

Formation Angular 2

Une formation pour les dev front



Présentation

- Formateur/ Auteur : Benoit COSTE

- Expérience

- Développement Mobile, Windows, Web,
 - Chef de projet en solutions web métier



- Révision : 02.11.2016

- Les référence de ce cours

- <https://www.lynda.com/AngularJS-tutorials/AngularJS-2-Essential-Training/422834-2.html>
 - <https://www.lynda.com/Typescript-tutorials/TypeScript-Essential-Training/421807-2.html>
 - Doc officielle : <https://www.typescriptlang.org/>
 - Doc officielle : <https://angular.io/>
 - Autre site : <https://angular-2-training-book.rangle.io/>

Qu'est ce qu'Angular

- Angular est un framework applicatif front end : coté client
- Permet d'organiser son code html/js/css pour garder un code propre
- Première version apparue en 2011
- Le framework est développé par Google



Qu'est ce que TypeScript

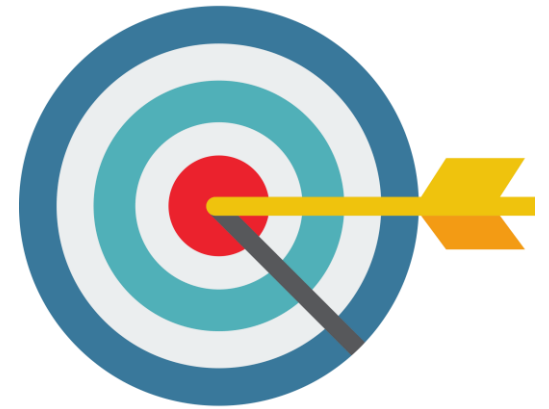
- Langage de programmation
- Développé par Microsoft
- Reposant sur le JavaScript
- Peut supporter les spécifications actuels ECMAScript
- Co-créé par l'inventeur du C# du framework.Net, du j++
- Créé en 2012
- Aquisitions des concepts : C# et JAVA
- OpenSource

TypeScript



Objectif de la formation

- Maîtriser les bases du framework front end Angular 2 JS
- Connaître les contraintes liées au framework
- Savoir donner les avantages et inconvénients de l'utilisation du framework
- Maîtriser les concepts d'architecture Front end avancées



Plan de la formation

Pré requis : html/css/js

1. TypeScript
2. Découverte Angular
3. les components
4. les directives, pipes et methods
5. Forms
6. Injection de dépendance et services
7. Le service http
8. Le routing
9. Un exercice complet

1 jour

1 jour

1 jour

Plan de la formation

1. TypeScript
2. Découverte Angular
3. les components
4. les directives, pipes et methods
5. Forms
6. Injection de dépendance et services
7. Le service http
8. Le routing

Pré requis : html/css/js

Rendre le développement JS plus durable

- Les principaux langages pouvant étendre les fonctionnalités du JS
 - Coffescript
 - Babel js
 - Typescript
- Leur but
 - Simplifier la syntaxe
 - Simplifier le debuggage
 - Simplifier l'utilisation
 - Augmenter la rapidité de développement
 - Ajouter des concepts de programmation

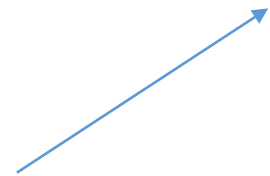
BABEL



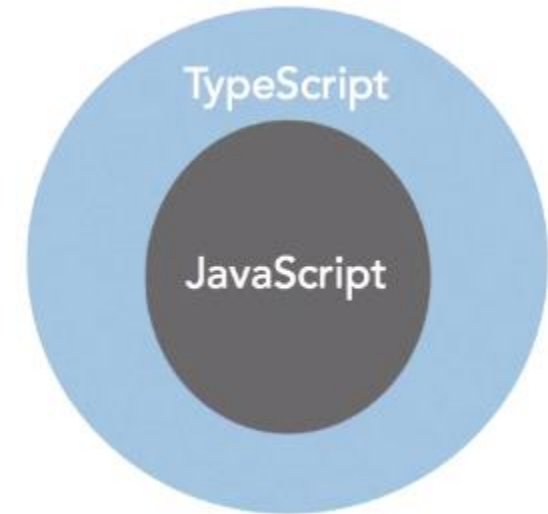
TypeScript : une extension du « objet »

- Présentation : Extension de js
- Un langage static : typé
- Ex:

```
Class Duck
{
  public string name;
  public void quack(){
    //quack
  }
}
```



```
New Duck { name : « bill »}
```





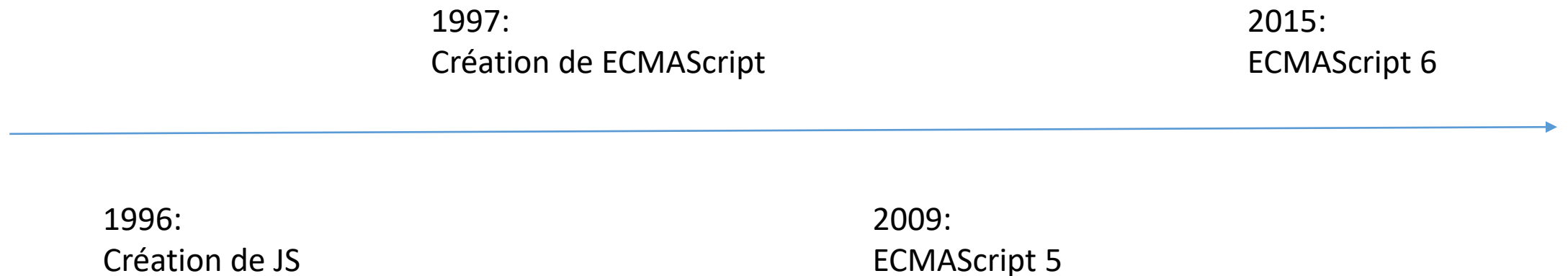
TypeScript – Dynamique vs Static

- Dynamic
 - Types - -
 - Trop tolérant (erreurs)
 - Parfait pour la manipulation du dom
- Static
 - Types ++
 - rigoureux
 - Parfait pour la durabilité d'un projet (si le projet devient gros)
- On parle aussi du langage prototype vs objet



Retour aux sources

- Qu'est ce que Javascript ?
 - Une histoire de standard






TypeScript est un « transpiler »

TypeScript

```
Class Duck {  
  name : string = « bill »;  
  quack(): void {  
    //quack  
  }  
}
```

ECMAScript 5



```
var Duck = ( function(){  
  Function Duck(){  
    This.name = 'bill';  
  }  
  Duck.prototype.quack =  
    Function(){  
      //quack  
    }  
  Return Duck;  
})();
```

Voir la compatibilité ecmascript =

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

TypeScript : installation de l'environnement



- Aller télécharger TypeScript sur le site : <https://www.typescriptlang.org/>
- On peut utiliser node js et npm pour l'installer
 - Installer node js ici : <https://nodejs.org/en/>
 - **Npm install -g typescript**
- On peut utiliser l'editeur de code en ligne et voir comment se transpile typeScript
 - <https://www.typescriptlang.org/play/index.html>
 - DEMO : montrer le le contrôle de typage
- Npm live-server : mini server http, rafraichi la page web dans la navigateur.
 - Pour l'installer : **npm install -g live-server**



TypeScript : C'est l'histoire d'un type

- En js

- `Var mavvariable = 3;`
- `Function test(mavvariable){}`

- En ts

- `Var mavvariable : string = '3';`
- `Function test(mavvariable :string){}`
- Ou avec en type en retour
- `Function test(mavvariable):string{}`



TypeScript : Un IDE pour TypeScript

- Il nous faut
 - Editeur de texte
 - TypeScript compiler
- Quelque éditeur de texte avec des extensions pour TS
 - Sublimetext, notepad++, brackets, textmate, atom, visual studio code
- Ou l'artillerie lourde
 - Visual studio, eclipse, webstorm, ...



TypeScript : Créons nos fichiers

- Créons les fichiers suivants
 - Index.html
 - App.ts

```
• Index
<!doctype html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>Mon app</title>
    <script src="app.js" />
  </head>
  <body>
    Bonjour
  </body>
</html>
```

- App.ts :

Var test: string = 'machaine';

- App.ts
- Le fichier app.js sera généré par la commande :
 - **Tsc app.ts**
- on peut aussi activer la transpilation en temps reel
 - **Tsc -w app.ts**



TypeScript : CLI

- Quelques commandes utiles
 - **Tsc** //lance une compilation unitaire
 - **Tsc -w app.ts**//lance un watcher sur un fichier ts
 - **Tsc -w** //surveille tous les fichiers
- Tsconfig.json
 - Fichier de configuration de la compilation
 - Sera regardé par l'app tsc à chage exécution

- Tsconfig.json

```
{  
  'compilerOptions': {  
    'target': 'es5'  
  }  
}
```



Les fonctionnalités ES6 (2015)

- Les paramètres
 - Paramètres des fonctions peuvent être optionnels ou fixé avec des valeurs par défaut
- Template strings
 - On peut injecter plus facilement des valeurs js dans du html
- Autres

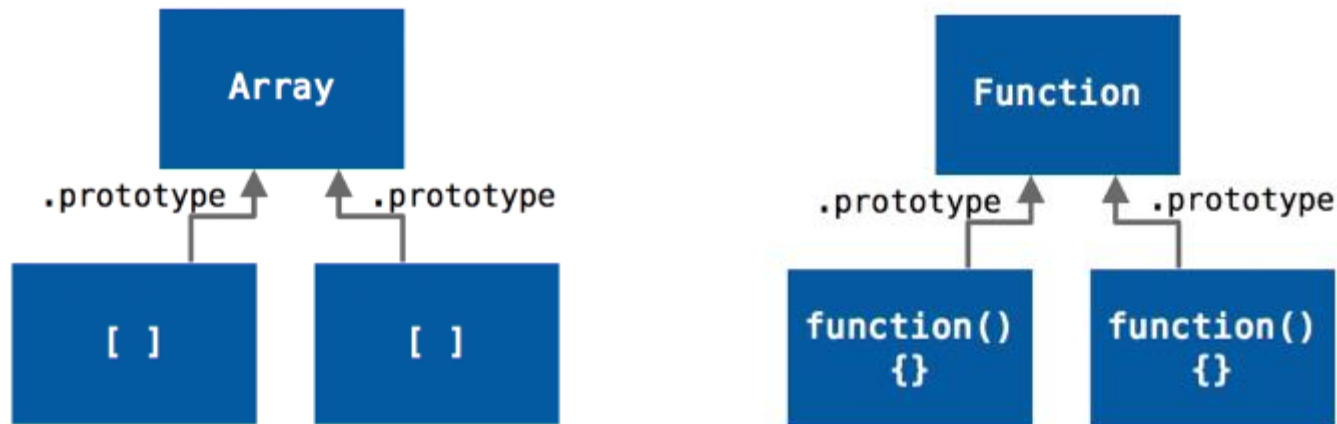


TypeScript : Javascript types

- ECMAScript 5 : Les types
 - Boolean
 - Number
 - String
 - Null / undefined
 - object
- Détail d'un object
 - Key
 - Value
 - Ex:
 - Name : Benoit
 - Age : 29
 - Genre : Homme
- Détail d'un object function
 - La valeur est la fonction : (pas de clé)

TypeScript : Javascript types

- Finalement object, function, array sont des objets !
- Ils héritent d'un prototype en JS
- Certains parlent d'un objet prototype



TypeScript : Un exercice JS – Exo1

- Vous savez faire du JS
 - Créer des objets ingrédient (nom, quantité, prix)
 - Créer un tableau d'ingrédient nommé **recette** dans lequel on ajoutera des ingrédients
 - Afficher la liste des ingrédients de la recette dans la page
 - Créer le formulaire d'ajout d'ingrédient dans la page

Mots clés de l'exercice : array, push, getElementById, json, function

Exercice corrigé : <https://github.com/benitocoste/Formation-Angular-2.0/tree/master/TypeScript/Exo1>

TypeScript : les types en JS

- Nous avons vu l'objet prototype, voici l'annotation en objet dit « littéral », on peut aussi l'appeler json : javascript object notation »
- On ajoute un json dans une variable qu'on peut utiliser dynamiquement

```
//voici un objet literal
{
  nom : "benoit",
  genre : "homme",
  age : 29,
  parler : function(){
    console.log("je peux parler !");
  }
}
```

```
1 //voici un objet literal
2 var moi = {
3   nom : "benoit",
4   genre : "homme",
5   age : 29,
6   parler : function(){
7     console.log("je peux parler !");
8   }
9 }
10 //on peut créer une fonction qui prend un objet en param
11 function faireParler(personne){
12   personne.parler();
13 }
14 //et on peut appeler l'objet
15 faireParler(moi);
16
```

TypeScript : TS, un typage implicite

- Maintenant utilisons le typage de ts !
- Voyons le type inférence ! TS nous dit si un type n'est pas correctement affecté
- Mieux encore, le type peut être implicite
 - TS est donc dynamique et statique

```
//voici un objet literal
var moi = {
  nom : "benoit",
  genre : "homme",
  age : 29,
  parler : function(){
    console.log("je peux parler !");
  }
}

[ts] Type 'number' is not assignable to type 'string'.

var moi: {
  nom: string;
  genre: string;
  age: number;
  parler: () => void;
}

moi.nom = 29 ; //ts nous n'est pas ok ici
```

```
//voici un objet literal
var moi = {
  nom : "benoit",
  genre : "homme",
  age : 29,
  parler : function(){
    console.log("je peux parler !");
  }
}

[ts] Type 'number' is not assignable to type 'string'.

var moi: {
  nom: string;
  genre: string;
  age: number;
  parler: () => void;
}

moi.nom = 29 ; //ts nous n'est pas ok ici
```



TypeScript : faire un typage explicite

- Typer un paramètre d'une fonction :

```
function rouler(kminitia, nbkm :number){}
```

- Type une variable :

```
var totalkm:number = kminitial + nbkm;
```

- Typer le retour d'une fonction

```
function rouler(kminitial, nbkm :number): number{  
    var totalkm:number = kminitial + nbkm;  
}
```

- Les types :
 - string, number, boolean
 - Tableau : string[], number[]
 - Tous les types : any
 - Tableau de tout type : any[]

TypeScript : Zoom sur les paramètres typés



- On a vu comment envoyer fixer un type sur un param de fonction
- On peut faire mieux en ajoutant plusieurs types, et cela avec le pipe
- Je peux ensuite tester le type dans ma fonction pour faire un traitement en fonction.

```
function rouler(kminitial, nbkm :number):
```

```
function rouler(x :(string|number), y: number){
```

```
function rouler(x :(string|number), y: number){  
  if(typeof x === 'string'){  
    console.log("c'est une chaine !");  
  }  
}
```



TypeScript : une surcharge de fonction

- On peut donc avoir des parametres à plusieurs type.
- On peut indiquer une surcharge de fonction à TS, il nous proposera le bon choix de parametre
- Attention, c'est seulement indicatif

```
function rouler(x :(string|number), y: (string|number)){  
  if(typeof x === 'string'){  
    console.log("c'est une chaine !");  
  }  
}
```

```
function rouler(x :string, y: string)  
function rouler(x :number, y: number)  
function rouler(x :(string|number), y: (string|number)){  
  if(typeof x === 'string'){  
    console.log("c'est une chaine !");  
  }  
}
```

```
//on te  
rouler()  
^1/2 rouler(x: string, y: string): any
```

TypeScript : création de nos custom types



- On utilise la notion d'interface
- On peut spécifier le type des propriétés
- On peut spécifier le type de notre variable (vide possible)
- TS nous dira si les assignations sont conformes
- On peut rendre une propriété optionnelle avec ?
- Enfin on peut créer une interface avec des methodes

```
interface Piaf{  
  couleur;  
  plume;  
}
```

```
1 interface Piaf{  
2   couleur: string;  
3   plume:string;  
4   poid: number;  
5 }
```

```
6  
7 var colibri: Piaf;
```

```
interface ListePiaf{  
  ajouter(piaf: Piaf): Piaf;  
  supprimer(piaf: Piaf): Piaf;  
  recupTout(): Piaf[];  
  recupParId(piafId: number): Piaf;  
}
```

```
interface Piaf{  
  couleur: string;  
  plume?:string;  
  poid?: number;  
}  
  
var colibri: Piaf = {  
  couleur: "noir"  
}
```



TypeScript : utilisation des interfaces

- Les fonction sont aussi des object en js
- Une interface peut définir les propriétés d'une fonction, et le paramètres qu'elle prend
- TS nous dit s'il manque des propriétés
- TS nous dit s'il y a une erreur de typage depuis l'interface

```
interface monModelFunction{  
    //on peut définir des propriétés de fonction  
    prop1: number;  
}  
  
var mafunction = <monModelFunction>function(param){  
    //ts me dit qu'il manque l'implémentation de prop1  
}  
  
mafunction.prop1 = "RET";  
//ts me dit que prop1 devrait être un number
```

```
1 interface monModelFunction{  
2     //on peut définir des propriétés de fonction  
3     (param: number);  
4     prop1: number;  
5 }
```

TypeScript : extension des interfaces

- Des interfaces dans des interfaces
- Mon piaf aurait des ailes ?
- On crée une interface : ailes
- Celle-ci permet de typer la propriété aile dans mon piaf !

```
1 interface Piaf{
2     nom:string;
3     couleur: string;
4     ailes: aile;
5 }
6 interface aile{
7     longueur: number;
8 }
9 var aileoisillon = { longueur: 5}
10 var oisillon: Piaf = {
11     nom : "monoiseau",
12     couleur : "jaunes",
13     ailes : {
14         longueur : 4
15     }
16 }
```



TypeScript : constantes et énumération

- Les custom type peuvent être de 3 types
 - Interfaces
 - Classes
 - Enums
- Voici comment utiliser les enums
 - Un enums ne spécifie qu'une structure de données !
 - Donc seulement les propriétés
- L'enum permet de gérer facilement un état sur un enregistrement

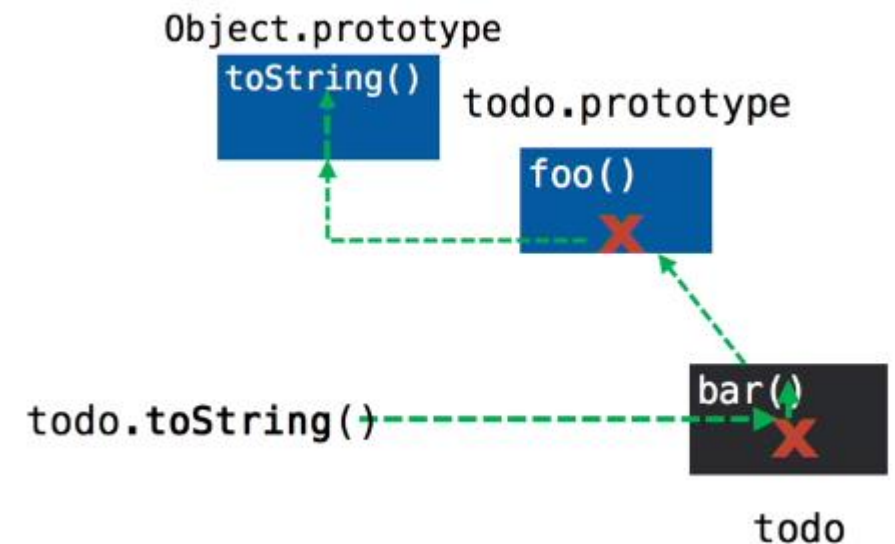
```
1 interface Piaf{
2     provenance : number;
3     nom : string;
4 }
5
6 enum laprovenance{
7     Europe = 1,
8     Asie = 2,
9     Amerique = 3,
10    Autre = 4
11 }
12
13 function verifPiafEurope(monpiaf: Piaf){
14     if (monpiaf.provenance = laprovenance.Europe){
15         console.log("mon piaf vient d'europe !");
16     }
17 }
```

TypeScript : Les classes et les prototypes



- Dans la console d'un navigateur :
 - Object.prototype

```
1 function TodoService(){  
2   this.todo = [];  
3 }  
4  
5 TodoService.prototype.getAll = function(){  
6   return this.todos;  
7 }  
8  
9 var service = new TodoService();  
10 service.getAll();
```





TypeScript : Définition d'une classe

- Une classe TS sera transpilée en prototype en ECMAScript5
- ECMAScript 6 ajoute la notion de classe !
- Une classe peut avoir des propriétés et des méthodes
- Il est aussi possible de créer les propriétés directement dans les paramètres du constructeur

```
class Piaf{  
    //définition de propriétés  
    nom: number;  
    couleur: string;  
    constructor(pcouleur){  
        //et voila un constructeur  
        this.couleur = pcouleur;  
    }  
    vole(){  
        //mon oiseau vole  
    }  
}
```

```
class Piaf{  
    //définition de propriétés  
    nom: number;  
    couleur: string;  
    constructor(private agemoyen: number){  
        //et voila un constructeur  
        this.couleur;  
        this.agemoyen; //c'est fou  
    }  
    vole(){  
        //mon oiseau vole  
    }  
}
```


TypeScript : les membres statics

- Utiliser une sorte de constante n'importe où à n'importe quel moment dans l'application
- En JS on aurait créé une variable dans le scope global
- En TS codera proprement
- On peut avoir une propriété static ou une méthode static
- On peut appeler la méthode via le nom de la classe à tout moment

```
class Piaf{
    static nombredepates: number = 2;
}
```

```
class Piaf{
    static nombredepates: number = 2;
    static afficheNombrePates(){
        console.log(this.nombredepates);
    }
}
```

```
static afficheNombrePates(){
    console.log(this.nombredepates);
}
roucouler(){
    var nbpa = Piaf.afficheNombrePates();
}
```

TypeScript : les getter et setter

- Les getter et setter permettent d'accéder aux propriétés
- Cela permet de sécuriser le code
- Peut être aussi utilisé pour un contrôle
- En TS il ne faut pas créer la propriété, les getter et setter se contentent de le faire

```

2
3 class Piaf{
4     couleur: string;
5     vivant: boolean;
6     get nom(){
7         return this.nom;
8     }
9     set nom(pnom){
10        if(this.vivant){
11            this.nom = pnom;
12
13        }else{
14            console.log("on ne peut plus lui changer de nom");
15        }
16    }
17 }
18 var monpiaf = new Piaf();
19 monpiaf.nom = "coucou";
20

```

```

class Piaf{
    couleur: string;
    get nom(){
        return this.nom;
    }
    set nom(pnom){
        this.nom = pnom;
    }
}
var monpiaf = new Piaf();
monpiaf.nom = "coucou";

```



TypeScript : héritage de class

- Permet à une class d'hériter des propriétés et méthode d'une class mère
- Ex : class Piaf extend Oiseau

```
class Oiseau{  
    plume: string;  
    static nombredepates: number = 2;  
}  
class Piaf extends Oiseau{  
  
}  
var monpiaf = new Piaf();  
monpiaf.plume = "plume longue";
```

TypeScript : la class abstract

- Abstract indique que la classe ne pourra pas elle-même être instanciée.
- Il faudra instancier la classe dérivée
- Les méthodes abstract devront aussi être surchargées dans les classes dérivées

```
abstract class Oiseau{  
    plume: string;  
    static nombredepates: number = 2;  
    abstract senvoler();  
}  
  
class Piaf extends Oiseau{  
  
}  
  
var monpiaf = new Piaf();  
monpiaf.plume = "plume longue";
```

TypeScript : visibilité des accesseurs

- Il est possible de rendre les propriétés ou méthodes privées pour les protéger
- Par défaut, tout est pu
- Si seule la class peut accéder aux propriétés et méthodes, alors on utilisera le mot clé **private**
- Si on veut que les classes dérivées puissent accéder aux propriétés et méthodes, on utilisera le mot clé **protected**
- Une bonne pratique est de préfixer les variable **private** par des underscores

```

1  class Oiseau{
2      plume: string;
3      static nombredepates: number = 2;
4  }
5  class Piaf extends Oiseau{
6      private poids:number;
7  }
8  var monpiaf = new Piaf();
9  monpiaf.plume = "plume longue";

```

```

1  class Oiseau{
2      protected plume: string;
3      static nombredepates: number = 2;
4  }
5  class Piaf extends Oiseau{
6      private poids:number;
7      setPlume(pplume){
8          this.plume = pplume;
9      }
10 }
11 var monpiaf = new Piaf();
12 monpiaf.setPlume("plume longue");
13

```



TypeScript : Exercices

- Vous savez faire du JS
 - Créer des objets ts ingrédient (interface, classes, getter, setter)
 - Créer un tableau d'ingrédient dans lequel on ajoutera nos ingrédients
 - Afficher le liste des ingrédients
 - Créer le formulaire d'ajout d'ingrédients
 - Transformer le tableau d'ingrédients en objet recette, un recette possede (string:nom, string:temps;array ingredients)
 - Afficher toutes les informations
 - Faites un sorte que les nom/quantité/prix soient obligatoires
 - Créer des méthode sur les ingrédients
 - Méthode qui divise le prix par 2
 - Méthode qui multiplie la quantité par x

TypeScript : Les génériques

- Une classe générique peut être réutiliser a tout moment
- Exemple avec GestionPoids
 - Peut s'appliquer à
 - Un piaf
 - Une voiture
 - Une personne
 - Un ingrédient
- Autre exemple
 - On peut créer une classe de Serialisation
 - Avec une méthode de sérialisation JSON
 - Avec une méthode de sérialisation CSV
 - Avec une méthode de sérialisation XML

TypeScript : Les modules

- Pourquoi utiliser des modules
 - Il est trop facile d'écrire du code dans le global namespaces (scope) .
 - Cela peut engendrer des conflits entre les librairies / autres
 - La notion de dépendance en JS n'existe pas
 - Le débogage était difficile et se fait pas à pas.
- Les modules
 - Permettent de déclarer explicitement des dépendances
 - Produire des composants propres avec des limites claires
- Il existe plusieurs possibilités de design pattern en js
 - Module Pattern
 - Namespaces
 - ECMAScript 2015 module

TypeScript : les namespaces

- Evite les conflits / collisions
- Permet d'organiser la structure d'une app. Il faut ajouter **export**
- On peut utiliser un alias avec **import Model = MonApp.Model;**

```
namespace MonApp.Model{ //et voila
    class Piaf{
        nom : string;
        couleur : string;
    }
}
//un autre namespace
namespace DataAccess{
    class Piaf{
        nom : string;
        couleur : string;
    }
}
```

```
namespace MonApp.Model{ //et voila
    export class Oiseau{
        nom : string;
        couleur : string;
    }
}
//un autre namespace
namespace DataAccess{
    export class Piaf extends Oiseau{
        nom : string;
        couleur : string;
    }
}
```

```
namespace MonApp.Model{ //et voila
    export class Oiseau{
        nom : string;
        couleur : string;
    }
}
//un autre namespace
namespace DataAccess{
    class Piaf extends MonApp.Model.Oiseau{
        nom : string;
        couleur : string;
    }
}
```

TypeScript : module interne et module externe



- Les namespaces font références à des modules internes
 - Ils permettent de faire travailler des modules ensembles !
- On peut créer plusieurs namespaces dans le même fichier
- Les modules externes sont gérés avec la notion de fichier
- On utilise **@import** ou **require** pour importer un module externe

TypeScript : importer un module

- Voici comment importer un module externe
- Il faut ajouter un paramètre de configuration dans le fichier
 - **Tsconfig.json**
- 2 syntaxes sont possibles pour importer les modules
- On ne spécifie pas l'extension de l'import
- Pour simplifier le code, on peut créer un alias de notre import

```
{
  "compilerOptions": {
    "target": "es5",
    "module": ""
  }
}
```

amd
commonjs
es2015
es6
none
system
umd

```
app.ts  app2.ts  tsconfig.json
1  import App = require('./app');
```

```
ts  app2.ts  tsconfig.json
1  import App = require('./app');
2  //j'ai une classe app.personne
3  import Personne = app.personne;
4  //et maintenant je peux l'utiliser
5  var Benoit = new Personne();
6
```

TypeScript : importer des modules en ECMAScript 2015



- Voici une autre façon d'importer un module
- On peut importer des modules externes et créer des alias sur la même ligne.

```
app2.ts  •  tsconfig.json
//syntaxe ECMAScript2015
import * as App from './app';
//et voici un alias de ma classe personne
import Personne = app.personne;
```

```
pp.ts  app2.ts  •  tsconfig.json
1 //syntaxe ECMAScript2015
2 import { Personne as PersonneObj, Voiture} as App from './app';
3
4 var benoit = new PersonneObj();
5
```

TypeScript : Débogage de TS



- Debuguer du TS dans le navigateur
 - Le bug sur le js ne sera pas sur la ligne du ts
- SourceMaps : Permet de trouver la ligne en ts. On trouve le source map dans le debugger chrome.
 - Il faut ajouter SourceMaps dans le fichier de conf de TS
 - En compilant le ts, ts crée un fichier.map, qui donne les références entre le ts et le js au navigateur
 - S'il y a un problème, le navigateur ouvre le fichier ts !

```
.ts    app2.ts    tsconfig.json
1  {
2      "compilerOptions": {
3          "target": "es5",
4          "module": "system",
5          "sourceMap": true
6      }
7  }
```

TypeScript : Les décorateurs

- Les decorateur permettent de modifier le comportements des classes, méthodes, propriété et paramètre.
- Permet de réduire le code
- Rendre le code plus durable, plus lisible
- Il faut ajouter le param `experimentalDecorators : true` dans le `tsconfig.json`
- Exemple : on veut calculer la surface d'un rectangle. On peut utiliser un décorateur pour modifier la méthode avant ou après pour vérifier si le rectangle est un carré, autres...



TypeScript: Exercice final

- Créer un namespace avec une classe recette
- Créer un namespace avec une classe ingrédient
- Créer les formulaires d'ajout d'ingrédient dans une recette.
 - 1 recette peut avoir plusieurs ingredients
- **Fin journée 1**

Angular 2



- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
2. Découverte d'Angular
3. les components
4. les directives, pipes et methods
5. Forms
6. Injection de dépendance et services
7. Le service http
8. Le routing



Angular : le différences entre A1 et A2

- Rapidité -> angular 2 est plus rapide
- Components – la notion principale est le component et non les controllers et le scope que l'on utilisait en v1
- Directives simples – Créer ses propres directive est plus simple
- Intuitive DataBinding – Une syntaxe plus facile à approcher
- Service as a class – les service sont maintenant des class ! (merci typescript)
- Autres améliorations.
- En bref : **faster, better, stronger**



Angular : reposant sur du js

- Javascript: le langage officiel des navigateurs
- 2 façons de travailler avec la version 2 :
 - Babel :
 - TypeScript :

Avantages :

- Structurer son code
- Développer en objet
- Débuguer facilement
- Autres ...

Angular : TypeScript, la base d'angular 2



- TypeScript, est développé par Microsoft pour étendre la puissance du js (souvent sous estimée)
- La librairie angular 2 est développée en TypeScript
- TypeScript peut être transpilé par
 - Un serveur
 - Plus rapide pour l'utilisateur
 - Un client
 - plus lent pour l'utilisateur

Angular : Le fonctionnement



1. Angular manipule le DOM HTML
2. Angular 2 est basé sur l'identification de component sur des blocs html
3. Les components peuvent être imbriqués
 1. Les components sont identifié dans des balises
4. Les components sont interprétés par angular
5. Un component est une classe typeScript dans laquelle il y aura la logique
6. Ex : MonAppComponent
 1. Class MonApp
 1. Method supprimeElement()

```
<monappcomponent>
<section>
<div>
  <h1>{{MonApp.nom}}</h1>
  <button>(click)="supprimeElement()"</button>
</div>
<view-note></view-note>
</section>
</monappcomponent>
```

MonAppComponent

MonElementComponent

Selector: 'view-note'



Angular : Directives et pipes

- Un component est une directive avec un template
- Les directives transforment le dom
- 2 types de directives
 - Structural
 - Modifie le layout
 - Attribute
 - Change le comportement des éléments ou l'apparence
- Une directive se configure avec un selector
- Il existe plusieurs expressions : ngIf, ngFor, routerLink
- Angular pipe : | permet de prendre une chaîne dans un tableau pour la manipuler. Ex passer une chaîne en majuscule

```
app.ts  app2.ts  tsconfig.json  tuto.html
1  <ul>
2    <li ngFor="item' of items">Mon élément</li>
3  </ul>
```

```
@Decorator({
  selector : 'monElement'
})

<div monElement>
  
</div>
```



Angular : DataBinding

- Interpolation : {{ piaf.nom }}
- Angular interprète tout ce qu'il y a dedans
- Système de template
- Angular manipule de dom suivant le model !
- Plusieurs éléments de gestion de données
 - Les expressions
 - Les statements
 - Value binding
 - Event binding
 - Expression operator
- Angular dispose d'outils pour gérer
 - La liaison en affichage
 - La liaison des formulaires

```
adresse du contact : <input #contact>
  Adresse : <input #adresse>
  <button (click)="adresse.adressecomplete = contact.adresse">
    Copier l'adresse du contact ici
  </button>
```

Angular : Dependency injection



- La création de dépendance ou de module permet de compartimenter son application
- Les constructeur des components prennent les modules en paramètres

Component
Formulaire

Component
Service HTTP

Component
Router

```
Constructor(formulaireContact: formulaireContact) {}
```



Angular : Services

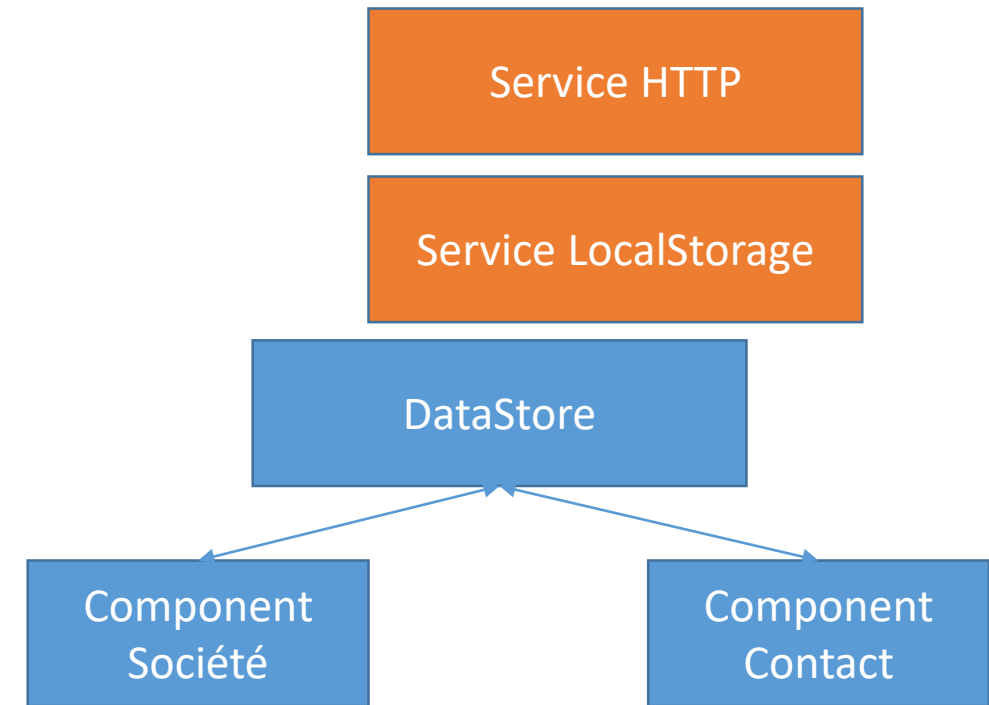
- C'est une référence à une classe / methodes
- Exemple : Mon Service
 - recupContact()
 - ajouteContact()
- Si la logique n'est pas dans les components, on la met dans les services
- Il faut référencer le service via l'injection de dépendances
- Evite du code doublon
- Rend le code modulable

```
1
2 //mon service
3 class MonService{
4     getById(id){
5     }}
```


Angular : Data persistence



- Une classe peut être persisté dans le DataStore
- A nous de choisir ce que l'on veut garder en mémoire
- Un component société peut récupérer la liste des contacts
- Un component contact peut récupérer les infos d'une société
- On peut dire au DataStoreService d'utiliser le local Storage Service
- On peut dire au DataStore Service d'utiliser le service http pour consommer des api





Angular : Rooting

- Single Page application
- Donner une indication dans l'url
- Les elements du module de routage
 - La configuration des routes
 - routeParams (permet d'envoyer des variables dans les urls)
 - Directive de rooting links
 - Créer du lien entre les pages
 - Directive de rooting outlets
 - Savoir ou chercher un template de page
 - Directive de rooting events
 - Pour déclencher des actions sur des urls
- Angular gere
 - L'url affiché dans le navigateur
 - L'historique de navigation
 - Donne l'impression de naviguer sur des pages server



Angular : Exercice

- Installer git sur la machine
- Git clone sur repo
 - `git clone https://github.com/angular/quickstart my-proj`
- Installer nodejs
- Npm install
- Npm start
- Modifier son premier component
 - <https://angular.io/docs/ts/latest/quickstart.html>



Angular : Exercice

- Créer un projet angular 2
 - Utiliser la doc pour créer un projet angular 2
 - Utiliser un hello world
- <https://angular.io/docs/ts/latest/quickstart.html>

Angular 2



- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
- ~~2. Le fonctionnement~~
3. les composants
4. les directives, pipes et methods
5. Forms
6. Injection de dépendance et services
7. Le service http
8. Le routing

Angular Components : créer un component



- Annotation components : permet de définir une classe component
 - Ex : @Component()
 - Ajouter les propriétés
 - Selector
 - Template
 - Ajouter la classe avec un export
- Ajouter notre component dans le bootstrap dans le fichier **main.ts**
- **Bootstrap est une fonction de angular2/platform/browser module**

```
//on importe le namespace
import {Component} from 'angular2/core';

//on peut maintenant l'utiliser
@Component({
  selector: 'app',
  template: '<h1>Mon titre</h1>'
  //templateUrl
})

export class AppComponent {
}
```

```
//voici le bootstrap
import {bootstrap} from 'angular2/platform/browser';
import {AppComponent} from './app.component';

bootstrap(AppComponent);
```



Angular : Sélecteur de component

- Le selector du décorateur
@Component doit être présent dans le dom
- Avec le selector, angular lie notre component à l'élément html
- 1component = 1 balise
- Il faut un tiret dans le nom du selecteur pour passer le w3C validator
- Dans l'exemple my-app

```
ichage Atteindre Aide  
onent.ts x main.ts  
import { Component } from '  
@Component({  
  selector: 'my-app',  
  template: '<h1>Ma ppage
```

```
<body>  
  <my-app>Loading...|  
  </my-app>  
</body>  
</html>
```

Angular : Les templates de component



- Le component prend un deuxième parametre : template
 - Template : le html est passé en chaine
 - templateUrl : le html est passé dans un fichier
- Il est possible d'utiliser les back tick ` pour ajouter facilement du ts dans le html ou multiligne
- On peut créer un fichier de Template :
ex: app.component.html

```
@Component({  
  selector: 'my-app',  
  template: '<h1>Ma premiere app</h1><p>Je peux ajouter ma description</p>'  
})  
export class AppComponent {  
  
}
```

```
@Component({  
  selector: 'my-app',  
  template: `  
    <h1>Ma premiere app</h1>  
    <p>Je peux ajouter ma description</p>  
  `,  
})  
export class AppComponent {  
  
}
```

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




Angular : Styliser les component

- 2 façon d'ajouter du style
 - Ajouter la propriété style
 - Ajouter la propriété style url
- Pour le style url : créons le fichier app.component.css ajoutons le.
- Le CSS est limité au scope du component,
- Si on veut ajouter un css pour tous le site il faudra l'ajouter dans index.html

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

@Component({
  selector: 'my-app',
  template: '<h1>coucou</h1>',
  styles : [
    h1 {
      color : red;
    }
  ]
})
export class AppComponent {
```

```
@Component({
  selector: 'my-app',
  template: '<h1>coucou</h1>',
  styleUrls: ['app.component.css']
})
export class AppComponent {
```

Angular : Imbrication de components



- Ajoutons un ContactComponent dans notre AppComponent
- Créons le fichier contact.component.ts
 - Eventuellement les fichiers
 - Contact.component.html pour le template
 - Contact.component.css pour le style
- Ajouter l'import
 - import { Component } from '@angular/core';
 - Ajout de la propriété directives
- Il faut ajouter le component dans le ngmodule du fichier **app.module.ts**

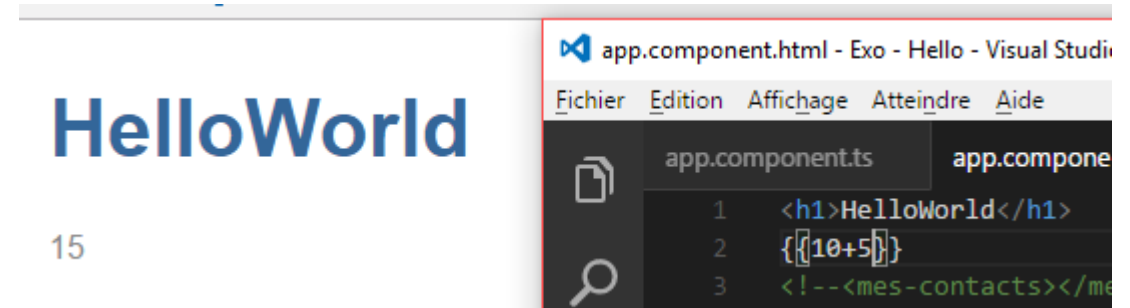
```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'mes-contacts',
5   template: '<h2>Contact</h2>'
6 })
7 export class ContactComponent {}
8
9
10
```

```
4 import { AppComponent } from './app.component';
5 import { ContactComponent } from './contact.component';
6
7 @NgModule({
8   imports: [ BrowserModule ],
9   declarations: [ AppComponent, ContactComponent ],
10  bootstrap: [ AppComponent ]
11 })
12 export class AppModule { }
13
```

Angular : La puissance de l'interpolation



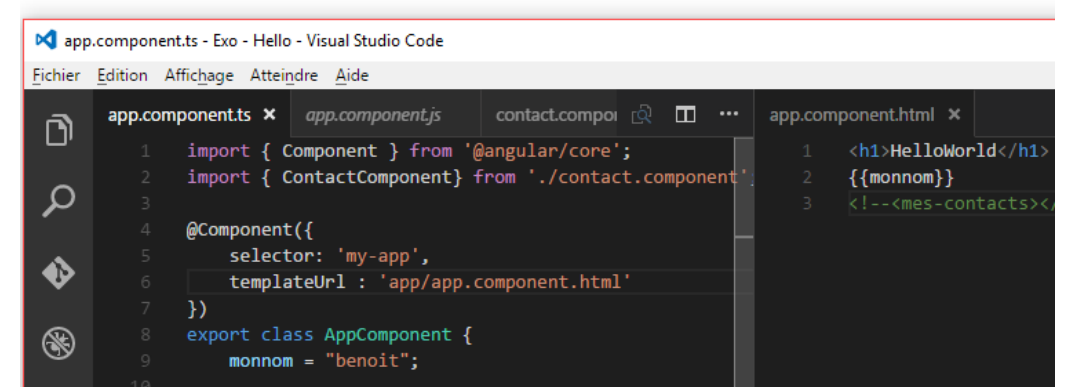
- Interpolation
- Permet d'afficher des données dans le template
- Permet d'afficher des données de notre classe

A screenshot of the Visual Studio Code editor. The top part shows a preview of a web browser displaying "HelloWorld" in a large blue font. Below the preview, the file explorer shows "app.component.html" and "app.component.ts". The "app.component.html" file is open, showing the following code:

```
1 <h1>HelloWorld</h1>
2 {{10+5}}
3 <!--<mes-contacts></me
```

HelloWorld

benoit

A screenshot of the Visual Studio Code editor showing the TypeScript code for the application. The file explorer shows "app.component.ts", "app.component.js", "contact.component", and "app.component.html". The "app.component.ts" file is open, showing the following code:

```
1 import { Component } from '@angular/core';
2 import { ContactComponent } from './contact.component';
3
4 @Component({
5   selector: 'my-app',
6   templateUrl: 'app/app.component.html'
7 })
8 export class AppComponent {
9   monnom = "benoit";
10
```



Angular : L'évent binding

- Lier une méthode de notre classe sur un élément de notre Template

HelloWorld

benoit

supprimer

The screenshot shows the Visual Studio Code interface with the following files and code:

Files in the Explorer:

- ct.component.ts
- app.component.html
- index.html
- app.component.ts

Code in app.component.html:

```
1 <h1>HelloWorld</h1>
2 <p>{{monnom}}</p>
3 <!--Equivalen-->
4 <a (click)="supprimer()">supprimer</a>
5
```

Code in app.component.ts:

```
1 import { Component } from '@angular/core'
2 import { ContactComponent } from './contact'
3
4 @Component({
5   selector: 'my-app',
6   templateUrl: 'app/app.component.html'
7 })
8 export class AppComponent {
9   monnom = "benoit";
10  supprimer(){
11    alert("je supprime");
12  }
13}
```

Angular : Envoyer une donnée avec l'input decorator



- Ajouter le Input dans l'import @angular2
- On ajoute le decorator @input dans une methode de contact.component.ts
- On ajoute la liaison du décorator dans le template app.component.html
- On crée un l'objet qu'on veut partager dans app.component.ts

```
<h1>HelloWorld</h1>
<p>{{monnom}}</p>
<!--Equivalen-->
<mes-contacts [contactElement]="monContactPartage"></mes-contacts>
<a (click)="supprimer()">supprimer</a>
```

```
app.component.ts • contact.component.ts
1 import { Component, Input, Output } from '@angular/core';
2 import { ContactComponent } from './contact.component';
3
4 @Component({
5   selector: 'my-app',
6   templateUrl: 'app/app.component.html'
7 })
8 export class AppComponent {
9   monnom = "benoit";
10  supprimer(){
11    alert("je supprime");
12  }
13  monElementPartage = {
14    id: 1,
15    nom : "coste",
16    prenom : "benoit"
17  }
18
19 }
20
```

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'mes-contacts',
  template: '<h3>{{contactElement.nom}}</h3>'
})
export class ContactComponent {
  @Input() contactElement:any;
}
```

Angular : Envoyer une donnée avec l'output decorator



- Il faut importer la classe output depuis le core angular

On peut faire une action sur une donnée qui change

@input et @output nous permettent de partager des données entre les composants

```
1 import { Component, Input, Output } from '@angular/core';  
2 import { ContactComponent } from './contact.component';  
3  
4 @Component({
```



Angular Component : Exercice

- Créer un component root
- Créer un component intégré « recette »
- Créer un component intégré « ingrédients »
- Afficher 2 recettes
- Mettre en forme la recette

Angular 2



- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
- ~~2. Le fonctionnement~~
- ~~3. les composants~~
4. les directives, pipes et methods
5. Forms
6. Injection de dépendance et services
7. Le service http
8. Le routing



Angular : Directives et Pipes

- Directive ngIf
- Change le layout html suivant la valeur
- Affiche l'élément dom imbriqué de façon conditionnelle
- Se positionne dans le template
- Valeur false ou null pour être désactivé
- Peut aussi se

```
<div *ngIf="contact.actif">
  <p>Nom : {{contact.nom}}</p>
  <p>Prénom : {{contact.prenom}}</p>
</div>
```

Angular : Directives et Pipes



- Directive ngFor
- Permet de faire une boucle dans un tableau
- Il faut un tableau dans le component
- on pourrait dédier une list d'élément dans un component
- Le ngFor s'ajoute dans un template
- On peut récupérer l'index du tableau

```
1 export class ContactComponent{
2     tabContact = [
3         {
4             id: 1,
5             nom : "nom",
6             prenom : "prenom"
7         },
8         {
9             id: 2,
10            nom : "nom",
11            prenom : "prenom"
12        },
13        {
14            id: 3,
15            nom : "nom",
16            prenom : "prenom"
17        },
18        {
19            id: 4,
20            nom : "nom",
21            prenom : "prenom"
22        }
23    ]
24 }
```

```
component.html • test.ts
<div *ngFor="#contact of tabContact; #idx = index">
    {{contact.nom}} - {{contact.prenom}}
    l'index est : {{idx}}
    <input [(ng-model)]="contact.prenom">
</div>
```



Angular : Directives et Pipes

- ngSwitch :
 - Permet d'afficher des valeurs suivant des expressions
 - Exemple : afficher une promo de 10 ou 20 pourcents si la valeur de la promo = 10 ou 20
 - On test une expression avec ngSwitch, on affiche un élément du dom si l'expression vaut vrai de *ngSwitchCase

```
<container-element [ngSwitch]="switch_expression">
  <some-element *ngSwitchCase="match_expression_1">...</some-element>
  <some-element *ngSwitchCase="match_expression_2">...</some-element>
  <some-other-element *ngSwitchCase="match_expression_3">...</some-other-element>
  <ng-container *ngSwitchCase="match_expression_3">
    <!-- use a ng-container to group multiple root nodes -->
    <inner-element></inner-element>
    <inner-other-element></inner-other-element>
  </ng-container>
  <some-element *ngSwitchDefault>...</p>
</container-element>
```



Angular : Directives et Pipes

- ngClass

- ngClass permet d'activer d'affecter ou non des class sur des éléments du dom,
 - Les valeurs sont manipulables en angular

```
<div [ngClass]="['bold-text', 'green']">array of classes</div>  
<div [ngClass]=" 'italic-text blue' ">string of classes</div>  
<div [ngClass]="{'small-text': true, 'red': true}">object of classes</div>
```

- ngStyle

- ngStyle permet de manipuler le style css avec angular
 - Ainsi, on peut lier du style sur des models de notre class component

```
<div [ngStyle]="{'color': color, 'font-size': size, 'font-weight': 'bold'}">  
</div>  
  
<input [(ngModel)]="color" />  
<button (click)="size = size + 1">+</button>  
<button (click)="size = size - 1">-</button>
```

Angular : Directives et Pipes



- Les events click
 - S'utilise : (click)='maMethode() '
 - La methode doit se trouver dans la classe du component

```
selector: 'my-app',  
template: `  
  <h1>test mon event</h1>  
  <button (click)="maMethode(hero)">test</button>  
`  
})  
export class AppComponent {  
  
  maMethode(moningredient: Ingredient): void {  
    alert("test ok");  
  }  
}
```

Angular : Directives et Pipes



- Les pipes
 - Il faut importer la classe **Pipe**
 - Un pipe prend une données en entrée et en fournis une en sortie
 - Ex : on créé un date que l'on veut rendre lisible pour les humains
 - La syntaxe : **{{ mvariable | formatdesortie }}**
- Il est possible d'utiliser des formats de sortie existants
 - UpperCasePipe, LowerCasePipe, PercentPipe
- Ces derniers sont souvent paramétrables

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

```
export class HeroBirthdayComponent {  
    birthday = new Date(1988, 3, 15); // April 15,  
    1988  
}
```

```
<p>The hero's birthday is {{ birthday | date }}  
</p>
```

```
<p>The hero's birthday is {{ birthday |  
date:"MM/dd/yy" }} </p>
```

Angular : Directives et Pipes



- Pipe suite
 - Il est possible faire succéder les pipes
- Enfin il est possible de créer des custom pipe
 - Ou peut vouloir passer un xml en json par exemple
- Il faut importer la classe **PipeTransform**
 - Puis avec une décoration, nous allons pouvoir créer notre pipe

```
{{ birthday | date | uppercase }}
```

```
{{ birthday | date:'fullDate' | uppercase }}
```

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

```
@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);  
    }  
}
```



Angular : Directives et Pipes : Exercice

- Utiliser les différentes directives
 - 1 `ngfor` dans une liste d'ingrédients
 - 1 `ngIf` , si l'ingrédient est une fraise > afficher une balise
 - 1 `ngSwitch`, affiche un comportement sur 2 ingrédients
 - 1 `ngClass`, afficher en gras les ingrédients qui commencent pas « fr »
 - 1 `ngStyle`, dans le cas ou l'ingrédient est `ingredient.perime = true`

Si ce n'ai pas fait, créer un

*ingrédient : { **id** : number; **nom** :string, **quantite**: number, **prix**: number }*

Angular 2



- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
- ~~2. Le fonctionnement~~
- ~~3. les composants~~
- ~~4. les directives, pipes et methods~~
5. Forms
6. Injection de dépendance et services
7. Le service http
8. Le routing

Angular Forms



- Angular forms
 - C'est un formulaire html, rien de plus compliqué
 - Il est possible de lier les données dans le sens Model vers Input
 - Il est possible de lier les données dans les deux sens (un grande force d'angular)
 - Suivre les changements dans un form
 - Effectuer un contrôle sur les champs du form
 - Fournir une interface utilisateur enrichies avec des classes CSS prédéfinies
 - Créer des variables propres au form pour les utiliser dans angular

A screenshot of a web browser window. The address bar shows 'file:///Users/Pierre/De'. The page title is 'Les formulaires HTML'. The form contains two text input fields. The first is labeled 'Entrez un pseudo :'. The second is labeled 'Présentation :'. Below the inputs is a button labeled 'Envoyer'.

```
<form>
  <fieldset>

    <label for="range">Slider</label>
    <input type="range" name="range" />

    <label for="text">Champ de texte avec placeholder</label>
    <input type="text" name="text" id="text">

    <label for="url">Champ d'Url</label>
    <input type="url" id="url" name="url">

    <label for="email">Champ Email</label>
    <input type="email" id="email" name="email">

    <label for="numeric">Format numerique</label>
    <input type="number" name="numeric" id="numeric">

    <label for="date">Format date</label>
```



Angular : Forms

- Template driven form
 - Cela consiste à créer notre formulaire dans un template manuellement
 - On peut utiliser plusieurs éléments angular
 - L'interpolation pour afficher une information
 - Le binding avec ngModel, créé le lien entre le champ et le model
 - Les directives d'angular, ngclass, ngstyle, ngif, ngswitch, ngfor
 - Les templates references variables
 - Les events : onSubmit, click, etc.
 - Autres...

Angular : Forms



- Exemple de formulaire créé à la main
- Dans cet exemple on retrouve les directives
 - ngIf
 - [hidden]
- Liaison bidirectionnelle avec le model avec [(ngModel)]

```
2
3  <input type="text" id="name" class="form-control"
4        required minlength="4" maxlength="24"
5        name="name" [(ngModel)]="hero.name"
6        #name="ngModel" >
7
8  <div *ngIf="name.errors && (name.dirty || name.touched)"
9        class="alert alert-danger">
10    <div [hidden]="!name.errors.required">
11        Name is required
12    </div>
13    <div [hidden]="!name.errors.minlength">
14        Name must be at least 4 characters long.
15    </div>
16    <div [hidden]="!name.errors.maxlength">
17        Name cannot be more than 24 characters long.
18    </div>
19 </div>
```



Angular : Forms

- Faire une validation manuelle du formulaire
 - Afficher dans la page un message d'erreur précis
 - Gérer une partie de logique métier depuis le formulaire
 - Afficher un message comportemental
- Cela se fait avec les directives minlength, maxlength et pattern

```
<input type="text" pattern="[A-Za-z0-9]{0,5}">
```

```
<input type="text" maxlength="5">
```

Angular : Forms



- Validation personnalisée
 - Il est possible de coder ses propres validations
 - Il faut passer par un form builder

Ajouter les dependances FormBuilder et FormControl

Créer un formbuilder

Créer un formcontrol

Créer un form custom validation

app/login-form.component.ts

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



Angular : Forms

- Il est possible de détecter les erreurs dans l'initialisation du composant
- Ajouter la methode init
- Instancier un objet Observable
- Ajouter des methodes setTimeout
 - Indiquer un message suivant le setTimeout

```
5
6 export class MyApp {
7
8   private data: Observable<Array<number>>;
9   private values: Array<number> = [];
10  private anyErrors: error;
11
12  init() {
13
14    this.data = new Observable(observer => {
15      setTimeout(() => {
16        observer.next(10)
17      }, 1500);
18      setTimeout(() => {
19        observer.error('Hey something bad happened I guess');
20      }, 2000);
21      setTimeout(() => {
22        observer.next(50)
23      }, 2500);
24    });
25
26    let subscription = this.data.subscribe(
27      value => this.values.push(value),
28      error => this.anyErrors = error
29    );
30  }
31}
```

Angular 2



- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
- ~~2. Le fonctionnement~~
- ~~3. les composants~~
- ~~4. les directives, pipes et methods~~
- ~~5. Forms~~
6. Injection de dépendance et services
7. Le service http
8. Le routing



Angular : dépendance et services

- Pourquoi utiliser des services
 - Structurer son code
 - Décharger la class component
 - Rendre des fonctionnalités réutilisable

- Sans les services

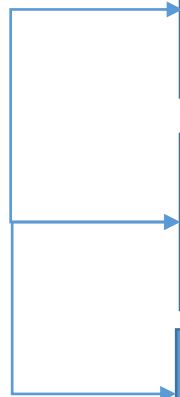
Class component : (récup donnée, traite données, calcul spécifique, autres, methodes event du template, autres..)

- Avec les services

Class component : (methodes d'événement, model, implémentation des services)

Class service : (traite les données, calcul spécifique, récup des données, implémente le service api)

Class service api : (récup des données en api)





Angular : dépendance et services

- Les services dans angular
 - Créer un service
 - Créer un classe dans un fichier ex:
 - Nom du fichier recette.service.ts
 - Contenu : export class RecetteService{}
 - Un service prend en la dépendance Injectable
 - Il prend aussi l'annotation @Injectable()

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

@Injectable()
export class HeroService {
}
```



Angular : dépendance et services

- Utilisation du service
 - Ajouter la dépendance du service avec Import dans le composant
 - Ne pas instancier le service !!!!
 - Nous allons ajouter le service dans le constructor de la classe de notre composant
 - De cette manière notre service sera instancié en même temps!
 - Enfin, on peut utiliser une methode du service dans une methode de notre classe de composant

```
import { HeroService } from './hero.service';
```

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

```
getHeroes(): void {  
    this.heroes = this.heroService.getHeroes();  
}
```

Angular : dépendance et services



- Les services
 - Créer un service contact peut permettre de manipuler les données
 - Dans mon contactService
 - ajoutContact
 - modifieContact
 - ...
 - Permet de séparer une partie de la logique de l'application
 - Ex : Créer un contact.service.ts
 - On importe la classe injectable
 - On ajoute la décoration @injectable
 - Cela permettra de gérer les dépendances

```
1 import { Injectable } from '@angular/core';  
2  
3 @Injectable()  
4 export class ContactService {  
5  
6 }
```

Angular : dépendance et services



- Les services
 - Ajoutons une methode getContacts
 - Le service se chargera de savoir d'où il prend les données
 - Créons un fichier de données mock-contacts.ts

```
3 @Injectable()
4 export class ContactService{
5     getContacts(): void {
6
7     }
8 }
```



Angular : dépendance et services

- ngOnInit pour exécuter une méthode au démarrage du composant
- Ajouter la dépendance OnInit
- Ajouter l'implémentation OnInit sur notre classe
- Ajouter la methode ngOnInit() {} pour exécuter du code au démarrage!
- Exemple : Aller chercher des données en ajax

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

export class AppComponent implements OnInit {
  ngOnInit(): void {
  }
}
```

Angular 2



- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
- ~~2. Le fonctionnement~~
- ~~3. les composants~~
- ~~4. les directives, pipes et methods~~
- ~~5. Forms~~
- ~~6. Injection de dépendance et services~~
7. Le service http
8. Le routing

Angular : HTTP



Json-server:

Package se trouvant ici :

<https://github.com/typicode/json-server>

S'installer comme cela : **npm install-g json-server**

Permet de simuler une api RES full basée sur un petit serveur http avec un persistance en json

- Exemple de fichier json :
- Ligne de commande d'ouverture de l'api :
 - **Json-server -watch database.json -port 8888**
- On peut tester l'api avec PostMan
 - **GET | PUT | POST | DELETE**

```
{  
  "posts": [  
    { "id": 1, "title": "json-server", "author": "typicode" }  
  ],  
  "comments": [  
    { "id": 1, "body": "some comment", "postId": 1 }  
  ]  
}
```



Angular : HTTP



Import du service HTTP

- Il faut importer la classe **HttpModule** présent dans **@angular/http** dans les import **@ngModule** de notre fichier *app.module.ts*
- Créons un service, si ce n'est déjà fait :
 - RecetteService dans lequel nous allons placer un `getRecettes()`
- Importer les classes header et http

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

```
@NgModule({  
  imports: [  
    BrowserModule,  
    FormsModule,  
    HttpModule,  
    AppRoutingModule  
  ],  
})
```

Angular : HTTP



HttpModule n'est **pas** un module Angular
!!!

- Le module est présent dans @angular/http
- Il est tout de meme référencé dans systemjs.config
- Dans le fichier app.module.ts – ajouter la ligne

```
1 import { NgModule } from '@angular/core';  
2 import { BrowserModule } from '@angular/platform-browser';  
3 import { HttpModule } from '@angular/http';  
4
```

```
. @NgModule({  
.   imports: [  
.     BrowserModule,  
.     FormsModule,  
.     HttpModule,  
.     AppRoutingModule  
.   ],  
  1
```

Angular : HTTP



Faire des appels http

- Créer le fichier recette.service.ts
- Ajouter la référence du service dans app.module.ts
- Ajouter dans le constructor du service la dépendance de la classe http
 - On utilise une dépendance, on a donc besoin d'importer la classe injectable
- Effectuer un appel avec notre objet http

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

import 'rxjs/add/operator/toPromise';

import { Recette } from '../recette';
export class RecetteService {

    constructor(private http: Http) { }

    recettesurlapi:string = "http://localhost/recettes";
    getAllRecetteApi(){
        return this.http.get(this.recettesurlapi)
            .toPromise()
            .then(response => response.json().data as Recette[])
            .catch(this.handleError);
    }
    //la methode retourne untableau de recette
}
```



Angular : HTTP

Les appels GET, post, put, delete

- La méthode Get prend seulement l'url de l'api en paramètre
 - On utilise la notion de promise . Then car l'appel est asynchrone
 - Vous pouvez créer un objet qui stocke toutes les urls de votre api.

```
return this.http.get(this.recettesurlapi)
    .toPromise()
    .then(response => response.json().data as Recette[])
    .catch(this.handleError);
```

- La méthode post, put pour les actions d'ajout ou de modification prend 2 paramètres :
 - Un identifiant que l'on ajoute sur l'url pour la modification
 - Un datarow qui prend un objet json (une recette dans notre cas)
- La méthode delete prend seulement l'identifiant de l'enregistrement en parametre



Angular : HTTP : Exercice

Utilisation du protocole http pour le POST, PUT, DELETE

- Créer un fake webservice avec json-server avec des recettes et des ingrédients
- Utiliser les service http pour récupérer les données du server à l'initialisation des composants

Angular 2



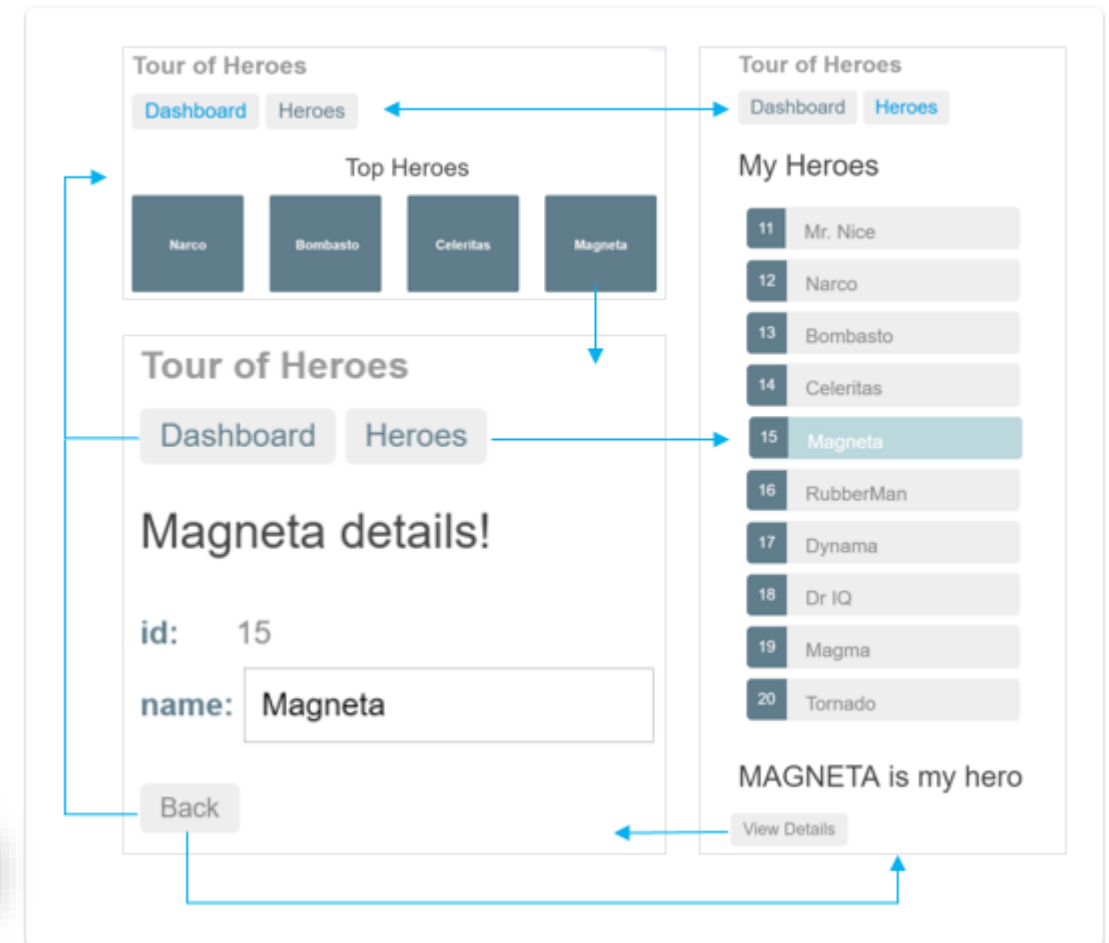
- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
- ~~2. Le fonctionnement~~
- ~~3. les composants~~
- ~~4. les directives, pipes et methods~~
- ~~5. Forms~~
- ~~6. Injection de dépendance et services~~
- ~~7. Le service http~~
8. Le routing

Angular : Le routage



- Mettre en place système de pages dans notre application
 - Basée sur l'url :
 - /
 - /heroes
 - /hero/id
 - /accueil
- Mettre en place un menu pour passer de page en page
 - Avec des liens dans les menus
- Mettre en place l'historique de navigation
- On ajoute l'import de RouterModule dans module.ts

```
import { RouterModule } from '@angular/router';
```



Angular : Le routage



Configuration des routes

- Indiquer la route racine dans le fichier html :
- Ajouter le module RouterModule dans app.module.ts
- L'objet RouterModule.forRoot prend un tableau de route en parametre
 - Un objet route prend
 - Un path (on peut ajouter un param avec :monparam)
 - Un component
- Le RouterModule est à déclarer dans les imports du @NgModule

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

```
@NgModule({  
  imports: [  
    BrowserModule,  
    FormsModule,  
    RouterModule.forRoot([  
      {  
        path: 'heroes',  
        component: HeroesComponent  
      }  
    ])  
  ]  
})
```

```
import { RouterModule } from '@angular/router';  
  
import { AppComponent } from './app.component';  
import { HeroDetailComponent } from './hero.detail.component';  
  
@NgModule({  
  imports: [  
    BrowserModule,  
    FormsModule,  
    RouterModule.forRoot([  
      { path: '/heroes', component: HeroesComponent },  
      { path: '/hero/:id', component: DetailHeroComponent },  
      { path: '/ajout', component: AjoutHeroComponent }  
    ])  
  ],  
  declarations: [ AppComponent, HeroDetailComponent ]  
})
```




Angular : Le routage

Les routes outlets

- router-outlet est une directive du router
- Le router va pouvoir injecter dans le outlet le component appelé par le router :
- Il se trouve dans le template de notre app.component.ts

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

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <a routerLink="/heroes">Heroes</a>
    <router-outlet></router-outlet>
  `
})
```

Angular : Le routage



Le router links

- Permet de créer des liens vers le routage créés dans notre module.ts
- Ex: Un bouton pour revenir à l'accueil
- La directive se place dans un bouton/ lien d'un template
- Attention : il faut bien avoir spécifier les routes avant
- Il y a le **RouterLink** ou le **RouterLinkActive**
 - Le router link active permet d'activer ou non une classe css

```
template: `
  <h1>{{title}}</h1>
  <a routerLink="/heroes">Heroes</a>
  <router-outlet></router-outlet>
`
```

```
template: `
  <h1>{{title}}</h1>
  <nav>
    <a routerLink="/dashboard">Dashboard</a>
    <a routerLink="/heroes">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`
```



Angular : Le routage : Exercice

Créer un routage permettant de passer sur les différents composants

- Liste des recettes
- Détail des recettes avec liste des ingrédients
- Ajout d'un ingrédient
- Ajout d'une recette
 - S'il vous reste du temps, travailler sur les modifications et suppressions de recette et ingrédients

Angular 2



- ~~1. TypeScript : une bonne nouvelle pour les dev oo~~
- ~~2. Le fonctionnement~~
- ~~3. les composants~~
- ~~4. les directives, pipes et methods~~
- ~~5. Forms~~
- ~~6. Injection de dépendance et services~~
- ~~7. Le service http~~
- ~~8. Le routing~~
9. Testing
10. Un dernier exercice



Angular 2 : Testing

- Angular testing permet
 - De faire des tests de non regression
 - D'identifier un code qui ne fonctionne plus
 - Trouver les erreurs d'architecture de l'app
- Les outils existants :
 - Jasmine : Peut faire du test dans l'interface utilisateur
 - Angular Testing utilities : passe votre app dans un environnement de test
 - Karma : Outil de test unitaire (test de classe, methode). Facilite les conception en intégration continue
 - Protractor : Plus poussé que Jasmin, permet de faire du test End 2 End, et permet de tester l'ui. (test de rapidité aussi)



Le dernier exercice

- Gestion de contacts
 - List des contacts par API
- Todo liste sur les contacts
 - Ajout/ liste / modification / suppression
- Utiliser le routage pour cliquer sur un contact et afficher la todo liste du contact
- Utiliser json-server pour simuler l'api contact
- Bonus – Trouver et intégrer un component Google Maps pour afficher la situation d'un contact
 - <https://github.com/SebastianM/angular2-google-maps>

Mots clés :
Component, forms,
http service,
directives,



Angular : Fin

Aller plus loin ?

- Vous connaissez la base d'Angular 2
- Visiter le site angular.io
- Explorer le code source sur github, pour comprendre le comportement du framework
- Participer aux events locaux – les NG-CONF
- Suivre les experts sur les réseaux sociaux
- Coder et toujours coder.
- Découvrez le développement mobile avec angular et ionic