé import est un nouvel espace de nommage englobant les éléments exportés du fichier module2.ts





# Les fonctionnalités de TypeScript

## Les décorateurs

- Les décorateurs représentent une nouvelle fonctionnalité ajoutée seulement en TypeScript 1.5 pour supporter Angular.
- Un décorateur est une façon de faire de la méta-programmation et ressemblent beaucoup aux annotations qui sont principalement utilisées en java, c#, et python.
- Les décorateurs peuvent modifier leur cible (classes, méthodes, etc...) et par exemple modifier les paramètres ou le résultat retourné, appeler d'autres méthodes quand la cible est appelée, ou ajouter des métadonnées destinées à un Framework à un bout de code (c'est ce que font les décorateurs d'Angular 2)
- En TypeScript, les annotations sont préfixées par @, et peuvent être appliquées sur une classe, une propriété de classe, une fonction, ou un paramètre de fonction. Pas sur un constructeur en revanche, mais sur ses paramètres oui.





# Les fonctionnalités de TypeScript

## Les décorateurs

### ❑ Décorateurs de fonctions :

- Pour mieux comprendre ces décorateurs, on illustre un exemple de décorateur de fonction **@Log()**, qui va écrire le nom de la méthode à chaque fois qu'elle sera appelée.

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



```
class RaceService {  
  
  @Log()  
  getRaces() {  
    // call API  
  }  
  
  @Log()  
  getRace(raceId) {  
    // call API  
  }  
}
```

- Selon ce sur quoi nous voulons appliquer notre décorateur, la fonction n'aura pas exactement les mêmes arguments. Ici nous avons un décorateur de méthode, qui prend 3 paramètres :
- **target** : la méthode ciblée par notre décorateur
  - **name** : le nom de la méthode ciblée
  - **descriptor** : le descripteur de la méthode ciblée, par exemple est-ce que la méthode est énumérable, etc...





# Les fonctionnalités de TypeScript

## Les décorateurs

### ❑ Décorateurs de propriétés:

- Dans cet exemple on retourne la chaîne de valeur "test" au lieu de la valeur de la propriété
- La valeur d'une propriété ne peut être modifiée. On utilise donc un accesseur (get)

```
function Override(label: string) {  
  return function (target: any, key: string) {  
    Object.defineProperty(target, key, {  
      configurable: false,  
      get: () => label  
    });  
  }  
}  
  
class Test {  
  @Override('test')    // invokes Override, which returns the decorator  
  name: string = 'pat';  
}  
  
let t = new Test();  
console.log(t.name); // 'test'
```





# TypeScript : TSC



- Récupérer l'install de Node.js <https://nodejs.org/en/download/>
- Installer Node.js
- Ouvrir l'invite de commande windows. (**cmd**)
- Vérifier que la commande le gestionnaire des paquets **npm** est bien référencé dans la variable d'environnement **%PATH% : npm -version**  

```
D:\Angular>npm -version  
3.10.8
```
- Pour installer TypeScript , taper : **npm install -g typescript**
- Le compilateur TSC et le serveur Typscript sont téléchargé et installé.
- Vérifier que le transpileur est bien installé : **tsc -version**  

```
D:\Angular>tsc -version  
Version 1.0.3.0
```

o Node représente un environnement d'exécution (runtime), un ensemble d'API JavaScript ainsi qu'une machine virtuelle (VM) JavaScript performante (parseur, interpréteur et compilateur) pouvant accéder à des ressources système telles que des fichiers (filesystem) ou des connexions réseau (sockets).





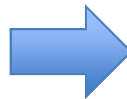
# TypeScript : le transpileur TSC



- Une petite recette de l'installation :

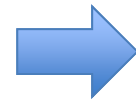
```
class Pizza {  
  toppings: string[];  
  constructor(toppings: string[]) {  
    console.log("executed code from node");  
    this.toppings = toppings;  
  }  
}  
new Pizza(null);
```

tsc Pizza.ts



```
var Pizza = (function () {  
  function Pizza(toppings) {  
    console.log("executed code from node");  
    this.toppings = toppings;  
  }  
  return Pizza;  
})();  
new Pizza(null);
```

node Pizza.js



?





# TypeScript : le transpileur TSC

- Exercice pour se familiariser : compilation de plusieurs fichiers :

Ecrire et compiler les deux classes ci-dessous : Recette.ts, Formation.ts

## Formation.ts

```
class Formation {  
  cours: string;  
  constructor(cours: string) {  
    console.log(cours);  
    this.cours = cours;  
  }  
}  
  
new Formation("Angular2");
```

## Recette.ts

```
class Recette {  
  nom: string;  
  constructor(nom: string) {  
    console.log(nom);  
    this.nom = nom;  
  }  
}  
  
new Recette("Soupe aux petit pois");
```

Aide : tsc file1 file2





# TypeScript : le transpileur TSC

## ➤ Compilation de plusieurs fichiers:

tsc peut utiliser un fichier de configuration lors de la compilation.


Par défaut , le nom du fichier est **tsconfig.json**

Une squelette de ce fichier peut être crée en tapant

tsc --init --target es5 --experimentalDecorators

Avec :

- « target » : langage de transpilation cible.
- « module » : Le chargeur des modules utilisé.
- sourceMap : permet de générer, si valeur égale à true, un fichier .map pour le mappage des fichiers. (util pour le débogage des fichiers .ts)



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

1. Créer le fichier de configuration et compiler en tapant **tsc**.
2. Modifier la valeur de sourceMap vers true et compiler une deuxième fois.
3. Que contient les fichiers .map générées ?
4. Accéder à la documentation en ligne du compilateur et expliquer à quoi sert **experimentalDecorators**

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







# TypeScript : le transpileur TSC

## ➤ Compilation à chaud:

Pointer sur le répertoire où se trouve **Formation.ts** et taper **tsc -watch** (ou **tsc -w**)

Modifier le fichier **Formation.ts** en ajoutant à la fin une nouvelle formation

```
new Formation("JAVA");
```

La modification est détectée automatiquement et le fichier **Formation.ts** a été compilée.





# TypeScript : le transpileur TSC

## ➤ Le fichier `package.config`

- Les projets TypeScript utilisent un fichier de configuration spécifique à node pour référencer les dépendances aux bibliothèques qui seront utilisées.
- **package.json** est le nom de ce fichier
- Ce fichier permet de décrire la configuration au gestionnaire de paquet npm et de dire quelles librairies on veut charger depuis le dépôt de node et quelles sont les scripts qui seront utilisé.
- Pour créer un fichier de configuration par défaut, taper la commande : **npm init**
- Répondre aux questions posées par la console (npm).





# TypeScript :



## ➤ Travaux pratiques

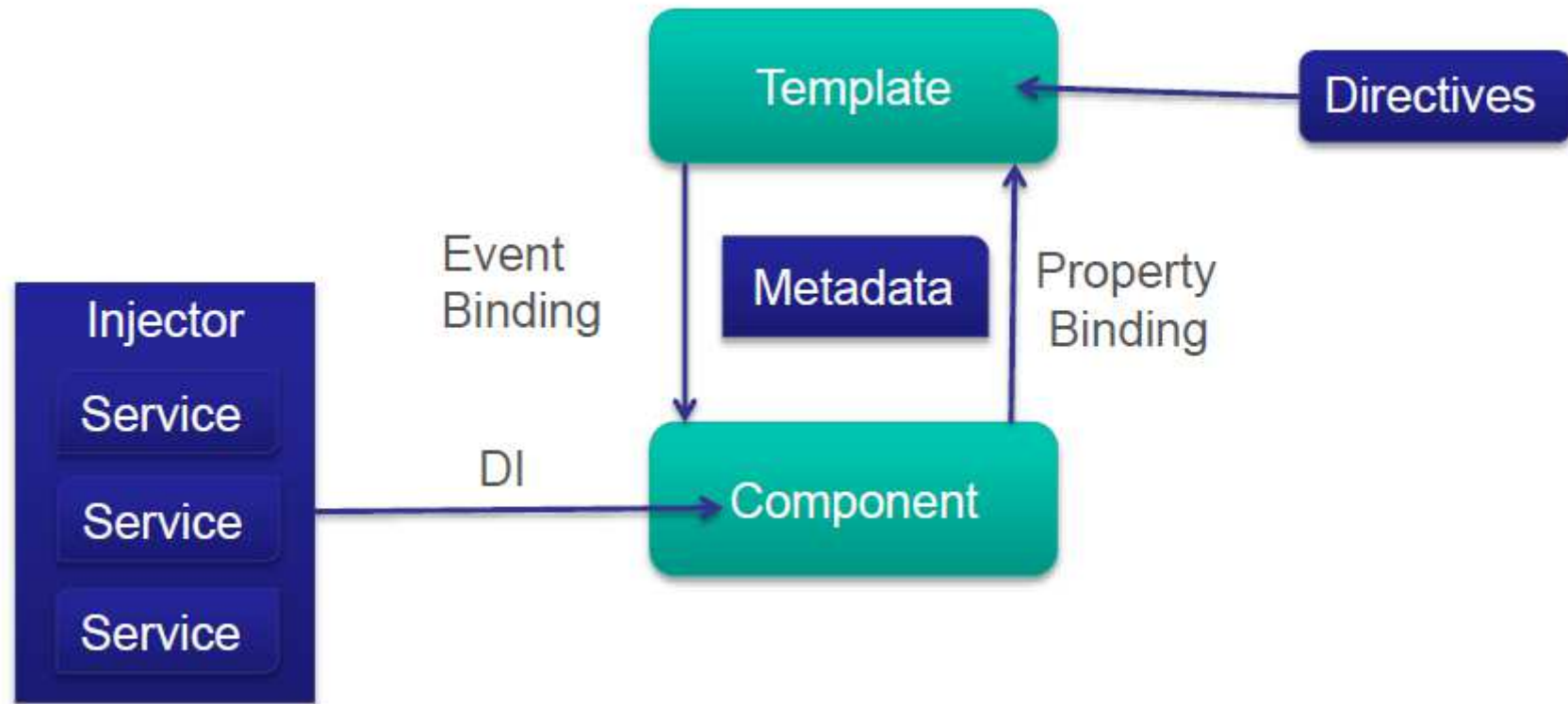
### G:\Transfert\Formation Angular 2\

- Récupérer le répertoire TP1-TypeScript
- Sur ce répertoire, créer les fichiers tsconfig, package.json
- Exécuter la commande **npm install** pour installer les dépendances
- Compiler le projet à l'aide du compilateur **tsc**
- Résoudre les problèmes de compilation s'ils y'en a.
- Réaliser le TP en complétant le code du fichier **ts/main.ts (Questions dans le code)**
  - Rappel : **tsc -w** pour une compilation à chaud.





# Angular 2 : Architecture





## Les modules



- Les applications Angular sont modulaires et Angular possède son propre système de gestion de modules appelé simplement modules Angular ou **NgModules**.
- Chaque application Angular possède au moins un module, le module racine, nommé par convention **AppModule**.





# Les modules



- Ce concept de module a été introduit à partir de la version **RC5** de Angular2
- Parmi les avantages de la modularité :
  - Une application peut être organisé en petit block fonctionnel.
  - Extension des applications en utilisant des modules existants :
  - Angular offre plusieurs modules standard comme FormsModule, HttpClientModule, RouterModule...
  - Un module regroupe : les composants, les directives et les pipes en blocs fonctionnels.





## Les modules



- Un module Angular, qu'il s'agisse d'un module racine ou d'un module fonctionnel, est une classe avec le décorateur **@NgModule**.
- L'utilisation de l'annotation @NgModule permet d'associer un ensemble de métadonnées à une classe.





# Les modules



- **Les métadonnées qu'on définit pour un module sont :**
  - **déclarations** - les classes de vue (view classes) appartenant à ce module. Angular possède trois types de classes de vue : les composants, les directives, et les pipes.
  - **imports** - les modules dont les classes exportées sont requises par les templates des composants déclarés dans ce module.
  - **providers** - les créateurs de services que ce module apporte à la collection globale de services de l'application ; ils seront accessibles partout dans l'application.
  - **bootstrap** - la principale vue de l'application, appelée le composant racine, qui accueille toutes les autres vues de l'application. Seul le module racine doit définir cette propriété bootstrap.







# Les modules



## ➤ Exemple : (app.module.ts)

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

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

## ➤ On lance une application en bootstrapant son module racine.

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

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





Ecrire son premier module



*Démo*





## Ecrire son premier module



### ➤ Travaux pratiques

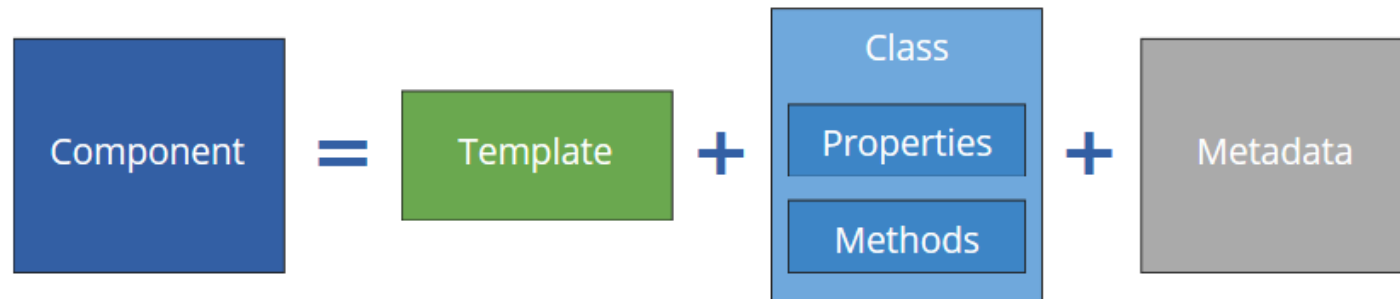
- Créer le fichier **tsconfig.json**
- Créer le fichier **package.json**
- Créer une classe AppModule (**app.module.ts**)
- Ouvrir l'invite de commande et exécuter la compilation en mode écoute
- Ajouter le décorateur @NgModule à la classe.
- Résoudre les problèmes de compilation du module (installer les packages requis : @angular/core, @types/core-js, Ajouter les imports)
- Créer le fichier main.ts pour bootstraper le module AppModule (consulter la documentation <https://angular.io/docs/ts/latest/guide/ngmodule.html#!#bootstrap>)





# Les composants : la philosophie d'Angular

- Angular est un Framework orienté composant : On constitue une application à partir de petits composants.
- Un composant est un groupe d'éléments HTML, dans un template, dédiés à une tâche particulière avec une logique métier derrière, pour peupler les données, et réagir aux événements.
- Les composants doivent être organisés de façon hiérarchique comme le DOM.





## Les composants : la philosophie d'Angular

- Les composants permettent une meilleure décomposition de l'application, facilitent le refactoring et le testing.
- Chaque composant est isolé des autres composants. Il n'hérite pas implicitement des attributs des composants parents.
- Les composants doivent communiquer et échanger des informations.
- Exemple d'utilisation de composants dans une page :

```
<wt-user-name-editor>  
  
  <wt-user-form></wt-user-form>  
  <wt-user-list>  
    <wt-user></wt-user>  
    <wt-user></wt-user>  
    <wt-user></wt-user>  
  </wt-user-list>  
  
</wt-user-name-editor>
```





# Les composants : la philosophie d'Angular





## Les composants : Templates et selecteurs

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

```
@Component({  
  selector: 'my-selector',  
  template: '<h1>Hello World</h1>'  
})
```

```
export class DemoComponent { }
```





## Les composants : Selecteurs

`<my-selector>Loading...</my-selector>`



```
@Component({  
  selector: 'my-selector',  
  template: '<h1>Hello  
World</h1>'  
})  
export class DemoComponent { }
```







## Les composants : Templates

```
<my-selector>Loading...</my-selector>
```



```
@Component({  
  selector: 'my-selector',  
  template: '<h1>Hello  
World</h1>'  
})  
export class DemoComponent { }
```

Output

Loading...





## Les composants : Templates

`<my-selector>Loading...</my-selector>`



```
@Component({  
  selector: 'my-selector',  
  template: '<h1>Hello  
World</h1>'  
})  
export class DemoComponent { }
```

Output

Hello World





## Les composants : MultiLine

```
<my-selector>Loading...</my-selector>
```



```
@Component({  
  selector: 'my-selector',  
  template:  
    <h1>Hello World</h1>  
    <div>  
      Welcome!  
    </div>  
})  
export class DemoComponent { }
```

Output

Hello World

Welcome



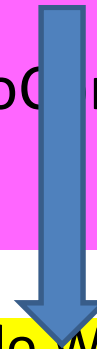


## Les composants : Templates dans fichiers externes

```
<my-selector>Loading...</my-selector>
```



```
@Component({  
  selector: 'my-selector',  
  templateUrl: 'myPage.html'  
})  
export class DemoComponent { }
```



Output

Hello World  
Welcome

myPage.html

```
<h1>Hello World</h1>  
<div>  
  Welcome!  
</div>
```



## Les composants : Propriétés

```
<my-selector>Loading...</my-selector>
```



```
@Component({  
  selector: 'my-selector',  
  templateUrl: 'myPage.html'  
})  
export class DemoComponent {  
  customerName:string = "David";  
}
```

Output

```
Hello World  
Welcome David
```

myPage.html

```
<h1>Hello World</h1>  
<div>  
  Welcome  
  {{customerName}}!  
</div>
```



## Liaison de données



- Interpolation
- One-Way Property Binding
- 2-Way Property Binding
- Event Binding





## Interpolation

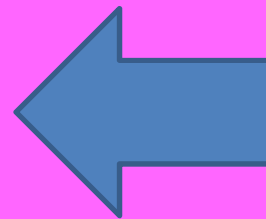
- Accolades autour des propriétés
- e.g.,  
{{customerName}}





## Interpolation

```
@Component({  
  selector: 'my-selector',  
  template: '<h1>Hello World</h1>'  
})  
export class DemoComponent {  
  id=1;  
  customerFirstName='David';  
  customerLastName='Giard';  
}
```

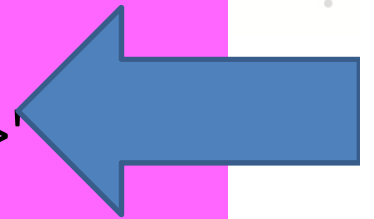






## Interpolation

```
@Component({  
  selector: 'my-selector',  
  template: '<h1>Hello {{customerFirstName}}</h1>',  
})  
export class DemoComponent {  
  id=1;  
  customerFirstName='David';  
  customerLastName='Giard';  
}
```



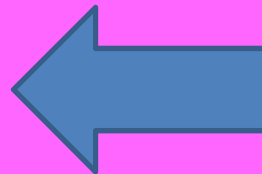
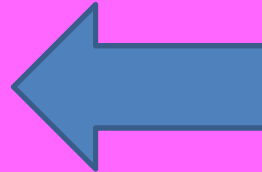
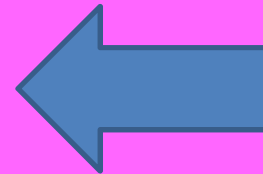
Hello David





# Interpolation

```
@Component({  
  selector: 'my-selector',  
  template: '<h1>Hello {{customer.FirstName}}</h1>'  
})  
export class DemoComponent {  
  id=1;  
  customer: Customer = {  
    FirstName='David';  
    LastName='Giard';  
  }  
}  
export class Customer{  
  FirstName: string;  
  LastName: string;  
}
```





# Interpolation

```
@Component({
  selector: 'my-selector',
  template: `
    <h1>{{customer.FirstName}} Details</h1>
    <div>First: {{customer.FirstName}}</div>
    <div>Last: {{customer.LastName}}</div>
  `
})
export class DemoComponent {
  id=1;
  customer: Customer = {
    FirstName='David';
    LastName='Giard';
  }
}
```

David  
Details

First: David  
Last: Giard





## 1-Way Data Binding

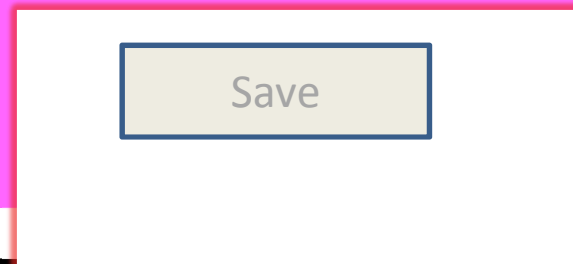
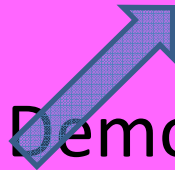
- Crochets autour des propriétés
- []





## 1-Way Data Binding

```
@Component({  
  selector: 'my-selector',  
  template: '<button  
[disabled]="dataNotChanged">Save</button>'  
})  
export class DemoComponent {  
  dataNotChanged= true;  
}
```

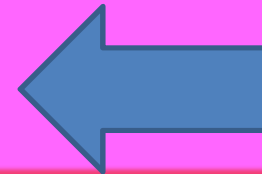




Liaison de données

# 1-Way Data Binding

```
@Component({  
  selector: 'my-selector',  
  template: '<button [disabled]="  
dataNotChanged">Save</button>',  
})  
export class DemoComponent {  
  dataNotChanged = false;  
}
```



Save





## 2-Way Data Binding

- Concerne les FormsModules
- [*property\_to\_bind*]





Liaison de données

## 2-Way Data Binding



```
@Component({
  selector: 'my-selector',
  template: `
<h1>{{customer.FirstName}} Details</h1>
<div>First: <input [(ngModel)]="customer.FirstName" </div>
<div>Last: <input [(ngModel)]="customer.LastName" </div>
`
})
export class DemoComponent {
  id=1;
  customer: Customer = {
    FirstName='David';
    LastName='Giard';
  }
}
```

1-way data binding

2-way data binding

David Details

First:

Last:







Liaison de données

## 2-Way Data Binding



```
@Component({
  selector: 'my-selector',
  template: `
    <h1>{{customer.FirstName}} Details</h1>
    <div>First: <input [(ngModel)]="customer.LastName" </div>
    <div>Last: <input [(ngModel)]="customer.FirstName" </div>
  `
})
export class DemoComponent {
  id=1;
  customer: Customer = {
    FirstName='David';
    LastName='Giard';
  }
}
```

D Details

First:

Last:





Liaison de données

## 2-Way Data Binding



```
@Component({
  selector: 'my-selector',
  template: `
    <h1>{{customer.FirstName}} Details</h1>
    <div>First: <input [(ngModel)]="customer.LastName" </div>
    <div>Last: <input [(ngModel)]="customer.FirstName" </div>
  `
})
export class DemoComponent {
  id=1;
  customer: Customer = {
    FirstName='David';
    LastName='Giard';
  }
}
```

Da Details

First: Da

Last: Giard





Liaison de données

## 2-Way Data Binding



```
@Component({
  selector: 'my-selector',
  template: `
    <h1>{{customer.FirstName}} Details</h1>
    <div>First: <input [(ngModel)]="customer.LastName" </div>
    <div>Last: <input [(ngModel)]="customer.FirstName" </div>
  `
})
export class DemoComponent {
  id=1;
  customer: Customer = {
    FirstName='David';
    LastName='Giard';
  }
}
```

Dan Details

First:

Last:





## Liaison des événements



`<control (eventname)="methodname(parameters)">`

### click event

`<control (click)="methodtocall(parameters)">`

e.g.,

`<div (click)="onClick(customer)">`



Ecrire son premier composant



*Démo*





## Ecrire son premier module



### ➤ Travaux pratiques

- Continuer sur le tp précédent sur la création du premier module.
- Créer une classe AppComponent (app.component.ts)
- Ouvrir l'invite de commande et exécuter la compilation en mode écoute
- Ajouter le décorateur **@Component** à la classe avec les métadonnées selector et template.
- Résoudre les problèmes de compilation du module (installer les packages requis : Ajouter les imports nécessaires)
- Créer le fichier index.html (Regarder le fichier déjà partagé sur le serveur)
- Installer et charger le module systemjs
- Créer la page index.html puis afficher le résultat.





## Exécuter sa premiere Application Angular

### Premier projet Angular 2:

- Récupérer le fichier **systemjs.config.js** partagé.
- Installer un serveur http local pour tester l'application : **npm install -g lite-server**
- Démarrer le serveur en ligne de command en pointant sur le répertoire de travail avec la command :  
lite-server
- **Accéder** à la page **http://localhost:3000**





## L'utilitaire ng ou @angular/cli



### Premier projet Angular 2 avec Angular-CLI

- Dans un vrai projet de développement il faudrait probablement mettre en place :
  - Des tests (unitaire, de bout en bout, non régression ...)
  - Un outil de construction pour orchestrer différents tâches (compiler, tester, packager, déployer ...)
- L'équipe Google a travaillé sur ce sujet profondément et ont sorti un outil très pratique : **angular-cli**
- angular-cli est un outil en ligne de commande pour démarrer rapidement un projet, déjà configuré avec un outil de construction , des tests, du packaging, etc...

- Installer Angular-CLI et créer un nouveau projet **ponyracer**

**npm install -g @angular/cli@latest**  
**ng new ponyracer**

- Cela va créer un squelette de projet plus complet que celui créé manuellement.
- Pour créer un composant taper ; **ng generate component pony**





## L'utilitaire ng ou @angular/cli

### Premier projet Angular 2 avec Eclipse

- Installez le plugin Angular2 Eclipse 1.1.0 .

#### Angular2 Eclipse 1.1.0



Angular2 Eclipse is a set of plugins which provide support for Angular2: it is based on TypeScript IDE for TypeScript support, it integrates angular-cli... [more info](#)

by [Angelo ZERR](#), EPL

[Angular2](#) [fileExtension js](#) [fileExtension jsx](#) [fileExtension ts](#) [fileExtension tsx](#)



Installs: **12,6K** (2 367 last month)

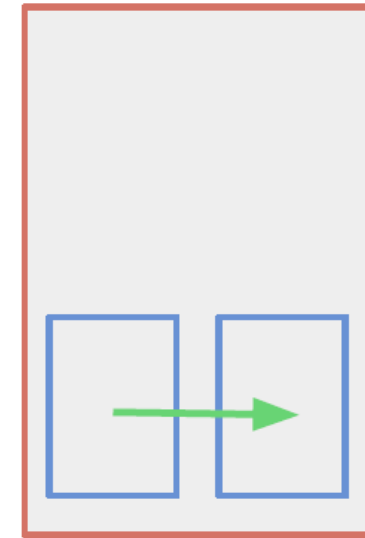
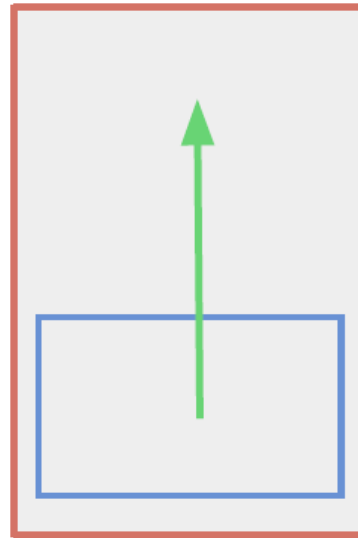
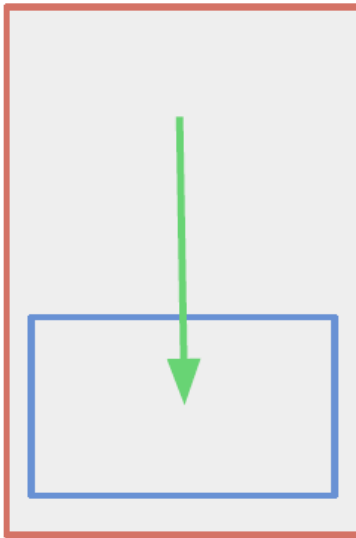
Uninstall

- Créer un nouveau projet Angular2. (New Angular2 project)
- Créer un composant EnovaRoot et bootstraper le.
- Modifier le fichier html en créant un tableau html contenant la liste des personnes présents dans la salle de formation.
- Créer une classe **Personne** avec les propriétés correspondantes. (Nom, prenom, age, formation , centresInteret...)
- Créer une instance de la classe Personne et Afficher les propriétés de cette instance dans la vue





## Communication entre les composants





## Communication entre les composants



Input

```
<html>  
  <myComp></myComp>  
</html>
```

Output

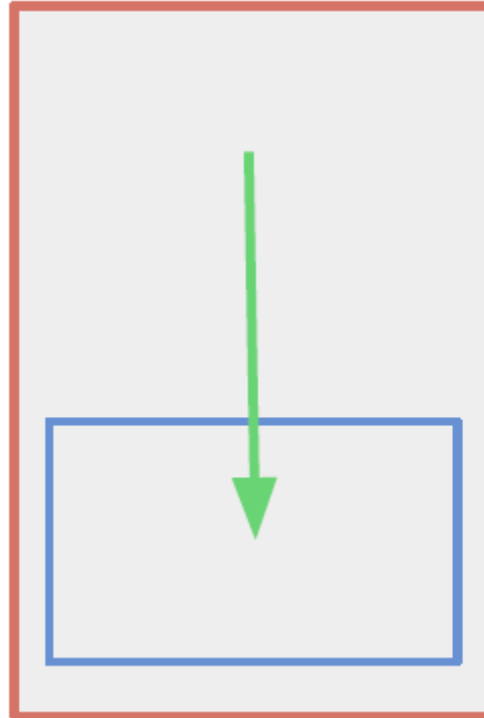
```
<html>  
  <myComp>  
    <div>  
      Content: hello  
    </div>  
  </myComp>  
</html>
```

```
@Component({  
  selector: 'myComp',  
  template: `  
    <div>  
      Content: {{myVar}}  
    </div>  
  `,  
})  
class MyComponent {  
  myVar: string;  
  
  constructor() {  
    this.myVar = 'hello';  
  }  
}
```





## Communication entre les composants





## Communication entre les composants

```
@Component({  
  selector: 'parent',  
  template: `  
    <div>  
      Parent content  
      <child [param]="myVar"></child>  
      Parent content  
    </div>  
  `,  
})  
class ParentComponent {  
  myVar = 'hello';  
}
```

```
@Component({  
  selector: 'child',  
  template: '<div>Child: {{param}}</div>',  
})  
class ChildComponent {  
  @Input() param: string;  
}
```

Input

```
<html>  
  <parent></parent>  
</html>
```

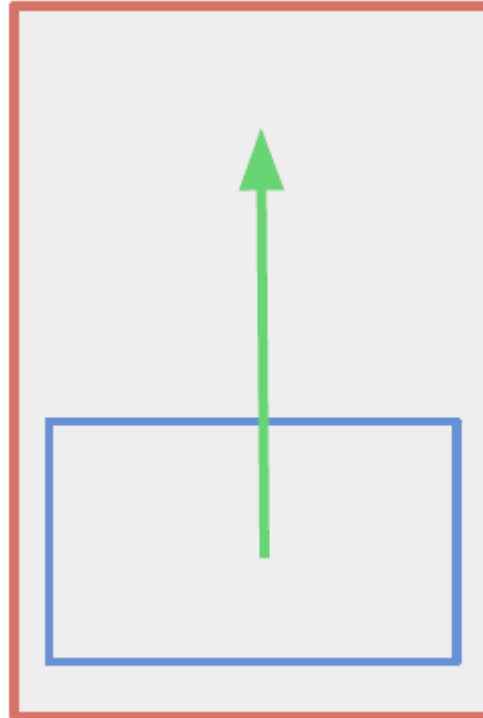
Output

```
<html>  
  <parent>  
    <div>  
      Parent content  
      <child>  
        Child hello  
      </child>  
      Parent content  
    </div>  
  </parent>  
</html>
```





## Communication entre les composants





## Communication entre les composants

```
@Component({
  selector: 'parent',
  template: `
    <div>
      <child (childEvent)="handleChildEvent($event)"></child>
    </div>
  `
})
class ParentComponent {
  handleChildEvent(message) {
    console.log(message);
  }
}
```

```
@Component({
  selector: 'child',
  template: `
    <button (click)="childEvent.emit('clicked')">Click me</button>
  `
})
class ChildComponent {
  @Output() childEvent = new EventEmitter();
}
```





# Les directives



Les directives permettent d'attacher un comportement à un élément du DOM.

Dans Angular2 il y a 3 types de directives :

- les **composants** : on a vu qu'un composant est en fait une directive avec son sélecteur et son template,
- les **directives structurelles** : elles agissent sur le DOM en ajoutant ou retirant des éléments, on a vu **NgFor** et **NgIf**,
- les **directives attributs** : elles changent l'apparence ou le comportement d'un élément, on n'a pas encore eu l'occasion d'en utiliser mais ça viendra.





## Directives

- ngFor
- ngIf
- ngSwitch
- ngClass





## \*ngfor



```
var customers: Customer[] = [  
  { "id": 1, "firstName": " Satya", "lastName" : " Nadella" },  
  { "id": 2, "firstName": "Bill", "lastName": "Gates" },  
  { "id": 3, "firstName": "Steve", "lastName": "Ballmer" },  
  { "id": 4, "firstName": " David ", "lastName": " Giard " }  
];
```

```
<div *ngFor="let cust of customers">  
  {{cust.lastName}}, {{cust.firstName}}  
</div>
```

Nadella, Satya  
Gates, Bill  
Ballmer, Steve  
Giard, David



## \*ngIf

- Syntaxe : `*ngIf="condition"`
- Supprime l'élément du DOM si la condition n'est pas remplie.





**\*ngIf**



```
<h1>People I  
hate:</div>  
<div *ngIf="true">  
    David Giard  
</div>
```

```
<h1>People I hate:</div>  
<div *ngIf="false">  
    David Giard  
</div>
```

People I hate:  
David Giard

People I hate:



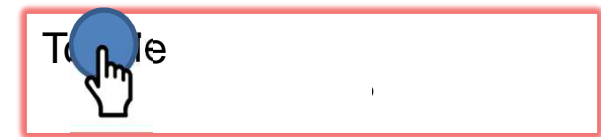


## \*ngIf



```
<div>
  <button (click)="clicked()">Toggle</button>
  <div *ngIf="show">
    Can you see me?
  </div>
</div>
```

```
export class DemoComponent {
  show: boolean = true;
  clicked() {this.show = !this.show; }
}
```



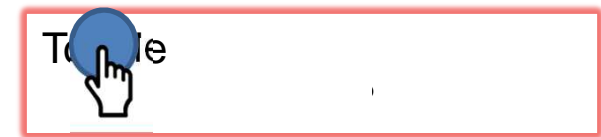


## \*ngIf



```
<div>
  <button (click)="clicked()">Toggle</button>
  <div *ngIf="show">
    Can you see me?
  </div>
</div>
```

```
export class DemoComponent {
  show: boolean = true;
  clicked() {this.show = !this.show; }
}
```





## \*ngSwitch



```
<ANY ng-switch="expression">  
  <ANY ng-switch-when="matchValue1">...</ANY>  
  <ANY ng-switch-when="matchValue2">...</ANY>  
  <ANY ng-switch-default>...</ANY>  
</ANY>
```





## Communication entre les composants



*Démo*







# Communication entre les composants



## Exercice 1: (suite)

- Créer une classe personne (nom, prenom, civilite, age, fonction, adresse, telephone)
- Créer un tableau de personne et afficher la liste en utilisant la directive (\*ngFor)
- Afficher les personnes dont l'age est supérieur à une valeur. (\*ngIf)
- Ajouter un button pour trier (croissant et décroissant) la liste des personnes par date d'arrivée à la formation.

## Exercice 2:

- Ajouter une classe Document (nomDoc:string, tailleDoc:number dossierParent:Dossier, extension:string)
- Ajouter une classe Dossier (nomDossier:string, listDocs,:Document[] )
- Concevoir une communication entre les deux composant pour :
  - ☐ Parcourir et Afficher la liste des documents d'un dossier.
  - ☐ Supprimer un document d'un dossier. (button supprimer)
  - ☐ Les sélecteurs du document doivent être utilisé dans la template du dossier uniquement.





## Les pipes



- Les "pipes" sont des filtres permettant de transformer des valeurs.
- La syntaxe des "pipes" est simplement inspirée des "pipes" des shell UNIX.

```
<div>{{ user.firstName | lowercase }}</div>
```

- Les "pipes" peuvent prendre des paramètres qu'il faut mettre après le "pipe" et séparés avec le symbole ":"

```
<div>{{ user.firstName | slice:0:10 }}</div>
```

- Les "pipes" peuvent être chaînés.

```
<div>{{ user.firstName | slice:0:10 | lowercase }}</div>
```





## Ecrire son premier pipe



- Ecrire un premier pipe pour filtrer la liste des personnes par age.
- Ecrire un deuxième pipe pour filtrer par Date aussi.

Utiliser la documentation en ligne pour vous aider.

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

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





# Hooks du Cycle de vie



- Ils permettent d'exécuter de la logique personnalisé à chaque étape de vie d'un composant
- Pourquoi ? Les données ne sont pas toujours disponible immédiatement dans le constructeur
- Utilisable seulement avec TypeScript
- Les interfaces des hooks sont optionnels (comme tout typage) mais fortement recommandé
- Hooks = methodes de notre composant





# Hooks du Cycle de vie



- **ngOnInit** - Appeler directement après la mise en place du binding
- **ngOnChanges(changes)** - Appeler quand un Input change
- **ngDoCheck** - Permet d'effectuer du code personnalisé
- **ngAfterContentInit** - Après que le contenu soit initialisé
- **ngAfterContentChecked** - Après chaque vérification du composant par Angular
- **ngAfterViewInit** - Après que la vue soit initialisé
- **ngAfterViewChecked** - Après vérification de la vue
- **ngOnDestroy** - Juste avant la suppression de ce composant





# OnInit



```
export class foo implements OnInit
{
    ...
    ngOnInit(){
    ...
    }
}
```





## Les hooks



```
import (Component, OnInit) from 'angular/core';

import {UserComponent} from '../user/user.component'

@Component({
  selector: 'users',
  templateUrl: '../users.component.html',
  directives: [UserComponent]
})
export class UsersComponent implements OnInit () {
  users: User[]
  defaultUser: User = {
    firstname: 'Default Firstname',
    lastname: 'Default Lastname',
  }
  constructor(
    private _userService: UserService
  ) {
    ngOnInit() {
      this.users = this._userService.get()
    }
    updateUser(index:number, user: User) {
      this.users = this._userService.update(index, user)
    }
  }
}
```





## Pratiques : Hooks



Utiliser la documentation en ligne pour choisir et créer des Hooks

Expliquer chacun l'utilité du Hook choisi.







# L'injection des dépendances

- L'injection des dépendances est un pattern de conception permettant de faciliter la gestion des dépendances, améliorer l'extensibilité d'une application et faciliter les tests-unitaires.
- Sans injection de dépendance ;

```
class UserStore {  
    getUser(userId: string): Observable<User> {  
        let restApi = new RestApi(new ConnectionBackend(), new RequestOptions({headers: ...}));  
        return restApi.users.get(userId);  
    }  
}
```

- ☐ Il faut savoir comment instancier `RestApi` ?
- ☐ Comment factoriser ?
- ☐ Comment contrôler l'implémentation de la classe `RestApi` ?





## L'injection des dépendances

- Angular 2 dispose d'un "injector" qui implémente une factory permettant d'instancier des classes et maintenir les instances.

```
const injector = new Injector([RestApi]);  
const restApi1 = injector.get(RestApi);  
const restApi2 = injector.get(RestApi); // restApi2 is the same instance as restApi1.
```

- Lors du "bootstrap", Angular 2 crée le "root injector" qui sera chargé d'injecter les dépendances de l'application.
- On indique qu'une dépendance est injectable à l'aide du "decorator" `@Injectable`.
- Angular 2 créera alors une instance unique de la dépendance disponible dans toute l'application.





# L'injection des dépendances

- Pour injecter une dépendance, il faut utiliser le "decorator" `@Inject``.

```
class UserStore {  
    constructor(@Inject(RestApi) restApi) {  
        ...  
    }  
}
```

- Mais grâce à TypeScript, Angular 2 arrive à retrouver la dépendance à partir de son type.

```
class UserStore {  
    constructor(restApi: RestApi) {  
        ...  
    }  
}
```

```
@Injectable()  
class RestApi {  
    ...  
}  
  
@NgModule({  
    bootstrap: [ UserNameEditorComponent ],  
    declarations: [  
        UserNameEditorComponent  
    ],  
    imports: [  
        BrowserModule,  
        FormsModule,  
        HttpClientModule  
    ],  
    providers: [  
        RestApi  
    ]  
})  
export class AppModule {  
}
```





## Services



- Classe qui contiennent une logique ou un traitement
- Code partagé et utilisé par d'autres modules ou composant.
- Pattern d'injection de dépendance

# Services

## CustomerService.ts

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

```
@Injectable()
export class CustService {
  getCustomers() {
    return customers;
  }
}
```

```
var customers: Customer[] = [
  { "id": 1, "firstname": "David", "lastname": "Giard" },
  { "id": 2, "firstname": "Bill", "lastname": "Gates" },
  { "id": 3, "firstname": "Steve", "lastname": "Ballmer" },
  { "id": 4, "firstname": "Satya", "lastname": "Nadella" }
];
```

# Services

CustomerService.ts

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

```
@Injectable()
export class CustService {
  getCustomers() {
    return customers;
  }
}
...
```

```
import { OnInit } from '@angular/core';
import { CustService } from CustomerService
```

```
@Component({
  selector: 'xxx',
  template: 'yyy',
  ...
  providers: [CustService]
```

```
})
export class DemoComponent implements OnInit {
```

```
  constructor(private customerService:CustService) { }
```

```
  ngOnInit() {
    this.customers = this.customerService.getCustomers();
  }
}
```



# Le pattern observable



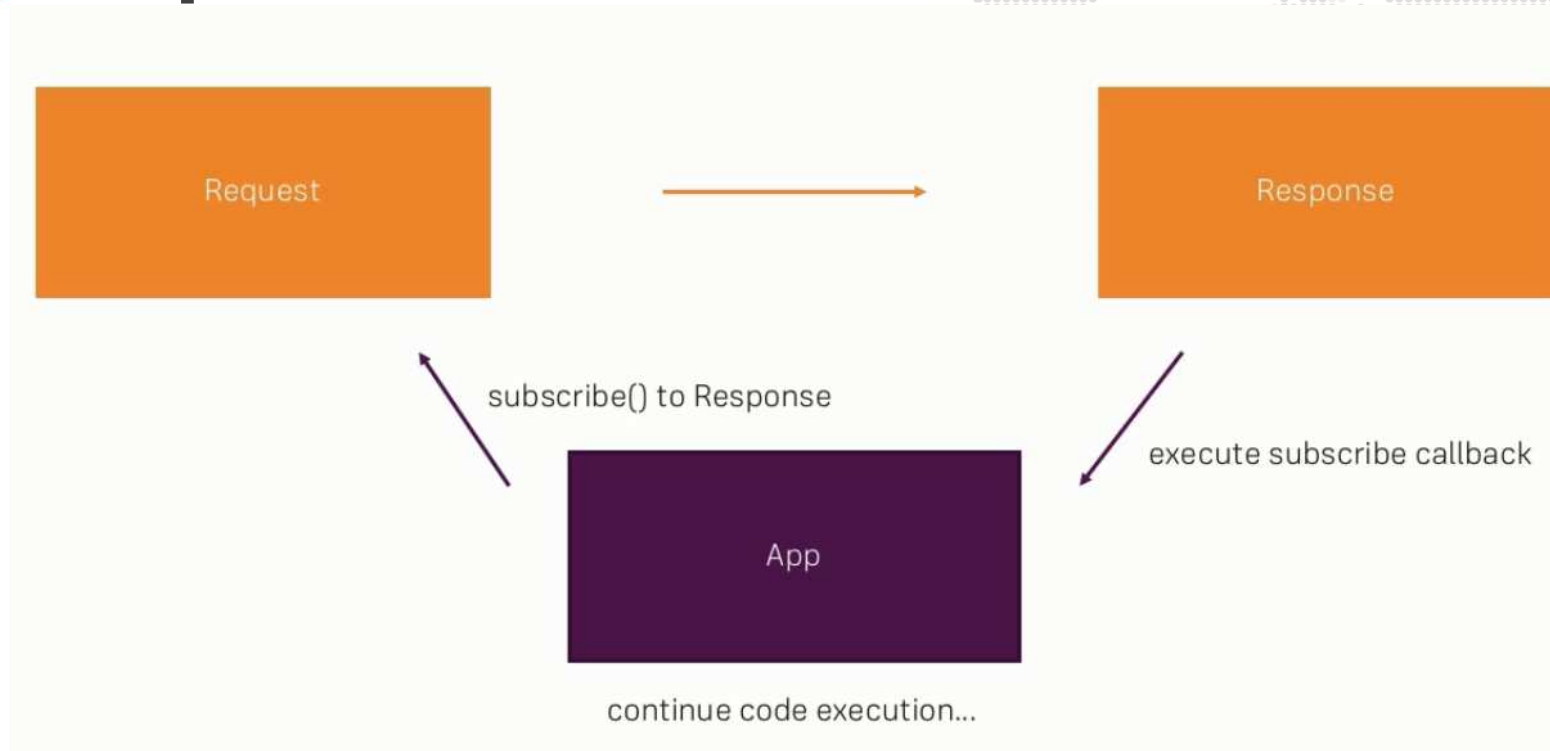
## Qu'est ce qu'un observable

- Un flux d'évènement poussé dans le temps (lazy event stream) qui peut émettre zéro ou plusieurs évènements
- Composé de *subjects* et d'*observers*
- Le *subject* applique de la logique au flux et notifie l'*observer* quand il est nécessaire





# Le pattern observable





# Le module HTTP

Angular dispose d'un client pour consommer les web services de type REST

Il s'agit de l'objet de la classe Http

```
Http http = ...  
  
http.get(url).subscribe(  
    response => console.log(response.json()),  
    error => console.error(error)  
);
```

<https://angular.io/docs/ts/latest/guide/server-communication.html>  
<https://angular.io/docs/ts/latest/api/http/Http-class.html>

# Le module HTTP



## *Démo*

<https://jsonplaceholder.typicode.com>



## Les formulaires

- Les formulaires sont la première forme d'interaction avec l'utilisateur pour mettre à jour des données.
- Angular propose deux types de formulaire :



FormsModule  
(Template-driven)

ReactiveFormsModule  
(Reactive)





## Les formulaires

- 3 composants principaux à mémoriser :



FormControl

FormGroup

FormArray





## Les formulaires



Value  
Validation status  
User interactions  
Events

First name \*

---





## Les formulaires

First name \*

---

### FormControl

Value

Validation status

User interactions

Events





## Les formulaires

```
const control = new FormControl();

control.value           // null

control.status         // VALID

control.valid          // true

control.pristine       // true

control.untouched      // true
```





## Les formulaires

```
const control = new FormControl();

control.setValue('Nancy');

control.value                // 'Nancy'

control.reset();

control.value                // null

control.disable();

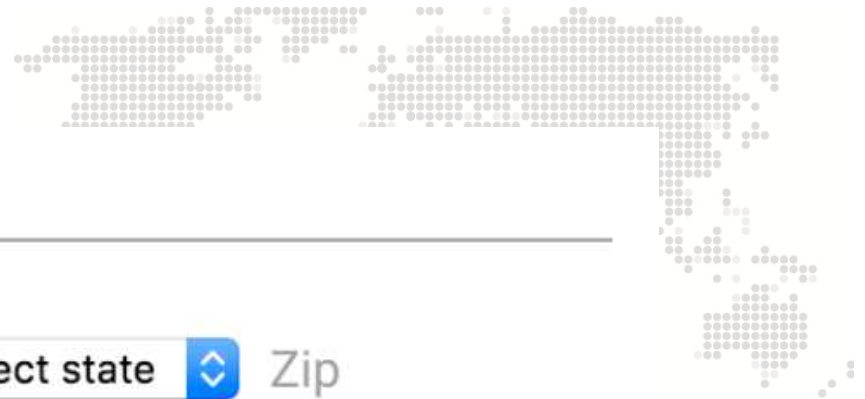
control.status                // DISABLED
```







# Les formulaires




Street

---

City

---

Select state  Zip

---

## FormGroup

FormControl name: street	FormControl name: city
FormControl name: state	FormControl name: zip





# Les formulaires

Street

City

Select state

Zip

Street

City

Select state

Zip

Street

City

Select state

Zip

Street

City

Select state

Zip



## FormArray

FormControl  
0

FormControl  
1

FormControl  
2

FormControl  
3





## Les formulaires



```
const arr = new FormArray([
  new FormControl('SF'),
  new FormControl('NY')
]);

arr.value // ['SF', 'NY']

arr.status // VALID

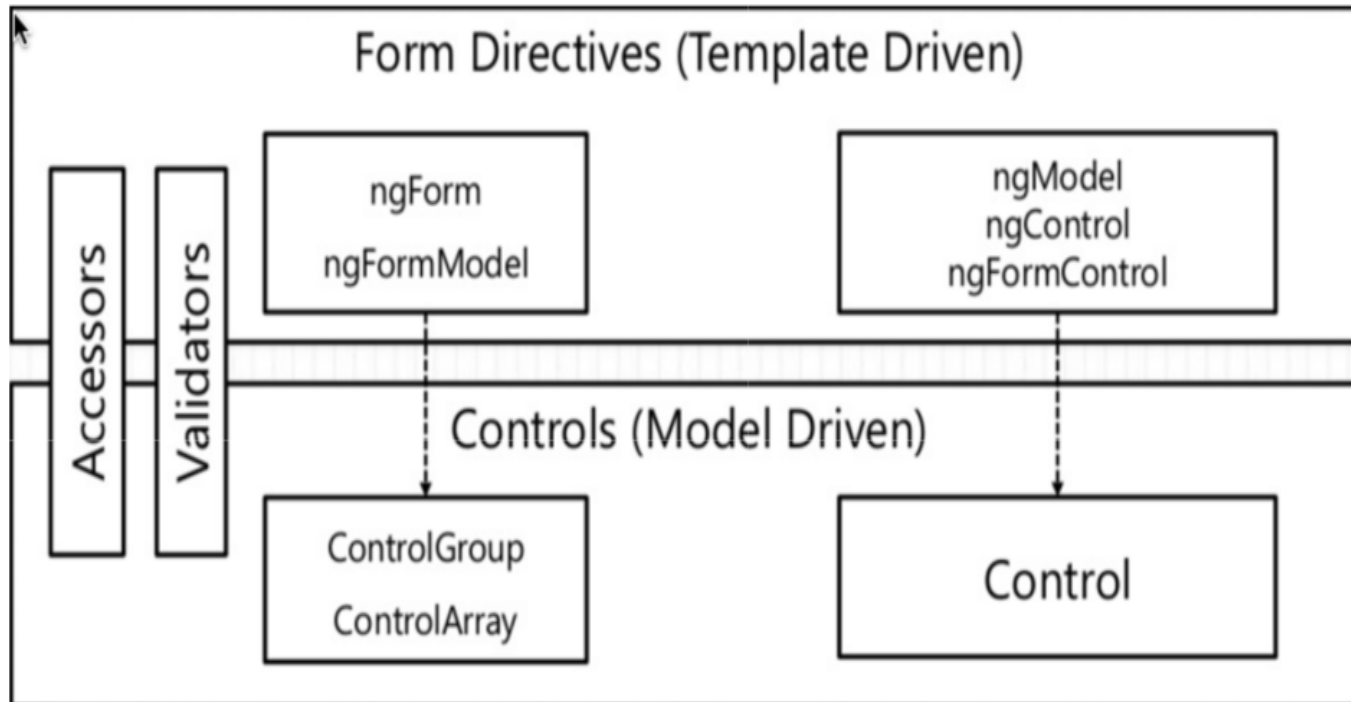
arr.setValue(['LA', 'LDN']); // ['LA', 'LDN']

arr.push(new FormControl('MTV')); // ['LA', 'LDN', 'MTV']
```





## Les formulaires





## Les formulaires



### ➤ Utilisation du ngModel :

- Utilise la notation du two-way bindings : [(ngModel)]
- Une des seules directives à utiliser le two-way binding
- Il permet de lier un champs input avec un model

```
import { FORM_DIRECTIVES } from '@angular/common'

@Component({
  selector: "my-form",
  directives: [FORM_DIRECTIVES],
  template: `<input type="text" [(ngModel)]="name">`
})
class MyForm {
  name: string
}
```





## Les formulaires



### ➤ Utilisation du ngModel :

ngModel nous donne accès à l'état du formulaire et du champs

```
@Component({  
  selector: "my-form",  
  template: `  
    <input type="text" [(ngModel)]="name" #field="ngModel">  
    <pre>  
      {{ field.valid }}  
    </pre>  
  `,  
})  
class MyForm {  
  name: string  
}
```

- pristine
- dirty
- touched
- untouched
- errors
- valid





## Les formulaires



- Utilisation des directives `ngFormModel` et `ngControl` dans la vue

```
<form [ngFormModel]="loginForm" (submit)="doLogin($event)">
  <input ngControl="email" type="email" placeholder="Your email">
  <input ngControl="password" type="password" placeholder="Your password">
  <button type="submit">Log in</button>
</form>
```





## Les formulaires

- Utilisation des directives `ngFormModel` et `ngControl` dans la vue

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

@Component({
  selector: 'login-page',
  templateUrl: 'login-page.html'
})
export class LoginPage {
  constructor(private _formBuilder: FormBuilder) {
    this.loginForm = _formBuilder.group({
      email: ['', Validators.required],
      password: ['', Validators.required]
    });
  }
  doLogin(event) {
    console.log(this.loginForm.value);
    event.preventDefault();
  }
}
```







## Les formulaires



### ➤ Les validateurs natifs

- Angular 2 propose 4 validations natives :
  - `Validators.required`
  - `Validators.minLength`
  - `Validators.maxLength`
  - `Validators.pattern`





## Les formulaires



### ➤ Les validateurs natifs

```
// Component
this.name = new Control('', Validators.minLength(4));

// View
<input required type="text" ngControl="name" />
<div [hidden]="name.dirty && !name.valid">
  <p [hidden]="name.errors.minlength">
    Your name needs to be at least 4 characters.
  </p>
</div>
```





## Les formulaires



### ➤ Les validateurs personnalisés

- Nos validateurs personnalisés doivent respecter :
  - retourner null si valide
  - Respecter l'interface suivante

```
interface ValidationResult {  
  [key:string]:boolean;  
}
```





# Les formulaires



## ➤ Les validateurs personnalisés

```
// Component
function containsTroll(c: Control) {
  if(c.value.indexOf('troll') >= 0) {
    return {
      noTroll: true
    }
  }
  return null
}
this.name = new Control('', containsTroll);

// View
<input required type="text" ngControl="name" />
<div [hidden]="name.dirty && !name.valid">
  <p [hidden]="name.errors.noTroll">
    There is "troll" in your name ...
  </p>
</div>
```





## Les formulaires



## Démo





# Les formulaires



## Exercice 1 : FormModules

- Créer un formulaire basé sur la template pour ajouter des personnes dans un tableau.
  - Appliquer le css Bootstrap au formulaire.
  - Les champs (**Nom, prénom, email, Age, téléphone**) sont obligatoires et ne dépassent pas en longueur 50 caractères.
  - Le champs **email** doit en plus contenir le suffix **@foyer.lu**
  - **L'adresse** doit regrouper 3 champs : **Rue, Ville, Pays**
  - Créer un composant de recherche pour chercher les personnes qui habite à Salé. Ce composant doit communiquer avec le composant du formulaire.

## Exercice 1 : ReactiveFormModules

- Récrire le même formulaire en se basant sur le modèle.





# Les routes

## SPA & routeurs

- SPA = navigation sans rafraichissement
- Les URLs doivent être lisible par un humain et porter l'état de l'application demandé
- ex: pour accéder à la page météo de Montpellier

`http://site.com/meteo/montpellier`

ou

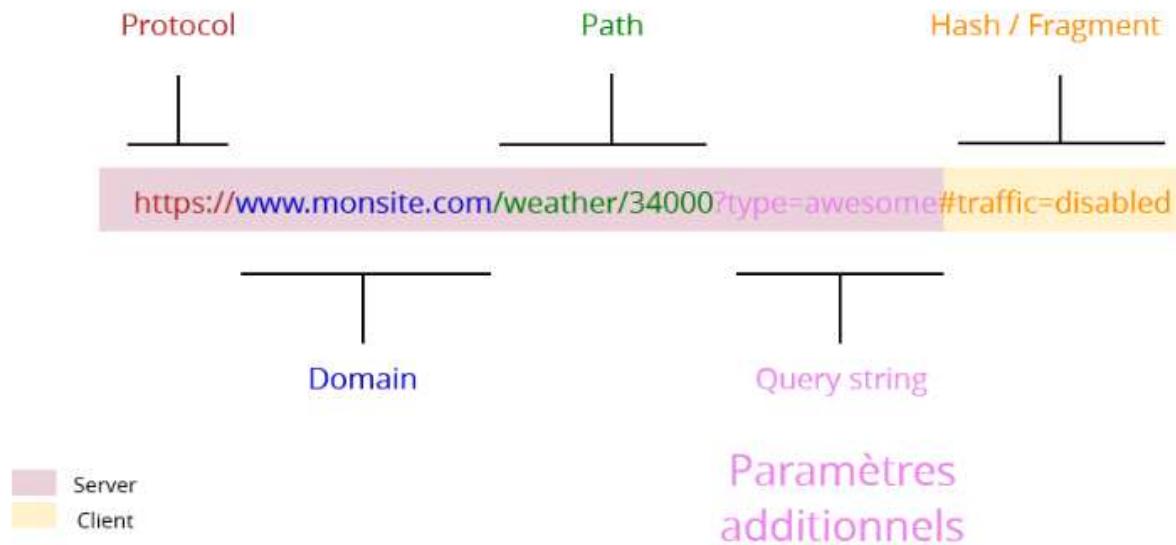
`http://site.com/meteo/43.6100788,3.8391422,13z`



# Les routes

## Une URL

Décrit l'état courant de  
l'application





# Les routes



## Configuration

```
export const routes: Routes = [  
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

# Les routes



Démo :

<https://plnkr.co/edit/YNMw1oaAeLSxIjCZZf2W?p=preview>



## Les routes



### Exercice 1 :

- En se basant sur la démo et sur la documentation en ligne. Créer une mini application basé sur les composants et le routage d'Angular 2 qui permet de :
- Afficher un menu (Ajouter/modifier/supprimer un dossier, Ajouter/modifier/supprimer un document)
- Ajouter des dossiers.
- Ajouter des documents à un dossier.
- Modifier un dossier
- Modifier un document
- Lister les dossiers et les documents dans une seule page, avec possibilité de faire un collapse (fermer et ouvrir le dossier)
- Ajouter des icons des dossiers et des documents dans l'affichage (par type de fichier : xls, xlsx, doc, docx ...)





## Testing et Intégration continue





# Plan du chapitre



Les tests dans Angular2

Outils et langages de tests (Karma, Jasmine, Protractor)

Tester un composant

Tester un service

La plateforme d'intégration continue

Angular 2 Style Guide





# Les tests dans Angular2



Angular a été conçu et créé en utilisant une architecture modulaire et testable.

On dit aussi que Angular2 a été fait pour être testé.



## Type des tests

- Unit tests
- Tester unitairement des fonctions du code.



Jasmine



- e2e test
- Tester le comportement réel d'une application.
- Simulation réel d'une interaction utilisateur.



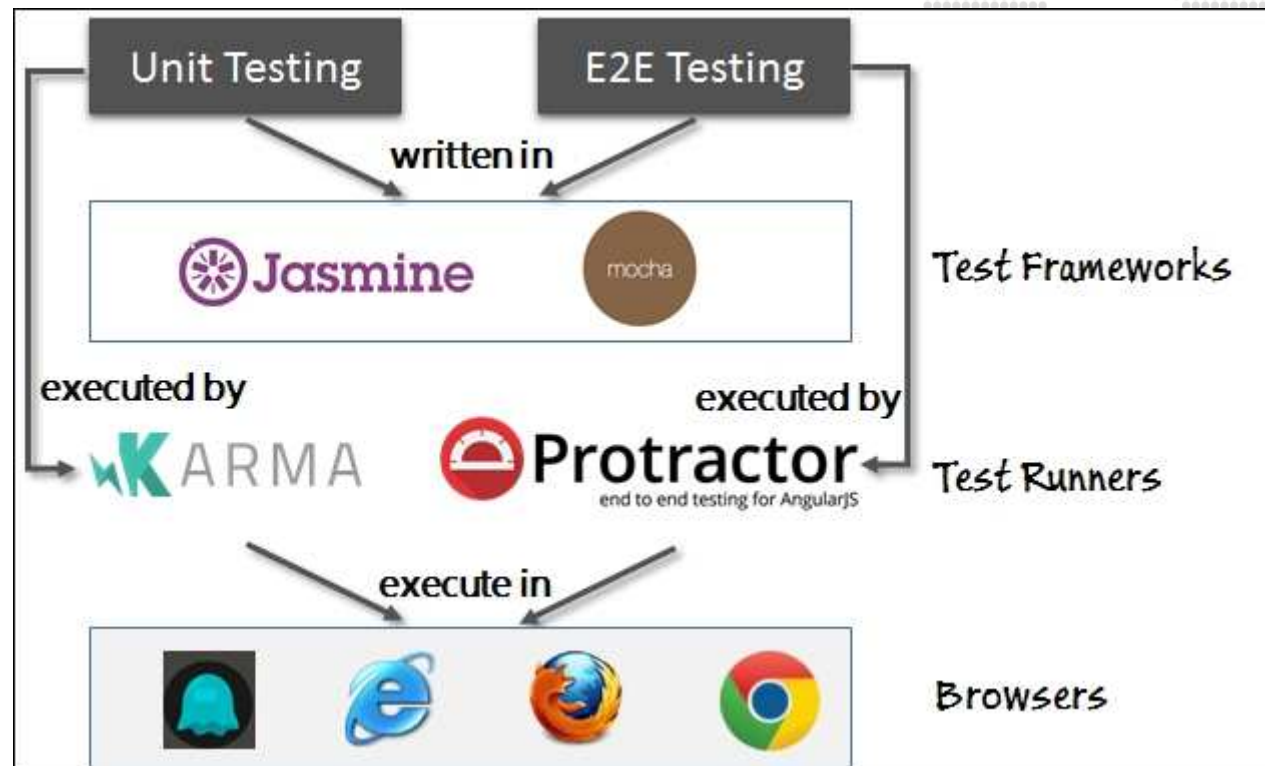
**Protractor**

end to end testing for AngularJS





# Architecture et outils de tests







Le but de [Karma](#) est de fournir un environnement d'exécution de vos tests Javascript. Les navigateurs Internet ne sont pas conçus pour charger nativement des fichiers de tests en javascript, les exécuter et afficher un rapport d'exécution.

Karma a été créé pour répondre à cette problématique. Voici les principales fonctionnalités de cette librairie :

Serveur web : il démarre un serveur web allégé pour lancer vos fichiers de tests. Ces fichiers peuvent être écrits à l'aide de différents framework (Jasmine, Qunit, Nunit, Mocha,...)

Runner : il fournit une page custom qui lancera les tests. Cette page est différente selon le framework de test utilisé.

Manager : il démarre un navigateur Internet (Client) pour charger cette page. Vous pouvez utiliser différents navigateurs comme Firefox, Chrome, IE, PhantomJS...

Reporter : il génère des rapports d'exécution soit sous forme de fichiers, dans la console d'exécution...

Watcher : Karma fournit des plugins pour analyser les changements sur le filesystem pour relancer automatiquement les tests...

Karma s'interface facilement avec vos serveurs d'intégration continue comme [Jenkins](#) ou [Travis](#).





## Installation :

`npm install -g karma karma-jasmine karma-chrome-launcher`

## Configuration :

`karma init`

## LANCEMENT:

`karma start`

Plugins : <https://www.npmjs.com/browse/keyword/karma-reporter>



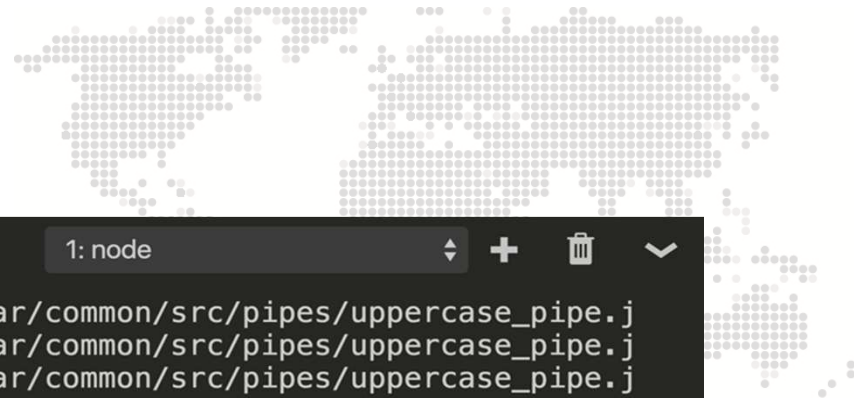
## karma et typescript :

<https://www.npmjs.com/package/karma-typescript>

```
npm init  
npm install --save-dev karma-typescript
```



# KARMA



TERMINAL

1: node

```
68% building modules 869/887 modules 18 active ...ar/common/src/pipes/uppercase_pipe.j
68% building modules 870/887 modules 17 active ...ar/common/src/pipes/uppercase_pipe.j
68% building modules 871/887 modules 16 active ...ar/common/src/pipes/uppercase_pipe.j
68% building modules 872/887 modules 15 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 873/887 modules 14 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 874/887 modules 13 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 875/887 modules 12 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 876/887 modules 11 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 877/887 modules 10 active ...ar/common/src/pipes/uppercase_pipe.j
12 01 2017 12:46:04.202:WARN [karma]: No captured browser, open http://localhost:9876/
12 01 2017 12:46:04.214:INFO [karma]: Karma v1.2.0 server started at http://localhost:9
876/
12 01 2017 12:46:04.214:INFO [launcher]: Launching browser Chrome with unlimited concur
rency
12 01 2017 12:46:04.224:INFO [launcher]: Starting browser Chrome
12 01 2017 12:46:05.077:INFO [Chrome 55.0.2883 (Mac OS X 10.12.2)]: Connected on socket
/#fU7Q4c3qyRaadrnuAAAA with id 720766
Chrome 55.0.2883 (Mac OS X 10.12.2): Executed 6 of 6 SUCCESS (0.649 secs / 0.639 secs)
```





# Karma : installation



Démo





# Jasmine : simple test

```
describe(`Component: JokeComponent`, () => {
```

```
  it(`should add 1 + 1 `, () => {  
    expect(1 + 1).toEqual(2);  
  });
```

```
  it(`should add 1 + 1 `, () => {  
    expect(1 + 1).toEqual(3);  
  });
```

```
});
```



Jasmine





# Jasmine : simple test

Démo



Jasmine





# Jasmine : Test d'un composant

```
describe(`Component: JokeComponent`, () => {  
  
  it(`should add 1 + 1`, () => {  
    expect(1 + 1).toEqual(2);  
  });  
  
  it(`should have a title of "Chuck Norris Quotes"`, () => {  
    const component = new JokeComponent(null);  
    expect(component.title).toEqual('Chuck Norris Jokes');  
  });  
  
});
```







# Jasmine : Test d'un composant

Démo



Jasmine





# Jasmine : TestBed API

```
Testbed.configureTestingModule({  
  imports: [HttpModule],  
  declarations: [JokeComponent],  
  providers: [JokeService]  
});
```



Jasmine





# Jasmine : TestBed API

```
fixture = TestBed.createComponent(JokeComponent);  
  
component = fixture.componentInstance;  
  
debugElement = fixture.debugElement;  
  
fixture.detectChanges();
```



Jasmine





# Jasmine : TestBed API



```
jokeText = debugElement.query(By.css(`p`)).nativeElement;
```



Jasmine





# Jasmine : TestBed API

```
describe(`Component: JokeComponent`, () => {  
  let component: JokeComponent;  
  let jokeService: JokeService;  
  let fixture: ComponentFixture<JokeComponent>;  
  let de: DebugElement;  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      imports: [HttpModule],  
      declarations: [JokeComponent],  
      providers: [JokeService],  
    });  
  
    fixture = TestBed.createComponent(JokeComponent);  
    component = fixture.componentInstance;  
    jokeService = TestBed.get(JokeService);  
    de = Fixture.debugElement;  
  });  
});
```





# Jasmine : TestBed API

```
it(`should get display the joke content`, () => {  
    spyOn(jokeService, 'getJoke')  
        .and.returnValue(  
            Observable.of('FAKE_JOKE');  
  
    fixture.detectChanges();  
    let joke = de.query(By.css('p')).nativeElement;  
    expect(joke.textContent).toEqual('FAKE JOKE');  
});
```





# Jasmine : TestBed API



## Démo



Jasmine





# Travaux dirigés



## Application des TDD pour écrire des tests unitaires.

Lorsqu'un utilisateur saisi dans la zone de texte (champs **id**) d'un formulaire, le client à spécifié ce qui suit :

- spec 1: les espaces sont remplacés automatiquement par des **underscores**.
- spec 2: le premier caractère est transformé systématiquement en Majuscule.
- spec 3: si la saisie contient le mot «virus », l'utilisateur est averti que cet id n'est pas valable et le mot virus est supprimé.

- 1- Ecrire un fichier de test **agenda.service.spec.ts** qui test les fonctions à développer pour les répondres aux specs.
- 2- Exécuter **npm test** et vérifier que les tests échous et que le code ne compile pas
- 3- Ecrire un fichier **agenda.service.ts** et implémenter les méthodes testées.
- 4- Ré exécuter npm test , le code doit compiler et les tests doivent aboutir.
- 5- Réfactorer le code et ré exécuter **npm test** une dernière fois







# Les tests E2E



```
import { browser, by, element } from 'protractor'
describe(`Page: Joke Page`, () => {
  it(`should have a title of "Chuck Norris Jokes"`, () => {
    browser.get('/');
    let title = element(by.css('h1')).getText();
    expect(title).toEqual('Chuck Norris Jokes');
  });
  it(`should have a new joke on button click`, async() => {
    browser.get('/');
    let firstJoke = element(by.css('p')).getText();
    element(by.css('button')).click();
    let secondJoke = await element(by.css('p')).getText();
    expect(title).not.toEqual(secondJoke);
  });
});
```



# Builds et déploiement

```
scripts: {  
  "build": "ng build -prod",  
  "postbuild": "npm run deploy",  
  "predeploy": "rimraf ../server/src/main/resources/static &&  
    mkdirp ../server/src/main/resources/static",  
  "deploy": "copyfiles -f dist/** ../server/src/main/  
resources/static"  
}
```

Spring Boot app

static resources

Bundled Angular app





# Documentation



<https://angular.io/docs/ts/latest/guide/testing.html>





# Angular 2 style Guide



Regardons ensemble :

<https://angular.io/styleguide>



## Migrer de AngularJS à Angular



## Marche à suivre



- La migration se fait en deux temps :
- **1. Préparer l'appli Angular 1** en l'alignant avec Angular 2 avant de démarrer la migration.
  - Suivre le **style guide** AngularJS 1.x ([LIEN](#)).
  - Utiliser un **module loader**.
  - Migrer vers **TypeScript**.
  - Utiliser des “**directives composants**”.
- **2. Upgrade incrémental**, en exécutant les deux frameworks côte à côte dans la même application (**UpgradeAdapter**).





# Exemple UpgradeAdapter (1/2)

- **Downgrade** - Utiliser un composant Angular 2 comme une directive Angular 1 :

```
1. import { HeroDetailComponent } from './hero-detail.component';
2.
3. /* . . . */
4.
5. angular.module('heroApp', [])
6.   .directive('heroDetail',
     upgradeAdapter.downgradeNg2Component(HeroDetailComponent));
```

- Puis la directive s'utilise normalement dans un template Angular 1 :

```
<hero-detail></hero-detail>
```





## Exemple UpgradeAdapter (2/2)

- **Upgrade** - Utiliser une directive Angular 1 comme un composant Angular 2. NB.
- Seules les “directives composant” peuvent être upgradées.

```
1. import { Component } from '@angular/core';
2. import { upgradeAdapter } from '../upgrade_adapter';
3.
4. const HeroDetail = upgradeAdapter.upgradeNg1Component('heroDetail');
5.
6. @Component({
7.   selector: 'my-container',
8.   template: `
9.     <h1>Tour of Heroes</h1>
10.    <hero-detail></hero-detail>
11.  `,
12.   directives: [HeroDetail]
13. })
14. export class ContainerComponent {
15.
16. }
```





# ng-forward



- Permet d'écrire du code Angular 1.3+ qui respecte les **conventions et les patterns d'Angular 2.**
- Peut être une **première étape** avant d'écrire du vrai code Angular 2. Complémentaire au chemin d'upgrade décrit précédemment.
- Uniquement compatible avec ES6/TypeScript, pas ES5.
- Ressources :

<https://github.com/ngUpgraders/ng-forward>

<http://www.codelord.net/2016/02/03/angular-2-migration-whats-ng-forward/>



# Présentation de la Plateforme d'Intégration Continue



# La Plateforme d'Intégration Continue

## → Objectifs

La Plateforme d'Intégration Continue (PIC) est une plateforme de services d'industrialisation des développements



Intégration automatique des développements et détection des régressions



Audit automatique et périodique de la qualité du code



Gestion centralisée des composants et paquetages applicatifs

- Les services de la PIC sont destinés aux :
  - Équipes de développements
  - Chefs de projets





## La Plateforme d'Intégration Continue

➔ Les outils au « cœur » de la PIC

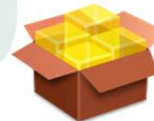
### Hudson/Jenkins

Intégration continue et périodique du code en vu de détecter les régressions.

PIC

### Sonar

Tableau de bord de suivi de la qualité du code.



### Nexus

Gestion centralisée des référentiels des composants techniques utilisés par les projets.



Des outils OpenSource, Matures et largement adoptés

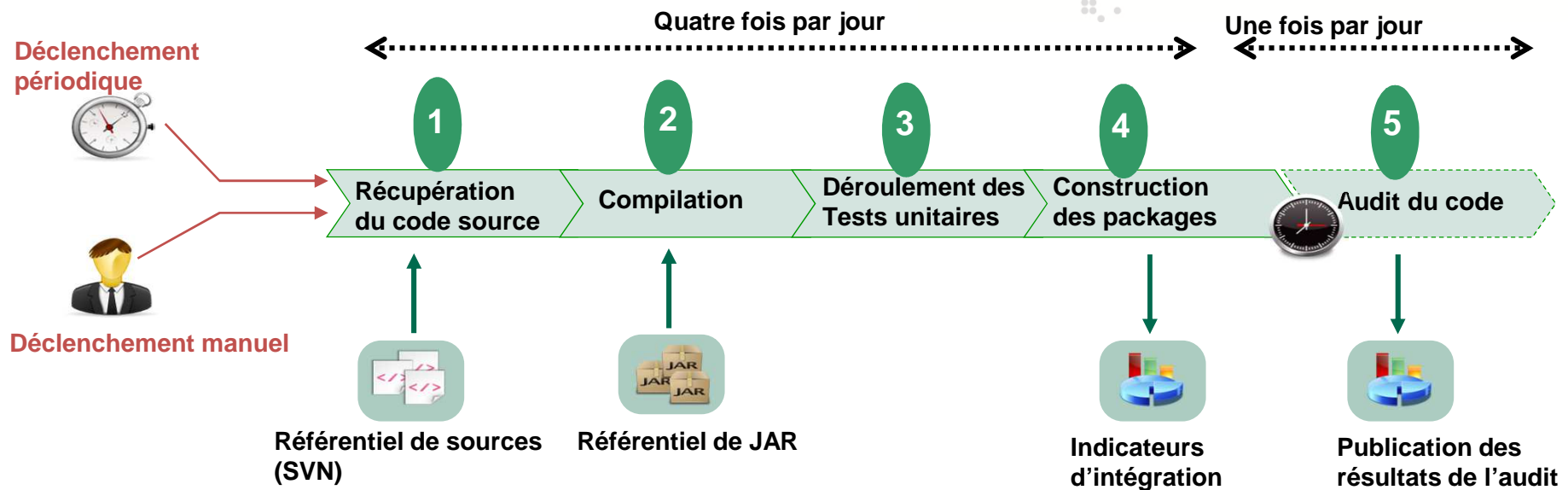




# La Plateforme d'Intégration Continue

## ➔ Processus d'intégration ?

- C'est un processus entièrement automatisé de transformation du code source en binaire avec déroulement des tests et audit du code.

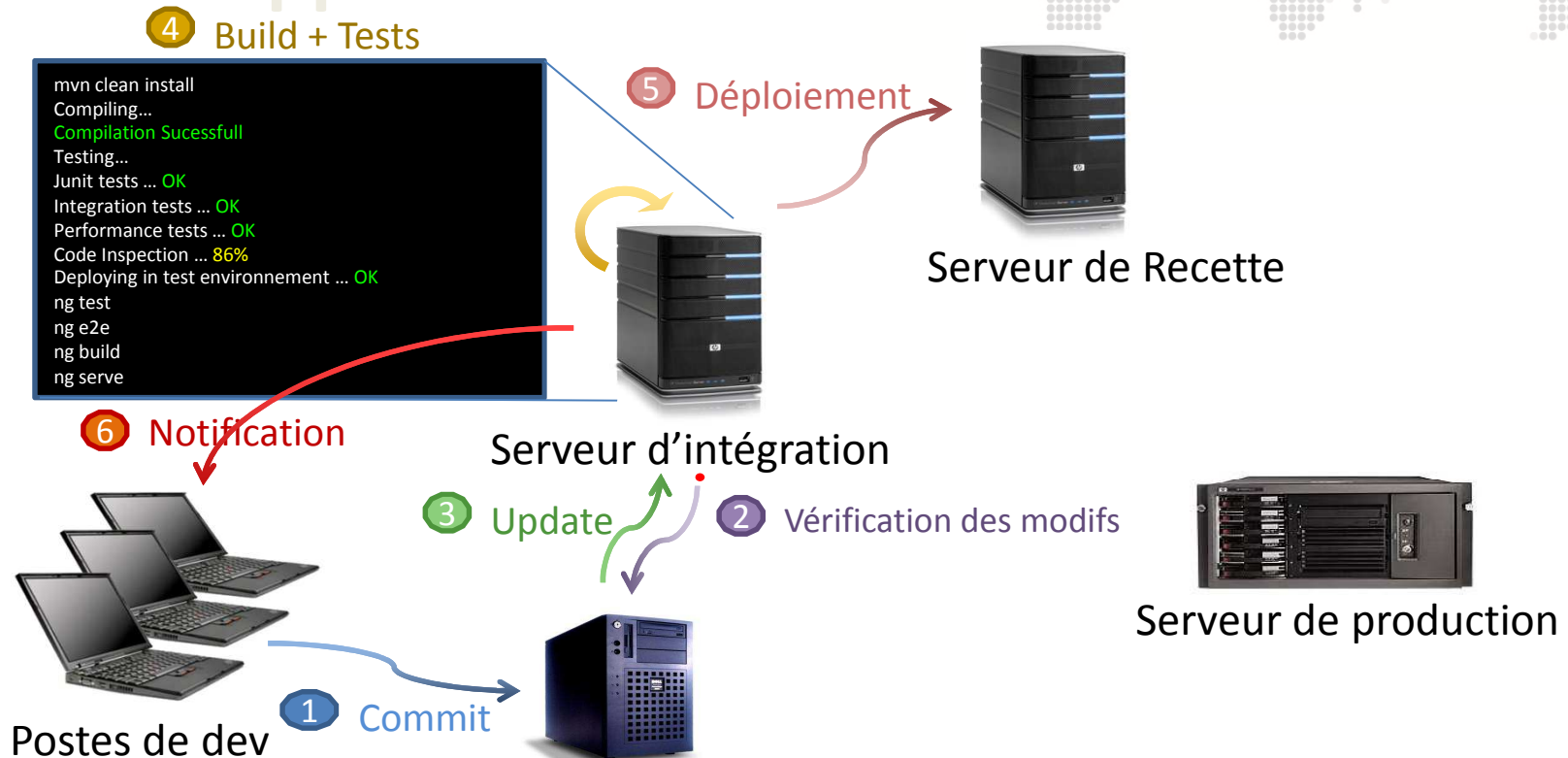




# La Plateforme d'Intégration Continue

→ Cas pratique : Déclenchement périodique

Le développeur soumet une modification.





# La Plateforme d'Intégration Continue

## → Service : Météo

- Ce service offre un tableau de bord pour suivre la Météo de l'intégration du code tout au long du développement.



[Modifier la description](#)

All	Application Blanche	Gestion des Provisions	Gestion des Stocks	RFW-V2	Socle Annuaire Groupe	+
S	W	Tâche ↓	Dernier succès	Dernier échec	Dernière durée	
		<a href="#">AB</a>	21 mn (#10)	N/A	48 s	
		<a href="#">AB_audit</a>	16 h (#5)	N/A	2 mn 48 s	
		<a href="#">F4</a>	21 mn (#11)	N/A	2 mn 38 s	
		<a href="#">F4_audit</a>	16 h (#8)	1 j 22 h (#5)	8 mn 32 s	

### Statut de la dernière intégration

- Statut initial : aucune intégration du projet
- Réussie
- Instable : Tests unitaires en échec.
- Échec : erreur de compilation, etc.

### Taux de réussite des intégrations

- Supérieure à 80%
- Entre 60% et 80%
- Entre 40% et 60%
- Entre 20% et 40%
- Au dessous de 20%







# La Plateforme d'Intégration Continue

## → Service : Tableau de bord « qualité »

- Des indicateurs « clé » pour suivre l'évolution de la qualité du code tout au long des développements :
  - Des indicateurs « macro » à destination des MOE
  - Des indicateurs « détaillés » à destination des développeurs
- Une vérification instantanée et automatique de la conformité du code aux normes, recommandations et bonnes pratiques à établir en interne.

Name	Version	Lines of code	Coverage	Unit test success (%)	Efficiency	Maintainability	Portability	Reliability	Usability	Build time
<a href="#">Socle LDAP du Crédit Agricole</a>	2.0	58,624	<span>2.3%</span>	100.0%	97.7%	<span>89.9%</span>	100.0%	96.1%	<span>74.9%</span>	01:15
<a href="#">Gestion des Provisions</a>	1.1-SNAPSHOT	25,523 ▲	<span>4.2%</span>	<span>24.4%</span>	97.7%	97.3%	100.0%	99.9%	<span>93.7%</span>	01:06
<a href="#">[Root] - Projet racine de RFW</a>	2.1.1	12,735	<span>2.6%</span>	100.0%	99.5%	97.6%	100.0%	99.6%	<span>93.5%</span>	11:17
<a href="#">og</a>	1.1.0-SNAPSHOT	11,594	<span>33.0%</span>	100.0%	99.2%	97.0%	99.9%	<span>92.2%</span>	<span>81.4%</span>	10:35
<a href="#">Root of Application Blanche project</a>	2.4.0	1,821	<span>0.2%</span>	<span>0.0%</span>	100.0%	98.4%	100.0%	99.9%	98.2%	11:03

Alerts feed







# La Plateforme d'Intégration Continue

➔ Service : Tableau de bord « qualité »

Une interface d'administration de la PIC permet de définir les indicateurs clé de la qualité et de préciser les seuils d'alertes :

Complexity /class	is greater than	10	30	Complexité par classe
Complexity /method	is greater than	3	5	Complexité par méthode
Coverage	is less than	50 %	30 %	Couverture par les tests unitaires
Efficiency	is less than	95 %	90 %	Efficacité
Maintainability	is less than	95 %	90 %	Maintenabilité
Portability	is less than	95 %	90 %	Portabilité
Reliability	is less than	95 %	90 %	Fiabilité
Unit test success (%)	is less than	100 %	90 %	Tests unitaires réussis
Usability	is less than	95 %	90 %	Facilité d'utilisation

Indicateurs

Seuil « Vigilance »

Seuil « Critique »

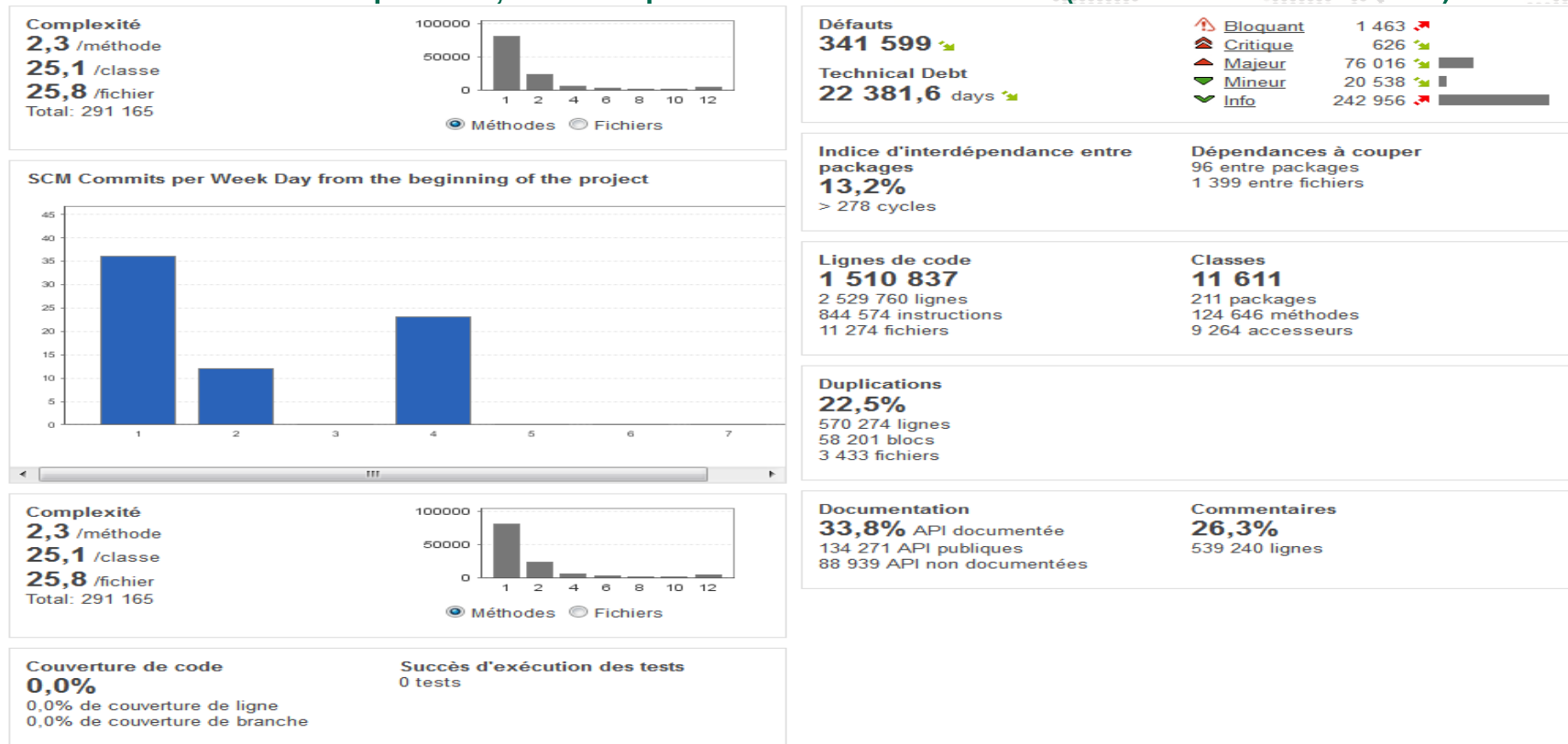




# La Plateforme d'Intégration Continue

➔ Service : Tableau de bord « qualité »

Chaque projet bénéficie d'un tableau de bord avec l'ensemble des indicateurs de qualité, ainsi que leurs tendances (hausse ou baisse)

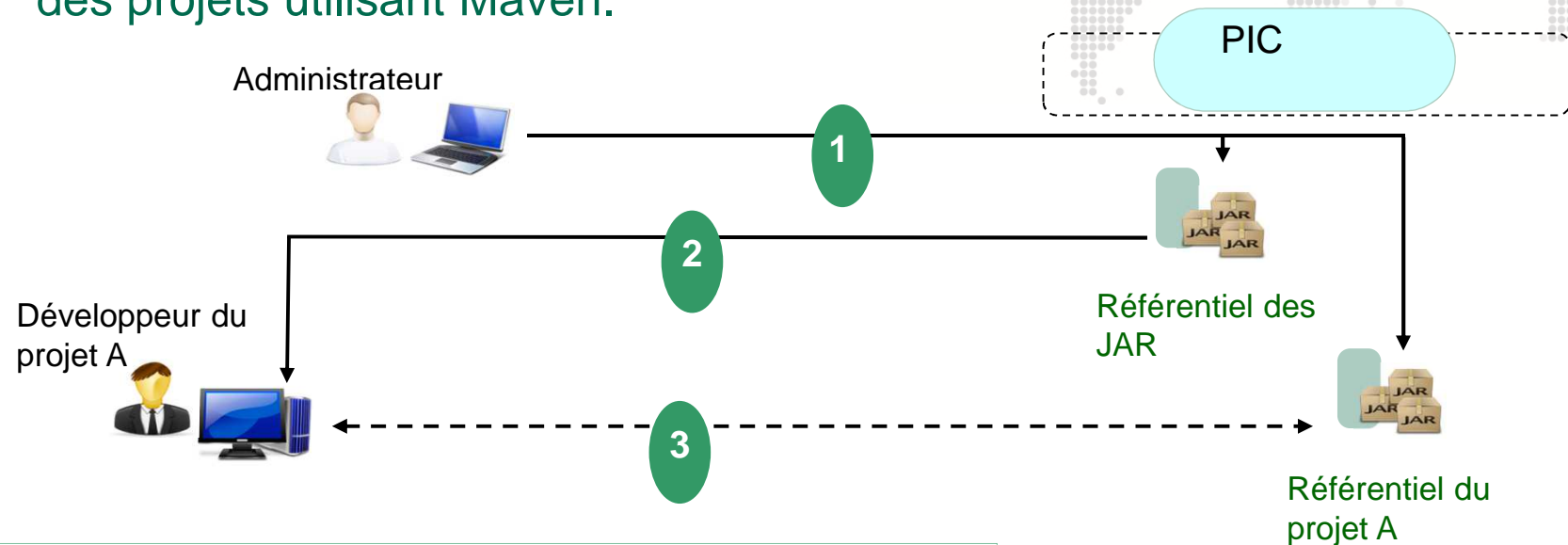




# La Plateforme d'Intégration Continue

## → Service : Référentiel de JAR

- La PIC centralise les référentiels de JAR et facilite ainsi l'intégration des projets utilisant Maven.



- 1) Alimentation des référentiels de JAR
- 2) Récupération automatique des JAR préconisés par la norme JEE
- 3) Partage et récupération des JAR spécifiques au projet





# La Plateforme d'Intégration Continue

## ➔ Service : Référentiel de JAR

■ Ce service est aussi un moteur de recherche permettant de trouver un JAR par plusieurs critères (nom, version, etc.)

The screenshot displays the Nexus Repository Manager interface. At the top, the 'Repositories' tab is active, showing a table of repositories. Below this, the 'SageRepository' is selected, and the 'Browse Storage' tab is active. The left pane shows a tree view of the repository contents, with 'commons-beanutils-1.4.pom' selected. The right pane shows the 'Maven Information' tab, displaying the artifact details and the XML content of the POM file.

Repository	Type	Quality	Format	Policy	Repository Status	Repository Path
Public Repositories	group	ANALYZE	maven2			http://picdev/nexus/content/groups/public
SageRepository	group	ANALYZE	maven2			http://picdev/nexus/content/groups/SageRepository
3rd party	hosted	ANALYZE	maven2	Release	In Service	http://picdev/nexus/content/repositories/thirdparty

**SageRepository**

Browse Index | **Browse Storage** | Configuration

Refresh Path Lookup:

- ch
- cicsj2ee
- classworlds
- com
- commons-beanutils
  - commons-beanutils-1.4
    - commons-beanutils-1.4.pom
    - commons-beanutils-1.4.pom.sha1
    - \_maven.repositories
  - 1.6
  - 1.7.0
- commons-cli
- commons-codec

**Maven Information** | Artifact Information

Group: commons-beanutils  
Artifact: commons-beanutils  
Version: 1.4  
Extension: pom  
XML:

```
<dependency>  
<groupId>commons-beanutils</groupId>  
<artifactId>commons-beanutils</artifactId>  
<version>1.4</version>  
<type>pom</type>  
</dependency>
```

Répositoire dédiée à un projet spécifique

Configuration à « Copier/Coller » par le développeur





# La Plateforme d'Intégration Continue

## ➔ Les 6 bonnes raisons



Historique centralisé **de la qualité de code** des développements.



**Détection automatique et notification rapide** de tout problème d'intégration du code



**Tableau de bord unique** pour suivre l'évolution des indicateurs de la qualité des développements selon un référentiel normalisé



**Adaptation des résultats** aux MOE (vision macro) et aux équipes de développement (vision détaillée)



Bénéfice d'une **Plate-forme mutualisée** (règles, surveillance, habilitation, évolution, etc.)



**Libérer les développeurs** de construire un environnement « *quick-and-dirty* » sur leurs propres postes de travail





# Intégration continue



Construisons ensemble une PIC Foyer

- 1- Création et partage d'un projet Angular2
- 2- Configuration SVN dans Hudson
- 3- Configuration des builds