
Angular

v2,v4 (avec npm et typescript)

Table des matières

I - Présentation de Angular.....	4
1. Présentation de Angular 2.....	4
II - Environnement de développement Angular.....	6
1. Environnement de développement pour Angular 2.....	6
III - Langage typescript.....	12
1. Bases syntaxiques du langage typescript (ts).....	12
IV - Essentiel sur templates , bindings , events.....	23
1. Templates bindings (property , event).....	23
V - Components (angular).....	30
1. Vue d'ensemble sur la structure du code Angular2.....	30
2. Les modules applicatifs.....	32

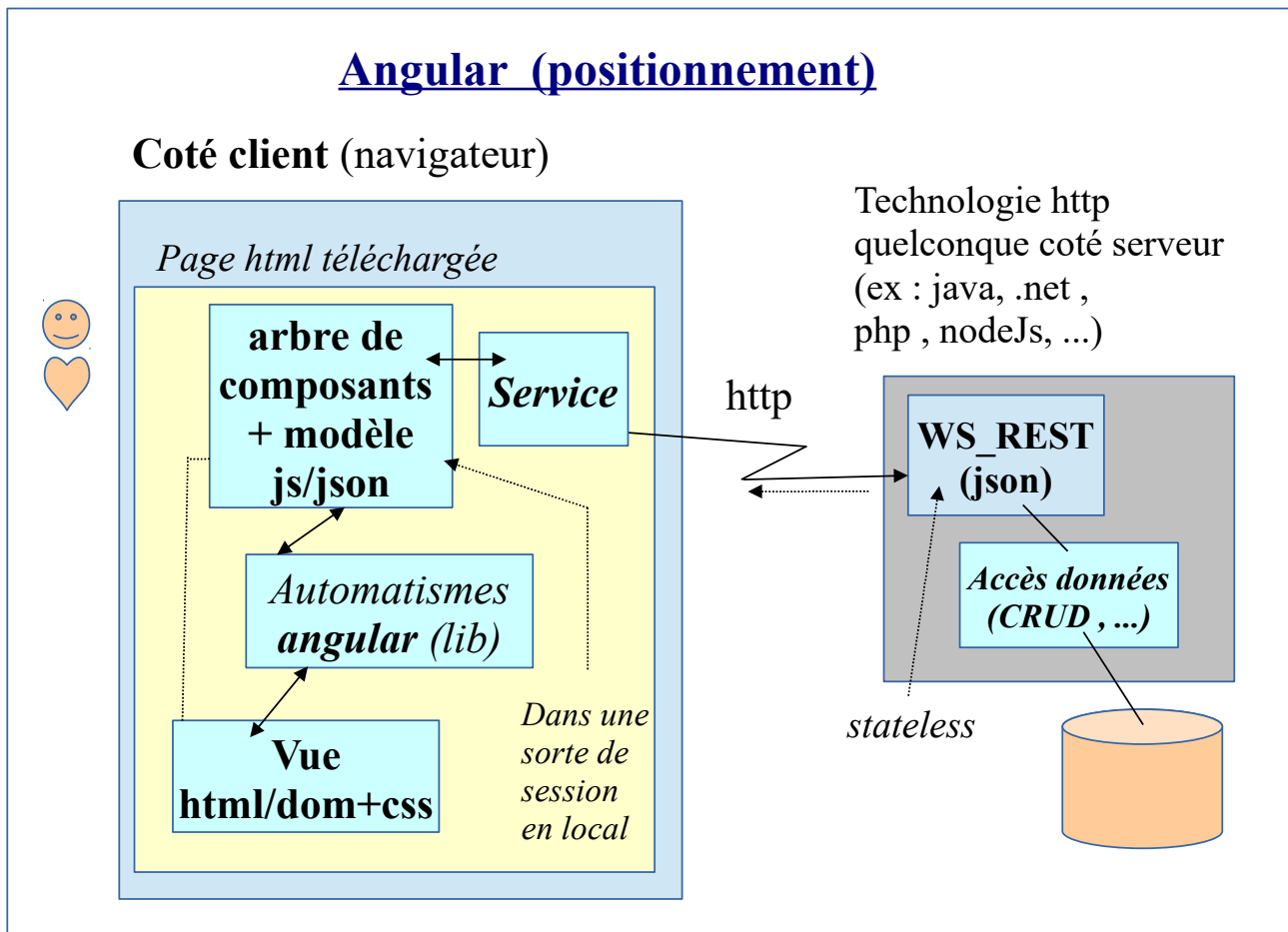
3. Les composants de l'application (@Component).....	32
4. Cycle de vie sur composants (et directives).....	38
VI - Directives , services et injections.....	41
1. Service.....	41
NB : Observable<...> ressemble un peu à Promise<...> et se consomme via ...	41
compteService.getComptesOfClientObservable(this.clientId).....	41
.subscribe(comptes =>this.comptes = comptes ,	41
error => console.log(error));.....	41
Observable (de rxjs) sera étudié de façon plus détaillée au sein d'un chapitre ultérieur (HTTP , ...)	41
2. Injection de dépendances.....	42
3. Les directives (angular2).....	44
VII - Switch et routing (navigations).....	50
1. Navigation via ngSwitch et router de angular2.....	50
VIII - Appels de W.S. REST (Observable, ...).....	56
1. Angular et dialogues HTTP/REST.....	56
2. Utilisation du service Http avec Observable (rxjs).....	59
3. Retransmission des éléments de sécurité.....	64
IX - Contrôles de formulaires , composants GUI.....	65
1. Contrôle des formulaires.....	65
X - Aspects divers de angular (pipes, ...).....	69
1. BehaviorSubject.....	69
2. Autres aspects divers.....	73
XI - Tests unitaires (et ...) avec angular.....	76
1. Tests autour de angular.....	76
XII - Packaging et déploiement d'appli. angular.....	93
1. Vue d'ensemble sur technologies "javascript".....	93
2. Précisions sur contexte développement "Angular".....	94
3. Packaging et déploiement avec ou sans bundle(s).....	96
4. Angular-CLI.....	98

XIII - Sécurité – application "Angular2".....	101
1. Sécurisation d'une application "angular2".....	101
XIV - Annexe – Détails Typescript et javascript.....	103
1. Détails du langage typescript (ts).....	103
XV - Annexe – Web Services REST (coté serveur).....	107
1. Généralités sur Web-Services REST.....	107
2. Limitations Ajax sans CORS.....	113
3. CORS (Cross Origin Resource Sharing).....	114
XVI - Annexe – Essentiel HTML , CSS.....	117
XVII - Annexe – Bibliographie, Liens WEB + TP.....	118
1. Bibliographie et liens vers sites "internet".....	118
2. TP.....	118

I - Présentation de Angular

1. Présentation de Angular 2

1.1. Positionnement du framework "Angular"



Angular est un **framework web** de **Google** qui **s'exécute entièrement du côté navigateur** et dont la programmation est basée sur le langage **typescript** (ou ...) transpilé/compilé en **javascript** (ES5).

En s'étant inspiré du design pattern "**MVVM**" (**Model-View-ViewModel**) proche de **MVC**, le framework Angular gère automatiquement une mise à jour de la vue HTML qui s'affiche dans le navigateur en effectuant (quasi-automatiquement) des synchronisations par rapports aux valeurs d'un modèle orienté objet (compatible JSON) qui est géré en mémoire par une **hiérarchie de composants** (codés en typescript "orienté objet" et traduits en "javascript").

Les récupérations de données s'effectue via des services (à programmer) et dont la responsabilité est souvent d'appeler des web services REST (transfert de données JSON via HTTP).

Les principaux intérêts de la technologie angular sont les suivants :

- une grande partie des traitements web s'effectue coté client (dans le navigateur) et le serveur se voit alors déchargé d'une lourde tâche (refabriquer des pages, gérer les sessions)

- utilisateurs, ...) ---> bien pour tenir la charge .
- meilleurs performances/réactivités du coté affichage/présentation web (navigateur) : c'est directement l'arbre DOM qui est réactualisé/rendu à partir des modifications apportées sur le modèle typescript/javascript (plus de html à transférer/ré-analyser).
- séparation claire entre la partie "présentation" (js) et la partie "services métiers" (java ou ".net" ou ".php" ou "nodejs" ou ...) . Google présente d'ailleurs parfois angularJs ou Angular2 comme un framework MVW (Model-View-**Whatever**) .

1.2. Particularités du framework "Angular" (V2)

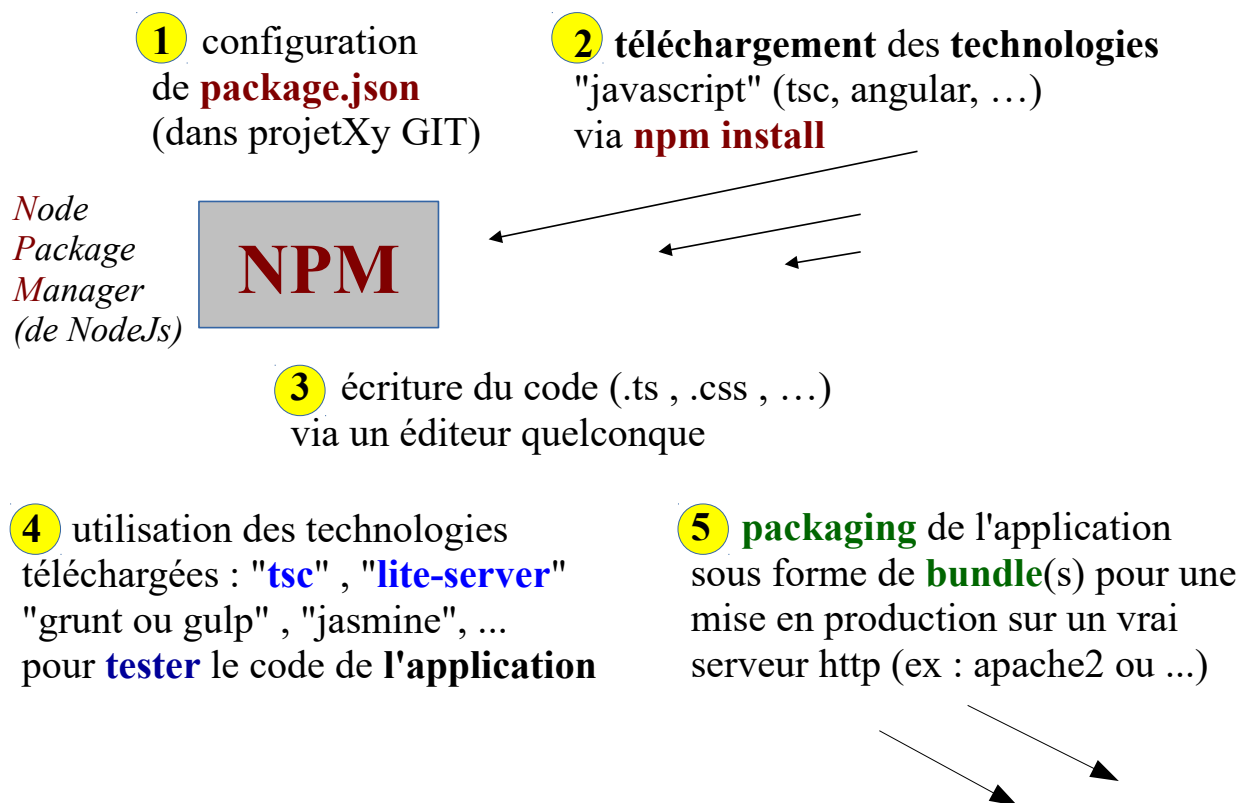
- bien structuré (orienté objet / syntaxe rigoureuse grâce à typescript)
- dès le départ très complet
- ...

II - Environnement de développement Angular

1. Environnement de développement pour Angular 2

1.1. Présentation de l'environnement de développement

Environnement de développement Angular



1.2. Configuration "npm" pour Angular

Un développement sérieux basé sur Angular s'effectue à partir de "**npm**" (de l'environnement NodeJs) .

Si par encore fait , installer "nodeJs" pour bénéficier de la sous partie intégrée "npm" .

Dans le répertoire du projet de l'application (ex : *angular2_quickstart*) ,
écrire (par copier/coller et éventuelles adaptation de versions)
le contenu du fichier **package.json** :

```
{
  "name": "angular2-quickstart",
  "version": "1.0.0",
```

```

"scripts": {
  "start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",
  "lite": "lite-server",
  "tsc": "tsc",
  "tsc:w": "tsc -w"
},
"licenses": [
  {
    "type": "MIT",
    "url": "https://github.com/angular/angular.io/blob/master/LICENSE"
  }
],
"dependencies": {
  "@angular/common": "~2.1.1",
  "@angular/compiler": "~2.1.1",
  "@angular/core": "~2.1.1",
  "@angular/forms": "~2.1.1",
  "@angular/http": "~2.1.1",
  "@angular/platform-browser": "~2.1.1",
  "@angular/platform-browser-dynamic": "~2.1.1",
  "@angular/router": "~3.1.1",
  "@angular/upgrade": "~2.1.1",
  "angular-in-memory-web-api": "~0.1.13",
  "core-js": "^2.4.1",
  "reflect-metadata": "^0.1.8",
  "rxjs": "5.0.0-beta.12",
  "systemjs": "0.19.39",
  "zone.js": "^0.6.25"
},
"devDependencies": {
  "@types/core-js": "^0.9.34",
  "@types/node": "^6.0.45",
  "concurrently": "^3.0.0",
  "lite-server": "^2.2.2",
  "typescript": "^2.0.3"
}
}

```

Lancer ensuite la commande **"npm install"** de façon à *télécharger les technologies nécessaires* dans le sous répertoire ordinaire *"node_modules"* géré par *npm* .

NB :

Les parties "...dependencies" servent à paramétrer les librairies à télécharger.

La partie "scripts" est paramétrée de telle sorte qu'un futur déclenchement de "npm start" lancera en parallèle (de manière "concurrent") le pré-compilateur "tsc" de typescript et le mini serveur web "lite-server" de façon à tester l'application.

Dans le répertoire du projet (à coté de *package.json*) , écrire (par copier/coller) le contenu du fichier **tsconfig.json** :

```
{
```

```
"compilerOptions": {
  "target": "es5",
  "module": "commonjs",
  "moduleResolution": "node",
  "sourceMap": true,
  "emitDecoratorMetadata": true,
  "experimentalDecorators": true,
  "removeComments": false,
  "noImplicitAny": true,
  "suppressImplicitAnyIndexErrors": true,
  "typeRoots": [
    "./node_modules/@types/"
  ]
},
"compileOnSave": true,
"exclude": [
  "node_modules"
]
}
```

Ceci servira à paramétrer le comportement du pré-processeur (ou pré-compilateur) "tsc" permettant de transformer des fichiers ".ts" en fichiers ".js" .

Angular 2 s'appuie en interne sur la technologie "SystemJs" pour charger des modules javascript en mémoire. Le fichier de configuration **systemjs.config.js** est ainsi nécessaire pour le démarrage d'une application "angular2" :

```
/**
 * System configuration for Angular application ( systemjs.config.js )
 * Adjust as necessary for your application needs.
 */
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',
      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic':
        'npm:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',
      '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
      '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
      '@angular/upgrade': 'npm:@angular/upgrade/bundles/upgrade.umd.js',
    }
  });
})
```



```
// other libraries
'rxjs':      'npm:rxjs',
'angular-in-memory-web-api': 'npm:angular-in-memory-web-api/bundles/in-memory-web-api.umd.js'
},
// packages tells the System loader how to load when no filename and/or no extension
packages: {
  app: {
    main: './main.js',
    defaultExtension: 'js'
  },
  rxjs: {
    defaultExtension: 'js'
  }
}
});
})(this);
```

Créer ensuite l'arborescence de fichiers et répertoire(s) suivante :

```
index.html
app
  app.component.ts
  app.module.ts
  main.ts
```

Contenu des fichiers :

app/app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<h1>My First {{message}} .. </h1>
    <table border="1">
      <tr> <th>i</th> <th>i*i</th> </tr>
      <tr *ngFor="let i of values" >
        <td>{{i}}</td> <td>{{i*i}}</td>
      </tr>
    </table>`
})

export class AppComponent {
  message: string ;
  values: number[] = [1,2,3,4,5,6,7,8,9];
  constructor(){
    this.message = "Angular 2 App";
  }
}
```

app/app.module.ts

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

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

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

app/main.ts

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

index.html

```
<html>
<head> <title>Angular 2 QuickStart</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="./css/styles.css">
  <!-- 1. Load libraries -->
  <!-- Polyfill(s) for older browsers -->
  <script src="node_modules/core-js/client/shim.min.js"></script>

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

  <!-- 2. Configure SystemJS -->
  <script src="systemjs.config.js"></script>
  <script>
    System.import('app').catch(function(err){ console.error(err); });
  </script>
</head>

<!-- 3. Display the application -->
<body>
  <my-app>Loading...</my-app>
</body>
</html>
```

Cet exemple montre qu'une application "angular 2" est structurée à partir de composants dont le code se situe dans des modules reprenant la même logique structurelle que nodeJs (import /export avec chemins relatifs).



III - Langage typescript

1. Bases syntaxiques du langage typescript (ts)

1.1. TypeScript / ts en tant qu'évolution de ES6

Le langage "typescript" est une évolution de ES6 (ECMAScript 6) avec un typage fort et des annotations.

Ce langage (à l'origine créé par Microsoft) s'utilise concrètement en écrivant des fichiers ".ts" qui sont transformés en fichiers ".js" via un pré-processeur "tsc" (typescript compiler) .

Cette phase de transcription (".ts → .js") s'effectue généralement durant la phase de développement.

NB : tant que l'on ajoute pas de spécificités "typescript" , le fichier ".js" généré est identique au fichier ".ts" .

Si par contre on ajoute des précisions sur les types de données au sein du fichier ".ts" alors :

- le fichier ".js" est bien généré (par simplification ou développement) si aucune erreur bloquante est détectée.
- des messages d'erreurs sont émis par "tsc" si des valeurs sont incompatibles avec les types des paramètres des fonctions appelées ou des affectations de variables programmées.

1.2. utilisation concrète de tcs

Le déclenchement de la **transcription/compilation de "ts" vers "js"** peut s'effectuer de différentes manières :

- via une ligne de commande (éventuellement placée dans un script) :

compile_ts_to_js.bat

```
REM npm run tsc hello_world.ts
REM      avec option :w signifiant "watch mode" (npm run tsc:w) , recompilation dès qu'une
REM      modification est détectée sur fichier .ts enregistré
npm run tsc:w hello_world.ts
```

- via le "task runner" Grunt.ts :
-

1.3. précision des types de données

boolean	<code>var isDone: boolean = false;</code>
number	<code>var height: number = 6;</code> <code>var size : number = 1.83 ;</code>
string	<code>var name: string = "bob";</code> <code>name = 'smith';</code>
array	<code>var list1 : number[] = [1, 2, 3];</code> <code>var list2 : Array<number> = [1, 2, 3];</code>
enum	<code>enum Color {Red, Green, Blue}; // start at 0 by default</code> <code>// enum Color {Red = 1, Green, Blue};</code> <code>var c: Color = Color.Green; //display as "1" by default</code> <code>var colorName: string = Color[1]; // "Green" if "Red" is at [0]</code>
any	<code>var notSure: any = 4;</code> <code>notSure = "maybe a string instead";</code> <code>notSure = false;</code>
void	<code>function warnUser(): void {</code> <code> alert("This is my warning message");</code> <code>}</code>

hello_world.ts

```
function greeterString(person : string) {
    return "Hello, " + person;
}
var userName = "Power User";
//i=0; //manque var (erreur détectée par tsc)

var msg = "";
//msg = greeterString(123456); //123456 incompatible avec type string (erreur détectée par tsc)
msg = greeterString(userName);
console.log(msg);
```

values: number[] = [1,2,3,4,5,6,7,8,9];

1.4. Approche orientée objet

Principaux mots clefs : **interface** , **class** , **constructor** , **static**, ...

person.ts

```
interface Person {
  firstname: string;
  lastname: string;
}

function greeterPerson(person : Person) {
  return "Hello, " + person.firstname + " " + person.lastname;
}

//var user = {name: "James Bond", comment: "top secret"};
//incompatible avec l'interface Person (erreur détectée par tsc)

var user = {firstname: "James", lastname: "Bond", country: "UK"};
//ok : compatible avec interface Person

msg = greeterPerson(user);
console.log(msg);

class Student {
  fullname : string;
  constructor(public firstname, public lastname, public schoolClass) {
    this.fullname = firstname + " " + lastname + "[" + schoolClass + "]";
  }
}

var s1 = new Student("cancre", "Ducobu", "Terminale"); //compatible avec interface Person
msg = greeterPerson(s1);
console.log(msg);
```

Remarque importante : via le mot clef "**public**" ou "**private**" (au niveau des paramètres du constructeur) , certains paramètres passés au niveau du constructeur sont automatiquement transformés en attributs/propriétés de la classe (ici "*Student*") qui devient donc compatible avec l'interface "*Person*".

Au sein du langage "typescript", une **interface** correspond à la notion de "**structural subtyping**".

Le véritable objet qui sera compatible avec le type de l'interface pourra avoir une structure plus grande.

Interface simple/classique :

```
interface LabelledValue {  
  label: string;  
}
```

```
function printLabel(labelledObj: LabelledValue) {  
  console.log(labelledObj.label);  
}  
  
var myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);
```

Interface avec propriété(s) facultative(s) (suffixée(s) par?)

```
interface LabelledValue {  
  label : string;  
  size? : number ;  
}
```

```
function printLabel(labelledObj: LabelledValue) {  
  console.log(labelledObj.label);  
  if( labelledObj.size ) {  
    console.log(labelledObj.size);  
  }  
}  
  
var myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);  
  
var myObj2 = { label: "Unknown Size Object"};  
printLabel(myObj2);
```

Interface pour type précis de fonctions :

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
```

→ deux paramètres d'entrée de type "string" et valeur de retour de type "boolean"

```
var mySearch: SearchFunc;

mySearch = function(src: string, sub: string) {
  var result = src.search(sub);
  if (result == -1) {
    return false;
  }
  else {
    return true;
  }
} //ok
```

Interface pour type précis de tableaux :

```
interface StringArray {
  [index: number]: string;
}
```

```
var myArray: StringArray;
myArray = ["Bob", "Fred"];
```

Interface pour type précis d'objets :

```
interface ClockInterface {
  currentTime: Date;
  setTime(d: Date);
}
```

```
class Clock implements ClockInterface {
  currentTime: Date;
  setTime(d: Date) {
    this.currentTime = d;
  }
  constructor(h: number, m: number) { }
}
```


Héritage (simple ou multiple) entre interfaces :

```
interface Shape {
    color: string;
}
```

```
interface PenStroke {
    penWidth: number;
}
```

```
interface Square extends Shape, PenStroke {
    sideLength: number;
}
```

```
var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

```
var square2: Square = { "color": "blue" , "sideLength": 10, "penWidth": 5.0 } ;
```

Classes avec héritage et valeurs par défaut pour arguments :

```
class Animal {
    name:string;
    constructor(theName: string ="default animal name") { this.name= theName;}
    move(meters: number = 0) {
        console.log(this.name + " moved " + meters + "m.");
    }
}
```

```
class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 5) {
        console.log("Slithering...");
        super.move(meters);
    }
}
```

```
class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 45) {
        console.log("Galloping...");
        super.move(meters);
    }
}
```

```
var a = new Animal(); //var a = new Animal("animal");

var sam = new Snake("Sammy the Python"); //var sam = new Snake();

var tom: Animal = new Horse("Tommy the Palomino");

a.move() ; // default animal name moved 0m.
sam.move(); // Slithering... Sammy the Python moved 5m.

tom.move(34); //avec polymorphisme (for Horse)
// Galloping... Tommy the Palomino moved 34m.
```

Propriété "private"

```

class Animal {
    private _size : number;
    name:string;
    constructor(theName: string = "default animal name") {
        this.name = theName;
        this._size = 100; //by default
    }
    move(meters: number = 0) {
        console.log(this.name + " moved " + meters + "m." + " size=" + this._size);
    }
}

```

```
var a1 = new Animal("favorite animal");
```

a1._size=120; //erreur détectée ' _size' est privée et seulement accessible depuis classe 'Animal'.

```
a1.move();
```

Remarques importantes :

- **public par défaut .**
- En cas d'erreur détectée sur "private / not accessible" , le fichier ".js" est (par défaut) tout de même généré par "tsc" et l'accès à ".size" est tout de même autorisé / effectué au runtime.
⇒ private génère donc des messages d'erreurs qu'il faut consulter (pas ignorer) !!!

Accesseurs automatiques **get xxx() / **set xxx()** avec option -t ES5 de tsc**

```

class Animal {
    private _size : number;
    public get size() : number { return this._size;
    }
    public set size(newSize : number){
        if(newSize >=0) this._size = newSize;
        else console.log("negative size is invalid");
    }
    ...} //NB : le mot clef public est facultatif devant "get" et "set" (public par défaut)

```

```
var a1 = new Animal("favorite animal");
a1.size = -5; // calling set size() → negative size is invalid (at runtime) , _size still at 100
a1.size = 120; // calling set size()
a1.move();
console.log("size=" + a1.size) ; // calling get size() → affiche size=120
```

Static

De la même façon que dans beaucoup d'autres langages orientés objets (c++, java, ...) , le mot clef **static** permet de déclarer des variables/attributs de classes (plutôt que des variables/attributs d'instances).

La valeur d'un attribut "static" est partagée par toutes les instances d'une même classe et l'accès s'effectue avec le préfixe "NomDeClasse." plutôt que "this." .

Exemple:

```
class CompteEpargne {
    static taux: number= 1.5;
    constructor(public numero: number , public solde :number = 0){
    }
    calculerInteret(){
        return this.solde * CompteEpargne.taux / 100;
    }
}

var compteEpargne = new CompteEpargne(1,200.0);
console.log("interet="+compteEpargne.calculerInteret());
```

1.5. Modules

Le langage typescript gère deux sortes de modules "internes/logiques" et "externes" :

internes/logiques	Via mot clef module <i>ModuleName</i> { ... } englobant plusieurs "export" (<i>sémantique de "namespace"</i> et utilisation via préfixe "ModuleName.")	Dans un seul fichier ou réparti dans plusieurs fichiers (à regrouper) , peu importe.
externes	Via (au moins un) mot clef export au <i>premier niveau d'un fichier</i> et utilisation via mot clef import (associé à requires('./moduleName')) .	Toujours un fichier par module externe (nom du module = nom du fichier) Selon contexte (nodeJs ou ...)

1.6. Programmation fonctionnelle (lambda , ...)

Rappels (2 syntaxes "javascript" ordinaires) valables en "typescript" :

```
//Named function:
function add(x, y) {
  return x+y;
}

//Anonymous function:
var myAdd = function(x, y) { return x+y; };
```

Versions avec paramètres et valeur de retour typés (typescript) :

```
function add(x: number, y: number): number {
  return x+y;
}

var myAdd = function(x: number, y: number): number { return x+y; };
```

Type complet de fonctions :

```
var myAdd : (a:number, b:number) => number =
  function(x: number, y: number): number { return x+y; };
```

NB : les noms des paramètres (ici "a" et "b") ne sont pas significatifs dans la partie "type de fonction". Ils ne sont renseignés que pour la lisibilité .

Le type de retour de la fonction est préfixé par **=>** . Si la fonction de retourne rien alors **=> void** .

Inférences/déduction de types (pour paramètres et valeur de retour du code effectif):

```
var myAdd: (a:number, b:number)=>number =
  function(x, y) { return x+y; };
```

Paramètre de fonction optionnel (suffixé par ?)

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

var result1 = buildName("Bob"); //ok
var result2 = buildName("Bob", "Adams", "Sr."); //error, too many parameters
var result3 = buildName("Bob", "Adams"); //ok
```

Valeur par défaut pour paramètre de fonction

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

var result1 = buildName("Bob"); //ok : Bob Smith
var result2 = buildName("Bob", "Adams", "Sr."); //error, too many parameters
var result3 = buildName("Bob", "Adams"); //ok
```

Derniers paramètres facultatifs (... [])

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

var employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Lambda expressions

Une "**lambda expression**" est syntaxiquement introduite via **() => {}**

Il s'agit d'une syntaxe épurée/simplifiée d'une fonction anonyme où les parenthèses englobent d'éventuels paramètres et les accolades englobent le code.

Subtilité de "typescript" :

La valeur du mot clef "this" est habituellement évaluée lors de l'invocation d'une fonction .
Dans le cas d'une "lambda expression" , le mot clef this est évalué dès la création de la fonction.

Exemples de "lambda expressions" :

```
var myFct : ( tabNum : number[]) => number ;

myFct = (tab) => { var taille = tab.length; return taille; }
//ou plus simplement:
myFct = (tab) => { return tab.length; }

//ou encore plus simplement:
myFct = (tab) => tab.length;

//ou encore plus simplement:
myFct = tab => tab.length;
```

```
var numRes = myFct([12,58,69]);
console.log("numRes=" + numRes);
```

```
var myFct2 : ( x : number , y: number ) => number ;
myFct2 = (x,y) => { return (x+y) / 2; }
//ou plus simplement:
myFct2 = (x,y) => (x+y) / 2;
```

1.7. Generics

Fonctions génériques :

```
function identity<T>(arg: T) : T {
    return arg;
}
```

T sera remplacé par un type de données (ex : string , number , ...) selon les valeurs passées en paramètres lors de l'invocation (Analyse et transcription "ts" → "js") .

```
var output = identity<string>("myString"); // type of output will be 'string'
```

```
var output = identity("myString"); // type of output will be 'string' (par inférence/déduction)
var output2 = identity(58.6); // type of output will be 'number' (par inférence/déduction)
```

Classes génériques :

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

var myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };

var stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };
```

```
interface Lengthwise {
    length: number;
}

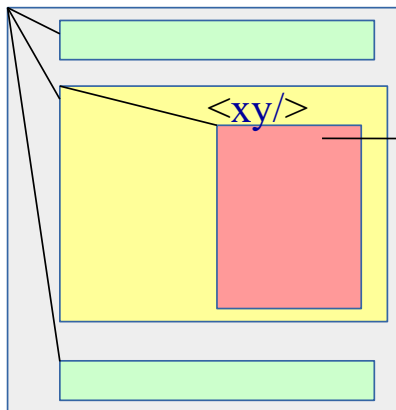
function loggingIdentity <T extends Lengthwise>(arg: T): T {
    console.log(arg.length); // we know it has a .length property, so no error
    return arg;
}
```

IV - Essentiel sur templates , bindings , events

1. Templates bindings (property , event)

1.1. Binding Angular

Page "angular" constituée d'une hiérarchie de composants



Binding Angular

Anatomie de chaque composant :

```
@Component({
  selector: 'xy',
  templateUrl: 'app/xy.component.html',
})
export class XyComponent {
  data : DataTypeZz ;
  ...
  OnNewXy = function(evt) { ...}
}
```

xy.component.html

```
<button (click)=
  "onNewXy($event)">
<p>{{data.label}}</p>
<input [(ngModel)]
  ="data.value">
```

(Modèle orienté objet)

```
data.id
.label
.value
```

Le principal intérêt du framework angular réside dans les liaisons automatiques établies entre les parties d'un modèle de vue HTML (template) et les données d'un modèle orienté objet en mémoire dans le composant.

Ce "mapping" ou "binding" peut soit être unidirectionnel ({{...}} ou [...]="...") ou bien bi-directionnel via [(...)]="....." .

Le déclenchement de méthodes événementielles via la syntaxe (evtName) = "....(\$event)" constitue également un paramétrage du mapping angular.

1.2. Syntaxe des liaisons entre vue/template et modèle objet

Syntaxes (template HTML)	Effets/comportements
<code><p>Hello {{ponyName}}</p></code>	Affiche Hello <i>poney1</i> si <i>ponyName</i> vaut <i>poney1</i>
<code><p>Employer: {{employer?.companyName}}</p></code>	Pas exception (affichage ignoré) si l'objet (facultatif/optionnel) est un "undefined"
<code><p>Card No.: {{cardNumber myCreditCardNumberFormatter}}</p></code>	Pipe(s) pour préciser un ou plusieurs(s) traitement(s) avant affichage
<code><input [value]="firstName"></code>	Affecte la valeur de l'expression <i>firstName</i> à la propriété <i>value</i> (one-way)
<code><div title="Hello {{ponyName}}"></code>	équivalent à: <code><div [title]=''Hello' + ponyName"></code>
<code><div [style.width.px]="mySize"></code>	Affecte la valeur de l'expression <i>mySize</i> à une partie de style css (ici <i>width.px</i>)
<code><button (click)="readRainbow(\$event)"></code>	Appelle la méthode <i>readRainbow()</i> en passant l'objet <i>\$event</i> en paramètre lorsque l'événement <i>click</i> est déclenché sur le composant courant (ou un de ses sous composants)
<code><input [(ngModel)]="userName"></code>	Liaison dans les 2 sens (lecture/écriture). <u>NB</u> : <i>ngModel</i> est ici une directive d'attribut prédéfinie dans le module <i>FormsModule</i> .
<code><my-cmp [(title)]="name"></code>	"two-way data binding" sur (sous-)composant. équivalent à: <code><my-cmp [title]="name" (titleChange)="name=\$event"></code>

1.3. Exemples d'interpolations {{ }}

`<p>My current hero is {{currentHero.firstName}}</p>`

`<p>The sum of 1 + 1 is {{a + b}}</p>`

`<p>The sum of 1 + 1 is not {{a + b + computeXy()}}</p>`

`<!-- computeXy() appelé sur composant courant associé au template →`

1.4. Principales directives prédéfinies (angular2/common)

<code><section *ngIf="showSection"></code>	Rendu conditionnel (si expression à true) . Très pratique pour éviter exception <code>{{obj.prop}}</code> lorsque obj est encore à "undefined" (pas encore chargé)
<code><li *ngFor="let item of list"></code>	Elément répété en boucle (forEach)
<code><div [ngClass]="{active: isActive, disabled: isDisabled}"></code>	Associe (ou pas) certaines classes de styles CSS selon les expressions booléennes .

1.5. Exemples

Cercle qui va bien

`(x,y)=(15,9) , r = 60`

<code>x: 15</code>	with <code>[value]="c1.x"</code> attribute synchronisation
<code>x: 15</code>	bi-directional via directive <code>[(ngModel)]="c1.x"</code>
<code>y: 9</code>	with <code>(input)="onNewY(\$event)"</code> event fuction
<code>y: 9</code>	with <code>(input)="c1.y = \$event.target.value;"</code> event fuction
<code>r: 60</code>	bi-directional via directive <code>[(ngModel)]="c1.r"</code>

app.component.ts

```
import {Component} from 'angular2/core';

interface Circle {
  x: number;
  y: number;
  r: number;
}

@Component({
  selector: 'my-app',
  template: `
    <h2>{{title}}</h2>
    <div> (x,y)=<b>({{c1.x}},{{c1.y}})</b> , r=<b> {{c1.r}} </b> </div>
    <div>
      <label>x: </label> <input [value]="c1.x" placeholder="x"/> <br/>
      <label>x: </label> <input [(ngModel)]="c1.x" placeholder="x"/> <br/>
      <label>y: </label> <input (input)="onNewY($event)" placeholder="y"/> <br/>
      <label>y: </label> <input (input)="c1.y = $event.target.value;" placeholder="y"/> <br/>
      <label>r: </label> <input [(ngModel)]="c1.r" placeholder="r"/> <br/>
    </div>
  `
})
```

```

})
export class AppComponent {
  public title = 'Cercle qui va bien';
  public c1: Circle = {
    x: 10,
    y: 20,
    r: 50
  };
  onNewY = function (evt : any){
    this.c1.y = evt.target.value;
  }
}

```

Remarques : un template sur plusieurs lignes peut être encadré par des quotes inverses `...` .

version fortement typée :

```

onNewY=function (evt : KeyboardEvent){
  this.c1.y = (<HTMLInputElement> evt.target).value;
}

```

NB : l'utilisation de **[(ngModel)]** nécessite l'importation de **FormsModule** dans **app.module.ts** :

```

import { NgModule }    from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

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

```

Autre exemple :

```

...
<ul class="heroes">
  <li *ngFor="let hero of heroes" [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>

<div *ngIf="selectedHero">
  <h2>{{selectedHero.name}} details!</h2>
  <div><label>id: </label>{{selectedHero.id}}</div>
  ...
</div>
....
export class AppComponent {
  public title :string = 'Tour of Heroes';
  public heroes : Hero[] = HEROES;
  public selectedHero: Hero;

  onSelect(hero: Hero) { this.selectedHero = hero; }
}

```

```

@Component({
  selector: 'key-up3',
  template: ` <input #box (keyup.enter)="values=box.value">
    <p>{{values}}</p> `
})
export class KeyUpComponent_v3 {
  values="";
}

```

NB : dans l'exemple précédent '**enter**' est un **filtrage** sur l'événement "**keyup**" (seul un relâchement de la touche "enter" est traité).

1.6. Précision (vocabulaire) "attribut HTML , propriété DOM"

`<input type="text" value="Bob">` est une syntaxe HTML au sein de laquelle l'attribut **value** correspond à la **valeur initiale de la zone de saisie**.

Lorsque cette balise HTML sera interprétée , elle sera transformée en sous arbre DOM puis affichée/rendue par le navigateur internet.

Lorsque l'utilisateur saisira une nouvelle valeur (ex : "toto") :

- la valeur de l'attribut HTML *value* sera inchangée (toujours même valeur initiale/par défaut) .
- La valeur de la propriété "value" attachée à l'élément de l'arbre DOM aura la nouvelle valeur "toto" .

Autrement dit, un attribut HTML ne change pas de valeur, tandis qu'une propriété de l'arbre DOM peut changer de valeur et pourra être mise en correspondance avec une partie d'un composant "angular2" .

NB : Au sein de l'exemple ci-dessous , la propriété "**disabled**" de l'élément de l'arbre DOM est évaluée à partir de la propriété "*isUnchanged*" du composant courant .

`<button [disabled]="isUnchanged">Save</button>`

et la propriété "**disabled**" de l'élément DOM a (par coïncidence non systématique) le même nom que l'attribut "**disabled**" de la balise HTML button .

Exception qui confirme la règle :

Dans le cas, très rare , où une propriété d'un composant "angular2" doit être associé à la valeur d'un attribut d'une balise HTML , la syntaxe prévue est **[attr.nomAttributHtml]="expression"** .

Exemple: `<td [attr.colspan]="1 + 1">One-Two</td>`

1.7. Style binding

```
<button [style.color] = "isSpecial ? 'red' : 'green'">Red</button>
<button [style.backgroundColor]="canSave ? 'cyan' : 'grey'" >Save</button>
```

Au sein des exemples ci-dessus, un seul style css n'était dynamiquement contrôlé à la fois.

De façon à contrôler dynamiquement d'un seul coup les valeurs de plusieurs styles css on pourra préférer l'utilisation de la directive **ngStyle** :

```
setStyles() {
  return {
    // CSS property names
    'font-style': this.canSave    ? 'italic' : 'normal', // italic
    'font-weight': !this.isUnchanged ? 'bold' : 'normal', // normal
    'font-size': this.isSpecial   ? 'x-large' : 'smaller', // larger
  }
}

<div [ngStyle]="setStyles()">
  This div is italic, normal weight, and x-large
</div>
```

1.8. CSS Class binding

La classe CSS "special" (.special { ... }) est dans l'exemple ci dessous associée à l'élément <div> de l'arbre DOM si et seulement si l'expression "isSpecial" est à true au sein du composant .

```
<div [class.special]="isSpecial">The class binding is special</div>
```

Cette syntaxe est appropriée et conseillée pour contrôler l'application conditionnée d'une seule classe de style CSS.

De façon à contrôler dynamiquement l'application de plusieurs classe CSS , on préférera la directive **ngClass** spécialement prévue à cet effet :

```
setClasses() {
  return {
    saveable: this.canSave,    // true
    modified: !this.isUnchanged, // false
    special: this.isSpecial,   // true
  }
}
```

pour un paramétrage selon la syntaxe suivante :

```
<div [ngClass]="setClasses()">This div is saveable and special</div>
```

V - Components (angular)

1. Vue d'ensemble sur la structure du code Angular2

1.1. Deux niveaux de modularité : classes et modules

Une **classe** (composant visuel , directive , service, ...) est un composant de petite taille (généralement de niveau **fichier**).

Un **module** (correspondant généralement à un répertoire ou sous répertoire) est un gros composant d'une application Angular2. Chaque module comporte un fichier principal *xyz.module.ts* comportant la décoration **@NgModule**.

Une petite application peut comporter un seul module fonctionnel (ex : "app") .

Une grosse application peut comporter plusieurs modules complémentaires.

1.2. Programmation "orientée objet" et "modularité par classe"

Selon une logique proche de celle de nodeJs , une application "Angular2" peut s'appuyer sur les mots clefs "**export**" et "**import**" (de TypeScript et de ES2015) de façon à être construite sur une base **modulaire** .

Chaque composant élémentaire est un fichier à part. (dans le cas d'un composant visuel , il pourra y avoir des fichiers annexes ".html" , ".css") .

Il **exporte quelque-chose** (classe , interface , données , ...).

Un composant angular2 doit importer un ou plusieurs autre(s) module(s) ou classe(s) / composant(s) s'il souhaite **avoir accès aux éléments exportés** et les utiliser. Le référencement d'un autre module s'effectue généralement via un chemin relatif commençant par **"/"** .

Exemple :

app/app.component.ts

```
...
export class AppComponent { ... }
```

app/app.module.ts

```
import {AppComponent} from './app.component'
...
@NgModule({
...
  bootstrap: [ AppComponent ]
})
...
```

Le cœur d'**angular2** est codé à l'intérieur de **bibliothèques de composants prédéfinis regroupés en modules**.

Les modules des "**bibliothèques**" prédéfinies d'angular2 sont par convention préfixés par "**@angular**".

Exemples :

```
import {Component}      from '@angular/core';
import {Http, Response} from '@angular/http';
import {Observable}      from 'rxjs/Observable';
import {BrowserModule}   from '@angular/platform-browser';
import {OnInit}          from '@angular/core';
import {Router, RouteParams} from '@angular/router';
import {RouteConfig, RouterOutlet} from '@angular/router';
...
```

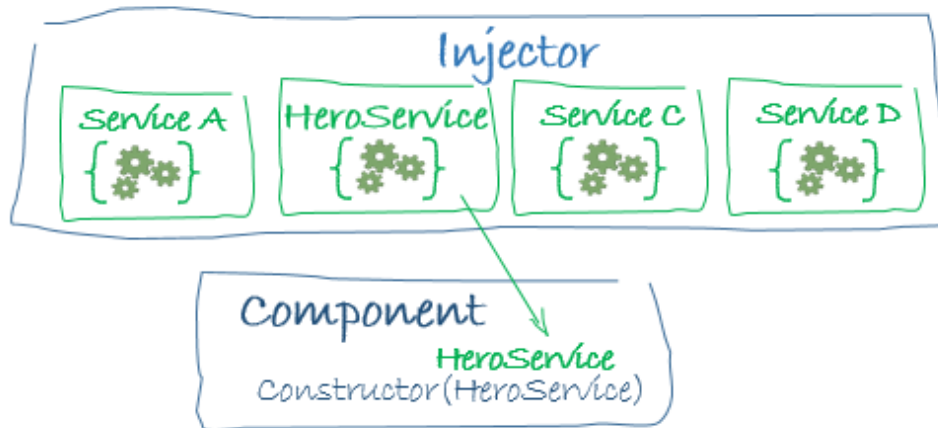
1.3. Principaux types de composants "angular":

Module (répertoire)	Ensemble de composants (généralement placés dans un même répertoire) et chapeauté par une classe décorée via @NgModule .
Component (composant)	Composant visuel de l'application graphique (associé à une vue basée elle-même sur un template) – généralement spécifique à une application
Directive	<p>élément souvent réutilisable permettant de générer dynamiquement une partie de l'arbre DOM (structure de code proche d'un composant) mais plutôt associé à un élément générique paramétrable d'une interface graphique (ex : onglet, histogramme, ...).</p> <p>Une directive s'utilise souvent comme une nouvelle balise ou un nouvel attribut.</p> <p>2 types de directives : "structurelles", "attribut"</p>
Service	Un service est un élément invisible de l'application (qui rend un certain service) en arrière plan (ex : accès aux données, configuration, calculateur, ...)

NB 1 : Un @Component est également associé à un tag (selector) et est techniquement un cas particulier de @Directive.

NB 2 :

Les services sont reliés aux composants via des **injections de dépendances** :



2. Les modules applicatifs

Attention: Entre les premières versions (alpha, bêta) d'angular2 et la première version finale d'angular2, les modules ont changés plusieurs fois de configuration (syntaxe, structure, ...). il faut donc se méfier des documents "Angular2" antérieurs à octobre 2016.

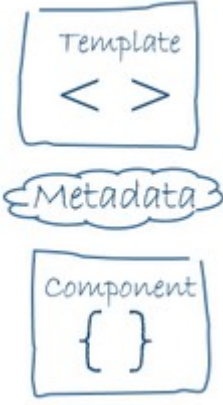
La **classe principale** d'un module Angular2 doit être (par convention) placée dans un fichier `xyz/xyz.module.ts` où `xyz` est le **nom du module** (ex : `"app"`).

Cette classe principale doit être décorée par `@NgModule`.

<p>providers : liste des "composants services" qui seront rendus accessible à l'ensemble des composants de l'application</p> <p>declarations : liste des classes visuelles (composants, directives, pipes) appartenant au module</p> <p>exports : sous ensemble des "déclarations" qui seront visibles par d'autres modules</p> <p>bootstrap : vue principale ("root", "main", ...) (pour module principal seulement)</p>	<pre>import { NgModule } from '@angular/core'; import { BrowserModule } from '@angular/platform-browser'; @NgModule({ imports: [BrowserModule], providers: [Logger], declarations: [AppComponent], exports: [AppComponent], bootstrap: [AppComponent] }) export class XyzModule { }</pre>
---	--

3. Les composants de l'application (@Component)

3.1. Anatomie d'un composant de angular 2

	<p>Un composant "angular 2" est essentiellement constitué d'une classe codant la structure et le comportement de celui-ci (par exemple : réactions aux événements).</p> <p>Les métadonnées sont introduites par une ou plusieurs décorations placées au-dessus de la classe (ex : <code>@Component</code>). <u>Vocabulaire</u> : décoration plutôt qu'annotation.</p> <p>La vue graphique gérée par un composant est essentiellement structurée via un template HTML/CSS comportant des syntaxes spécifiques "angular 2" (<code>*ngFor</code> , <code>{{ }}</code>).</p>
---	---

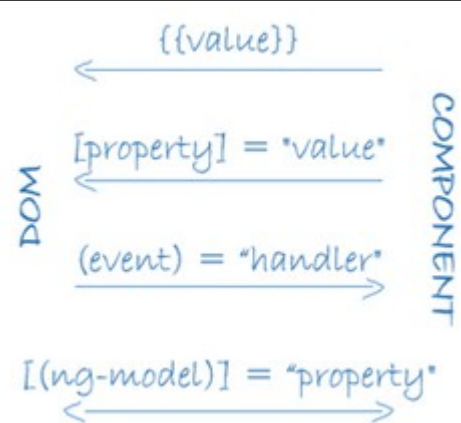
Les liaisons/correspondances gérées par le framework angular2 entre les éléments du composant "orienté objet" et les éléments de la vue "web/HTML/DOM" issue du template sont appelées "**data binding**".

Nuances :

[propertyBinding]

(eventBinding)

[(two-way data binding)]



3.2. @Input et @Output

@Input (du côté sous-composant) permet de récupérer des valeurs (fixes ou variables) qui sont spécifiées par un composant parent/englobant.

Exemple élémentaire :

myheader.component.ts

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

@Component({
  selector: 'my-header',
  template: `<h3>MyHeader {{title}} .. {{fixedValue}}</h3>
    date: {{date}}
    <hr/>
  `
})
```

```

})
export class MyHeaderComponent {
  @Input()
  title : string;

  @Input()
  fixedValue : string;

  date: Date ;

  constructor(){
    this.date = new Date();
  }
}

```

Utilisation depuis un composant parent :

app.component.ts

```

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

@Component({
  selector: 'my-app',
  template: `
    <my-header [title]="headerTitle" fixedValue="***" ></my-header>
    <h1>My First {{message}} .. </h1>
  `
})
export class AppComponent {
  headerTitle :string = 'titre1';
  message: string = "Angular 2 App";
}

```

MyHeader titre1 .. **

date:Mon Dec 12 2016 11:54:09 GMT+0100 (CET)

My First Angular 2 App ..



Eventuelles variantes :

```
@Input('titre') // pour <myheader titre='titre 1' /> (vue externe)
titre : string ; //pour this.titre en interne dans le sous composant
```

@Attribute dans constructeur ressemble un peu à @Input

```
export class Child {
  isChecked;
  constructor(@Attribute("checked") checked) {
    this.isChecked = !(checked === null);
  }
}
```

avec cette utilisation potentielle :

```
<child checked></child>
<child checked='true'></child>
<child></child>
```

@Output (au niveau d'un sous-composant) permet d'indiquer un **événement** qui sera potentiellement remonté et géré par un composant parent/englobant.

Exemple élémentaire :

myheader.component.ts

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

```

@Component({
  selector: 'my-header',
  template: `<h3>MyHeader .. </h3> date:{{date}} -
    <input type='button' (click)="riseEvent($event)" value="evt" /> <hr/>
  `,
})
export class MyHeaderComponent {
  @Output()
  public myEvent : EventEmitter<{value:string}> = new EventEmitter<{value:string}>();

  riseEvent(evt){
    this.myEvent.emit({value:'texte evement'});
  }
}

```

Gestion de l'événement au niveau du composant parent/englobant :

```

@Component({
  selector: 'my-app',
  template: `
    <my-header ... (myEvent)="onMyEvent($event)" ></my-header>
    <h1>My First {{message}} .. {{message2}} </h1>
  `,
})
export class AppComponent {
  ...
  message2: string ;

  onMyEvent(evt){
    console.log(evt);
    this.message2 = evt.value;
  }
}

```

MyHeader titre1 .. **

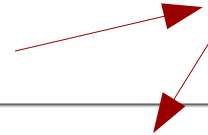
date:Mon Dec 12 2016 15:18:58 GMT+0100 (CET) -

My First Angular 2 App ..

MyHeader titre1 .. **

date:Mon Dec 12 2016 15:18:58 GMT+0100 (CET) -

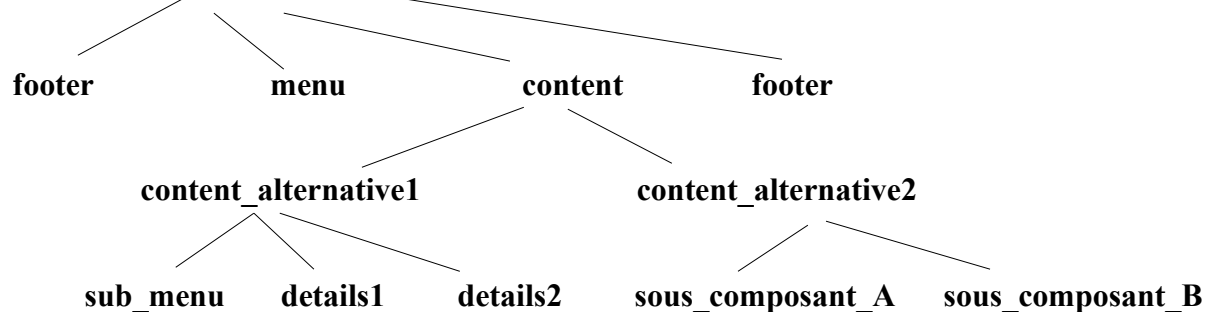
My First Angular 2 App .. texte evement



3.3. logique et structure arborescente d'une application Angular 2

Une application "angular2" est structurée comme un **arbre de composants**.

Exemple : "main_page_component"



Chaque composant englobe :

- un sous template HTML
- une classe (propriétés , méthodes événementielles , ... , plus du \$scope d'angular1 mais "this")
- métadonnées (paramètres du décorateur @Component)

Cette structure arborescente (finalement très classique pour une technologie d'interface graphique – comme DOM , comme JSF) **permet au framework angular2 d'automatiser les points fondamentaux suivants :**

- rendus HTML/DOM via un parcours de l'arbre en profondeur d'abord .
- détection des changements (... , ...)
- ...

4. Cycle de vie sur composants (et directives)

<i>Interfaces (à facultativement implémenter)</i>	<i>Méthodes (une par interface)</i>	<i>Moment où la méthode est appelée automatiquement par angular2</i>
OnChanges	ngOnChanges()	Dès changement de valeur d'un "input ou output binding" (exemple : "propriété initialisée selon niveau parent")
OnInit	ngOnInit()	Après le premier ngOnChanges()
DoCheck	ngDoCheck()	Détection d'un changement personnalisé (selon développeur)
AfterContentInit	ngAfterContentInit()	Après que le contenu d'un composant soit initialisé
AfterContentChecked	ngAfterContentChecked()	Après chaque vérification du contenu d'un

		composant
AfterViewInit	ngAfterViewInit()	Après que la vue associée à un composant soit (ré-)initialisée. Pratique pour "refresh" lorsque l'on retourne à une vue préalablement affichée et quittée.
AfterViewChecked	ngAfterViewChecked()	Après chaque vérification d'une vue associée à un composant
OnDestroy	ngOnDestroy()	Juste avant destruction d'une directive

Exemples :

liste-comptes.component.ts

```
import { Component, EventEmitter, Output, OnInit, AfterViewInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { Compte } from '../compte';
import { CompteService } from '../compte.service';

@Component({
  selector: 'liste-comptes',
  template: `
    <div id="divListeComptes"
      style="background-color:rgb(160,250,160); margin:3px; padding:3px;">
      <h3> liste des comptes </h3>
      <table border="1">
        <tr> <th> numero </th> <th> label </th> <th> solde </th> </tr>
        <tr *ngFor="let cpt of comptes">
          <td style='color : blue'
            (click)="displayLastOperations(cpt.numero)"> {{cpt.numero}} </td>
          <td> {{cpt.label}} </td>
          <td> {{cpt.solde}} </td> </tr>
        </table>
        <i>Un click sur un numero de compte permet de obtenir la liste des dernieres operations</i>
      </div>
    `
})
export class ListeComptesComponent implements OnInit, AfterViewInit {

  @Output()
  public selectedCompteEvent : EventEmitter<{value:number}> =
    new EventEmitter<{value:number}>();

  clientId: number = 0;
  comptes : Compte[] = null ;
  constructor( private _compteService : CompteService,
    private route: ActivatedRoute){
  }
}
```

```
ngOnInit() {  
  this.route.params.forEach((params: Params) => {  
    this.clientId = Number(params['clientId']);  
  });  
}  
  
ngAfterViewInit(){  
  this.fetchComptes(); //refresh  
}  
  
fetchComptes() {  
  this._compteService.getComptesOfClientObservableWithAlternativeTry(this.clientId)  
    .subscribe(comptes => this.comptes = comptes ,  
      error => console.log(error));  
}  
  
displayLastOperations(numCpt : number){  
  console.log("affichage des operations du compte selectionne : " + numCpt);  
  this.selectedCompteEvent.emit({value:numCpt}); // fire event with data  
}  
}
```


VI - Directives , services et injections

1. Service

Un service est un module de code "invisible" comportant des traitements "ré-utilisables" et souvent "partagés" tels que :

- des accès aux données
- des mises à jours de données
-

Exemple simplifié (en version "mocked" sans accès http):

compte.service.ts

```
import { Injectable } from '@angular/core';
//import { Headers, Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable'; // _http.get() return Observable<Response> !!!
import { Compte , Operation , Virement } from './compte';

@Injectable()
export class CompteService {

  public getComptesOfClientObservable(numCli: number) : Observable< Compte[] > {
    return this.mockedComptesObservable; //simulation sans tenir compte de numCli
  }

  //pour test temporaire (sans base):
  private sampleComptes : Compte[] = [
    { "numero" : 1, "label" : "compte 1 (courant , mock)", "solde" : 600.0 },
    { "numero" : 2, "label" : "compte 2 (LDD , mock)", "solde" : 3200.0 },
    { "numero" : 3, "label" : "compte 3 (PEL , mock)", "solde" : 6500.0 } ];

  private mockedComptesObservable = Observable.of(this.sampleComptes);
}
```

NB: Observable<...> ressemble un peu à Promise<...> et se consomme via

```
compteService.getComptesOfClientObservable(this.clientId)
    .subscribe(comptes => this.comptes = comptes ,
               error => console.log(error));
```

Observable (de rxjs) sera étudié de façon plus détaillée au sein d'un chapitre ultérieur (HTTP , ...).

2. Injection de dépendances

A l'époque du framework "angular 1" , l'injection de dépendances consistait à automatiquement relier entre eux deux composants via des correspondances entre "nom de service" et nom d'un paramètre d'une fonction "contrôleur" .

Angular2 gère l'injection de dépendances de manière plus typée et plus orientée objet.

2.1. Enregistrement des éléments qui pourront être injectés:

L'injection de dépendances gérée par Angular2 passe par un enregistrement des fournisseurs de choses à potentiellement injecter.

Ceci s'effectue généralement au moment du "bootstrap" et se configure au niveau de la partie ("providers :") de la décoration **@NgModule** d'un module applicatif .

Exemples :

```
...
@NgModule({
  ...
  providers: [ ConfigService, ClientService ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
```

Si un élément potentiellement injectable n'est pas, globalement, enregistré au niveau global (@NgModule) , il pourra éventuellement être déclaré, localement, au niveau des "providers" spécifiques d'un composant :

```
@Component({
  selector: 'my-app',
  template: `...`,
  providers: [HeroService]
})
export class AppComponent { ...
}
```

La documentation officielle d'Angular 2 parle en terme de "**root_injector**" (pour de niveau @NgModule) et de "**child_injector**" (pour les sous niveaux : @Component , ...)

2.2. Rare récupération directe d'instance via injector.get()

```
import { Injector, ReflectiveInjector } from '@angular/core';
...
var injector : Injector = ReflectiveInjector.resolveAndCreate([ HeroService ]);
                                                                    // ou [ HeroService , S2, S3, ... ]
this._heroService = injector.get(HeroService);
ou bien (selon contexte) :
var hService = injector.get(HeroService);
```

NB1 : Cette utilisation explicite de ReflectiveInjector est censée fonctionner (d'après la documentation officielle de Angular2) . En pratique un bug a été introduit entre les anciennes versions "bêta" et les premières versions officielles (2.1.1 , 2.3.0 , ...).

NB2 : La méthode conseillée (et qui fonctionne toujours) pour l'injection de dépendance est celle qui va suivre (*par constructeur*).

2.3. Injection de dépendance via constructeur

```
@Component({
  selector: 'my-app',
  template: `...` /*,
    providers: [HeroService] nécessaire que si pas déjà enregistré globalement dès le NgModule()
  */
})
export class AppComponent {
  ...
  constructor(private _heroService: HeroService) {

    // this._heroService (de type HeroService) est automatiquement initialisé
    // par injection si métadonnées introduites via @Component ou @Injectable ou ...

  } ;
  ...
}
```

Angular2 initialise automatiquement les éléments passés en argument des constructeurs lorsqu'il le peut (ici par injection du service de type HeroService). Cet automatisme n'est déclenché que si la classe du composant est décorée par @Component() ou @Injectable() ou ...

Pour que des injections de dépendances puissent être gérées au niveau du constructeur de la classe courante :

- au minimum @Injectable() (au sens "sous-composants , sous-services automatiquement injectables")
- assez souvent @Component() (qui peut plus peut moins)

3. Les directives (angular2)

3.1. Les 3 types/niveaux de directives d'angular2:

Attribute Directive	Change l'apparence ou le comportement d'un (souvent seul) élément de l'arbre DOM (exemple <i>ngStyle</i>)
Structural Directive	Change la structure de l'arbre DOM (et donc des éléments affichés) en ajoutant ou supprimant des sous éléments dans l'arbre DOM (exemple : <i>ngIf</i> , <i>ngFor</i> , <i>ngSwitch</i>)
Component	élément composite de l'arbre DOM (selon structure du template HTML associé)

Au final , **@Component** peut être vu comme un cas particulier (et assez fréquent) de directive.

3.2. Directive (de niveau "attribut")

Exemple (tiré du "tutoriel officiel") :

app/highlight.directive.ts

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

@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

Le paramétrage le plus important est la décoration **@Directive** .

Le nom du **sélecteur** CSS doit être encadré par des **crochets** lorsqu' il s'agit d'une directive.

el de type **ElementRef** correspond à un élément de l'arbre DOM dont il faut mettre à jour le rendu.

Exemple d'utilisation :

app/app.module.ts

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

```
...
import {HighlightDirective} from './highlight.directive'

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

app/app.component.ts

```
import {Component} from 'angular2/core';
@Component({
  selector: 'my-app',
  template: ' ... <span myHighlight>Highlight me!</span>'
})
export class AppComponent { }
```

Résultat:

My First Angular 2 App

Highlight me!Version améliorée (avec paramètre via @Input et gestion d'événements) :

```
import {Directive, ElementRef, HostListener, Input} from '@angular/core';
@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
  @Input('myHighlight')
  public highlightColor: string;

  private _defaultColor = 'red';

  @Input() set defaultColor(colorName:string){
    this._defaultColor = colorName || this._defaultColor;
  }

  constructor(private el: ElementRef) {
  }

  @HostListener('mouseenter')
```

```

onMouseEnter() { this._highlight(this.highlightColor || this._defaultColor); }

@HostListener('mouseleave')
onMouseLeave() { this._highlight(null); }

private _highlight(color: string)
{ el.nativeElement.style.backgroundColor = color; }
}

```

NB :

Sans argument, `@Input()` fait que la propriété exposée a le même nom que celle de la classe (public ou get / set).

Avec un argument , `@Input` permet de préciser un alias sur la propriété exposée (ex : 'myHighlight' plutôt que `highlightColor`).

@HostListener permet d'associer des noms d'événements (déclenchés sur l'élément DOM courant) à une méthode événementielle .

Utilisation :

```
<p [myHighlight]=" 'yellow' " [defaultColor]=" 'violet' " >Highlight me!</p>
```

ou bien (plus simplement) :

```
<p myHighlight=" yellow " defaultColor=" violet " >Highlight me!</p>
```

Pick a highlight color

☐ Green ☐ Yellow ☐ Cyan

ou bien (avec un choix dynamique) :

```

<h4>Pick a highlight color</h4>
<div>
  <input type="radio" name="colors" (click)="color='lightgreen' " id="r1" />
  <label for="r1">Green</label>
  <input type="radio" name="colors" (click)="color='yellow' " id="r2" />
  <label for="r2">Yellow</label>
  <input type="radio" name="colors" (click)="color='cyan' " id="r3" />
  <label for="r3">Cyan</label>
</div>
<span [myHighlight]="color"> Highlight with choosen color</span> <br/>

```

3.3. Directive structurelle

Une directive structurelle (ajoutant ou retirant des sous éléments dans l'arbre DOM) se programme de façon très semblable à une directive d'attribut (même décoration `@Directive` , même syntaxe (avec crochets) pour le sélecteur CSS) . La principale différence tient dans les éléments injectés dans le constructeur :

- `TemplateRef` correspond à la branche des sous éléments imbriqués (à supprimer ou ajouter ou ...)
- `ViewContainerRef` permet de contrôler dynamiquement le contenu via des méthodes prédéfinies telles que `.clear()` ou `.createEmbeddedView()`

Exemple "myUnless" (tiré du tutoriel officiel) :

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

@Directive ({ selector: '[myUnless]' })
export class UnlessDirective {

  constructor( private _templateRef: TemplateRef,
               private _viewContainer: ViewContainerRef ) {}

  @Input() set myUnless(condition: boolean) {
    if (!condition) { this._viewContainer.createEmbeddedView(this._templateRef); }
    else { this._viewContainer.clear(); }
  }
}
```

Utilisation (comme `*ngIf`) :

age: <input type='text' [(ngModel)]="age" />

<p *myUnless="age>=18">

condition "age>=18" is false and myUnless is true.

</p>

<p *myUnless="!(age>=18)">

condition "age>=18" is true and myUnless is false.

</p>

3.4. Exemple de directive basée sur @Component

Ce premier exemple est assez spécifique (et peu réutilisable).

hero-detail.component.ts

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

/*
'hero' property of HeroDetail (Sub)Component must be declared as part of "@Input()"
and this HeroDetailComponent should be declared in app.module.ts
*/

@Component({
  selector: 'my-hero-detail',
  template: `
    <div *ngIf="hero">
      <h2>{{hero.name}} details!</h2>
      <div><label>id: </label>{{hero.id}}</div>
      <div>
        <label>name: </label>
        <input [(ngModel)]="hero.name" placeholder="name"/>
      </div>
    </div>
  `,
})
export class HeroDetailComponent {
  @Input()
  public hero: Hero;
}
```

3.5. Utilisation d'une directive dans un composant parent

Le composant précédent peut s'utiliser comme une directive en l'important et en le renseignant dans la liste des directives/composants utilisés au sein de @NgModule() de app.module.ts

```
...
import {HeroDetailComponent} from './hero-detail.component';

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

Pour y faire référence au sein d'un template , il suffit d'utiliser la balise correspondant à son

sélecteur (ici `<my-hero-detail></my-hero-detail>`).

Finalement , pour renseigner la valeur de la propriété `hero` déclarée que '@Input()' de la directive , il suffit d'utiliser la syntaxe habituelle [] pour affecter le résultat d'une expression à un attribut ou propriété d'une balise (prédéfinie ou spécifique/directive) :

```
<my-hero-detail [hero]="selectedHero" > </my-hero-detail>
```

Exemple :

```
import {Component} from '@angular2/core';
import {Hero} from './hero';

@Component({
  selector: 'my-app',
  template:`
    <h1>{{title}}</h1>
    <h2>My Heroes</h2>
    <ul class="heroes">
      <li *ngFor="let hero of heroes" [class.selected]="hero === selectedHero"
        (click)="onSelect(hero)">
        <span class="badge">{{hero.id}}</span> {{hero.name}}
      </li>
    </ul>

    <my-hero-detail [hero]="selectedHero" > </my-hero-detail>
  `,
  styles:[
    .selected { width: 150; background-color: yellow; color: blue; }
    .heroes { color: green; }
    .heroes li:hover { color: red; background-color: white; }
    .heroes .badge { color: white; background-color: green; }
  ]
})
export class AppComponent {
  public title :string = 'Tour of Heroes';
  public heroes : Hero[] = HEROES;
  public selectedHero: Hero;

  onSelect(hero: Hero) { this.selectedHero = hero; }
}
...
```

VII - Switch et routing (navigations)

1. Navigation via ngSwitch et router de angular2

1.1. Paramétrage dans index.html

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

```

1.2. Switch (local) de sous-templates :

```
<div [ngSwitch]="variableXy">
  <div *ngSwitchCase="'case1Exp'">...</div>
  <div *ngSwitchCase="'case2LiteralString'">...</div>
  <div *ngSwitchDefault>...</div>
</div>
```

NB : Attention à bien placer des "simples quotes" dans les "doubles quotes" .

Ceci permet simplement de switcher de "détails à afficher" (et donc souvent de sous-composant).
On reste dans un même composant principal . Pas de changement d'URL .

1.3. navigation avec changement d'url

De façon à naviguer efficacement (tout en pouvant enregistrer des "bookmarks" sur une des parties de l'application) , il faut utiliser le "routeur" d'angular2 qui sert à basculer de composants (ou sous composant) tout en gérant bien les URLs/Paths relatifs .

Le service de routage est prise en charge par le mode "*RouterModule*" que l'on peut directement enregistrer dans le fichier `app.module.ts` en même temps qu'une définition simple de route(s) via la syntaxe `RouterModule.forRoot(arrayOfPaths)` :

```
...
import { RouterModule } from '@angular/router';
...
@NgModule({
  imports: [ BrowserModule , FormsModule , HttpModule,
```

```

    RouterModule.forRoot([
      { path: 'Welcome', component: WelcomeComponent }
    ]),
    declarations: [ AppComponent , ... ],
    providers: [ XyService ],
    bootstrap: [ AppComponent ]
  })
export class AppModule { }

```

Ceci dit , lorsque sur une application sérieuse , la définition des routes devient plus complexe, il est alors conseillé (et habituel) d'externaliser cela dans un fichier ad-hoc "**app-routing.module.ts**" :

app-routing.module.ts

```

import { NgModule }      from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { WelcomeComponent } from './welcome.component';
import { IdentificationComponent } from './identification.component';
import { IdentifieComponent } from './identifie.component';

const routes: Routes = [
  { path: 'welcome', component: WelcomeComponent },
  { path: '', redirectTo: '/welcome', pathMatch: 'full'},
  { path: 'identification', component: IdentificationComponent },
  { path: 'clientIdentifie/:clientId', component: IdentifieComponent }
];

@NgModule({
  imports: [ RouterModule.forRoot(routes) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule {}

```

et d'un point de vue routage, dans le fichier principal du module **app.module.ts** il ne reste plus que :

```

import { AppRoutingModule } from './app-routing.module';
....
@NgModule({
  imports: [ BrowserModule , FormsModule , HttpModule, AppRoutingModule , ... ],

```

```
...})
export class AppModule { }
```

1.4. router-outlet

router-outlet = partie d'un template qui changera de contenu selon la route courante .

Exemple :

```
@Component({
  selector: 'my-app',
  template: `
    <header id="mainHeader" role="banner">
      <h3>Minibank App </h3>
    </header>
    <div id="mainArea" >
      <section id="simpleMainContent" >
        <b>router-outlet</b>
      </section>
    </div>
    <footer id="mainFooter" >
      ... status , mentions legales , ... <a routerLink='welcome'> retour accueil </a>
    </footer>
  `,
  providers: []
})
```

Dans la plupart des cas simple/ordinaire comme celui-ci , un seul router-outlet (sans nom) suffit.

Dans certains cas sophistiqués et bien structurés , il est possible qu'un des sous-composants comporte en lui un autre <router-outlet> (à un niveau imbriqué) pour ainsi pouvoir switcher de sous-sous-composant .

Dans des cas très complexes, il est possible de configurer des "router-outlet" annexes (avec des noms) et de leurs associer des contenus variables selon un suffixe particulier placé en fin d'URL.

1.5. Routage simple (sans paramètres)

Au sein de **app-routing.module.js**

```
const routes: Routes = [
```

```
{ path: 'welcome', component: WelcomeComponent },
{ path: '', redirectTo: '/welcome', pathMatch: 'full'},
{ path: 'identification', component: IdentificationComponent } ,
];
```

suffit pour se retrouver automatiquement redirigé de index.html vers l'URL .../welcome (d'après la seconde règle avec redirectTo:) .

La première route associe le composant "WelcomeComponent" à la fin d'url "welcome" et dans ce cas la balise <router-outlet></router-outlet> sera remplacé par le contenu (template) du composant "WelcomeComponent" .

Finalement un click sur un lien hypertexte dont l'url relative est "identification" (ou bien une navigation équivalente) déclenchera automatiquement un basculement de sous composant (le template de "IdentificationComponent" sera affiché au niveau de <router-outlet></router-outlet>).

1.6. Déclenchement d'une navigation avec paramètre

```
// dans app-routing.module.ts
const routes: Routes = [
...
  { path: 'yy/:yyId', component: YyComponent }
];
```

Solution 1 (par méthode événementielle) :

```
import {Router} from '@angular/router';
...
<button (click)="onNavigate()" > vers yy </button>
et
export class XxComponent {
  numYy : number;
  constructor(private _router: Router){
  }
  onNavigate():void {
    let link = ['/yy', this.numYy];
    this._router.navigate( link );
    // ou bien this._router.navigateByUrl(`yy/${this.numYy}`);
    //avec quote inverse `...` !!!
  }
}
```

Solution 2 (via paramètre de la directive routerLink) :

```
<a [routerLink]="['/yy', numYy ]" ...> vers yy </a>
```

1.7. Récupération du paramètre accompagnant la navigation :

```
import {Component , OnInit} from '@angular/core';
import { ActivatedRoute, Params} from '@angular/router';
import { Location } from '@angular/common';
...
export class YyComponent implements OnInit{
  message : string = "...";
  yId: number ;
  y : Yy;
  constructor(private _yyService : YyService,
               private _route: ActivatedRoute,
               private _location: Location){
  }
  ngOnInit() {
    this._route.params.subscribe((params: Params) => {
      this.yId = Number(params['yyId']); //où 'yyId' est le nom du paramètre
                                         // dans l'expression de la route
                                         // (fichier app-routing.module.js)
    });
    this.fetchYy();
  }

  fetchYy(){
    this._yyService.getYyObservable(this.yId).subscribe(yy=>this.y = yy ,
                                                         error => this.message = <any>error);
  }

  goBack(): void {
    this._location.back();
  }
}
```

```
}  
}
```

VIII - Appels de W.S. REST (Observable, ...)

1. Angular et dialogues HTTP/REST

1.1. Traditionnel XMLHttpRequest ou librairie "fetch" de ES6

Une application Angular s'exécute au sein d'un navigateur web.

Si celui-ci est ancien , il est toujours possible d'utiliser le traditionnel XMLHttpRequest permettant d'effectuer des appels ajax. (le \$.ajax() de jQuery est une ancienne référence) .

Seuls les navigateurs les plus récents permettent l'utilisation native de la nouvelle librairie "fetch" .

AngularJs (1.x) s'appuyait en interne sur jQuery Lite

Angular (2.x) ne s'appuie plus sur jQuery et propose un service "http" de haut niveau qui permet de se détacher un peu des couches basses (XMLHttpRequest ou fetch ou ...).

Il vaut mieux donc se reposer sur le service HTTP de Google/Angular2 et laisser le soin au code interne du framework de s'adapter aux technologies de bas niveau au fur et à mesure des nouvelles versions.

1.2. Promise ou Observable ?

Les habitués de AngularJs (1.x) connaissent bien les "Promise" . Celles-ci sont encore utilisables avec Angular2 (avec quelques adaptations).

Pour des développements nouveaux, on pourra cependant préférer la nouvelle api "Observable" de rxjs qui est par certains cotés plus évoluée que les "Promise" et qui s'utilise de manière très similaire.

"Promise" ou "Observable" est un choix pour notre code de haut niveau.

Le service Http de Angular2 peut s'adapter aux deux modes et les appels de bas niveaux seront effectués de la même façon.

NB : Dans la suite de ce chapitre , l'api RxJs et les "Observable" seront mis en avant
Les "Promises" sont placées dans une des annexes

Le service prédéfini Http de angular >=2 retourne des "**Observable<Response>**" que l'on peut éventuellement transformer en **Promise<...>** via la méthode **.toPromise()** .

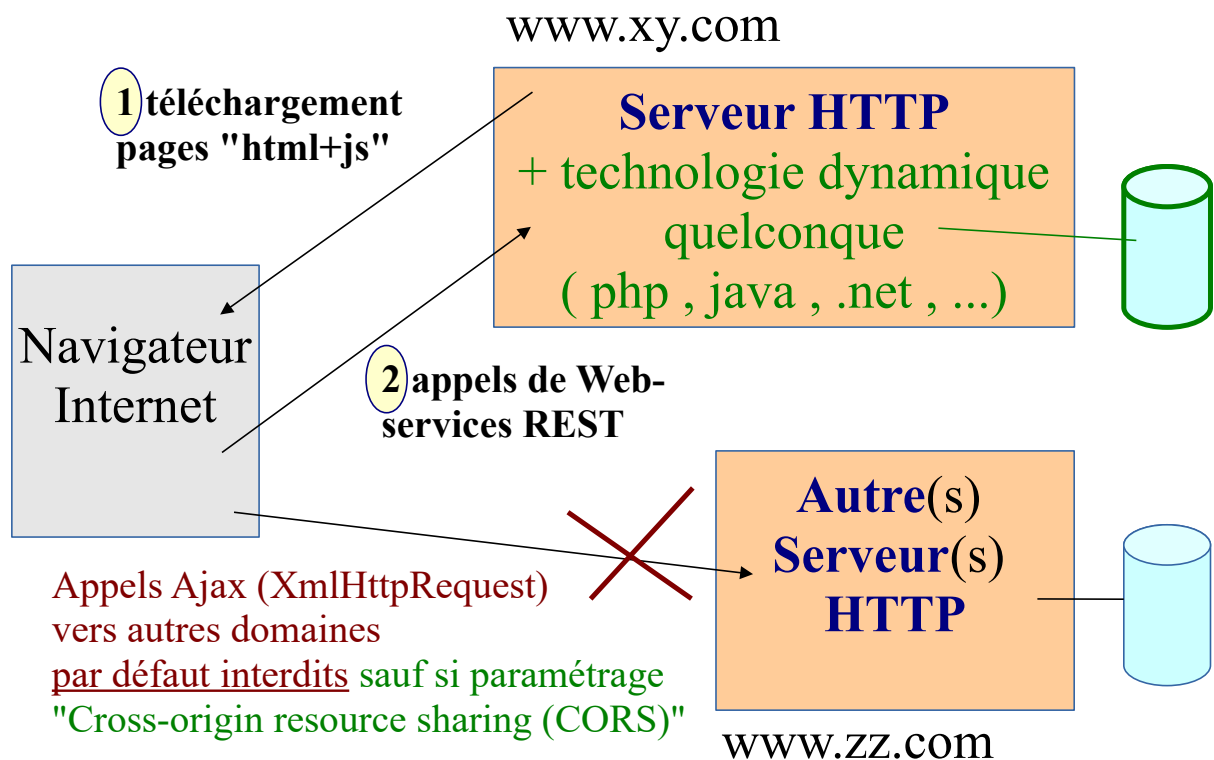
1.3. Contexte des appels HTTP/REST et CORS

Dès qu'une application Angular2 a besoin de récupérer des données via HTTP/REST, elle devient tributaire des contraintes de communications entre le navigateur web et le serveur HTTP en arrière plan.

Rappel important : si les appels HTTP/ajax sont effectués vers un autre nom de domaine il faudra prévoir des autorisations "**CORS**" du côté des web-services REST côté serveur (ex : nodeJs/Express ou Java/JEE/JaxRS ou SpringMvc) .

D'autre part, beaucoup de web-services REST nécessitent des paramètres de sécurité pour pouvoir être invoqués (ex : jetons/tokens, ...) . Une adaptation au cas par cas sera souvent à prévoir.

Cadre des appels "html/js/ajax" vers services REST



// Exemple : **CORS** enabled with express/node-js :

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*"); // "*" ou "xy.com , ..."
  res.header("Access-Control-Allow-Methods", "POST, GET, PUT, DELETE, OPTIONS"); //default: GET, ...
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept, Authorization");
  next();
});
```

1.4. Simulation d'appels HTTP via angular-in-memory-web-api

Angular (v>=2) propose une api prédéfinie nommée "**angular-in-memory-web-api**" qui permet de simuler un dialogue HTTP/REST avec un service web.

Avantages :

- * possibilité d'effectuer des tests avec angular2 seulement (pas besoin de nodeJs ou autre)
- * documenté (avec exemples) sur le site officiel de Angular2
- * tests possibles en écriture/mise à jour, suppression

Inconvénients (à court terme, version actuelle) :

- * quelques limitations importantes (structures de données, ...)
- * La valeur de retour est encapsulée par un niveau intermédiaire ".data" . Ce qui n'est pas toujours transposable avec un réel web service existant.
- * il faudra restructurer (de façon non négligeable) la configuration du module pour basculer sur l'appel de véritables services web externes
- * en connaissant bien une technologie serveur (ex : nodeJs/express/Mongoose ou Java/Jee/Jax-Rs) on va presque aussi vite à développer un véritable service externe.

Détails dans une des annexes ou ...

1.5. Simulation d'invocation de WS REST via Observable.of()

Observable.of(...) permet de retourner immédiatement un jeu de données (en tant que simulation d'une réponse à un appel HTTP en mode get) .

1.6. Simulation d'invocation de WS REST via "http mock ou équivalent"

D'autres solutions sont disponibles pour simuler des appels HTTP ...

2. Utilisation du service Http avec Observable (rxjs)

Angular (v2,v4) met clairement en avant la nouvelle api tierce-partie rxjs et "Observable" .

Plus évoluée que l'api "Promise" , rxjs/Observable est cependant aussi simple d'utilisation et apporte (au sein d'angular v2.x) les avantages suivants :

- * **possibilité d'effectuer plusieurs traitements lorsqu'un sujet "Observable" est prêt** (d'un point de vue *asynchrone*)
- * **"Observable" est le format "par défaut" de l'api "Http" de angular (v2.x)**
http.get() retourne Observable<Response> !!!
- * **l'api "rxjs" offre tout un tas de combinaisons** (programmation fonctionnelle asynchrone avec "lambda expression") , est extensible et existe même au dehors de js/javascript (il existe une version "java") .

Configuration nécessaire pour rxjs/Observable au niveau du module :

app.module.ts

```
import './rxjs-extensions';

import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/http';
// AppConfig facultatif mais pratique pour une configuration souple/modulaire/évolutive :
import { AppConfig, MY_APP_CONFIG_TOKEN, MY_PROD_APP_CONFIG,
        MY_DEV_APP_CONFIG } from './app.config';

import { ClientService } from './client.service'; import { CompteService } from './compte.service';

@NgModule({
  imports: [ ..., HttpClientModule ],
  declarations: [ AppComponent , ... ],
  providers: [ { provide: MY_APP_CONFIG_TOKEN, useValue: MY_DEV_APP_CONFIG },
               ClientService , CompteService ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

NB: Le fichier *rxjs-extensions.ts* (rassemblant plein de import) est facultatif mais conseillé .
 Toute la partie " AppConfig , MY_APP_CONFIG_TOKEN , MY_DEV_APP_CONFIG " ci-dessus en italique est facultative et permet simplement un basculement de configuration.

rxjs-extensions.ts

```
// Observable class extensions
import 'rxjs/add/observable/of';           import 'rxjs/add/observable/throw';
// Observable operators
import 'rxjs/add/operator/catch';          import 'rxjs/add/operator/toPromise';
import 'rxjs/add/operator/do';             import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';            import 'rxjs/add/operator/switchMap';
```

Eventuelle configuration variable/modulable de l'application :**app.config.ts**

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

export interface AppConfig {
  api_base_url: string;
  alternative_api_base_url: string;
  mode: string;
}

export const MY_PROD_APP_CONFIG : AppConfig = {
  "api_base_url": "http://www.xyz.com:80/minibank" ,
  "alternative_api_base_url": "http://www.xyz.alt.com:80/minibank" ,
  "mode": "production"
};

export const MY_DEV_APP_CONFIG : AppConfig = {
  "api_base_url": "http://localhost:8282/minibank" ,
  "alternative_api_base_url": "app/data" ,
  "mode": "development"
};

//export const MY_APP_CONFIG_V3: AppConfig = { ... };

export const MY_APP_CONFIG_TOKEN = new OpaqueToken('app.config');
```

Le basculement de configuration s'effectue simplement en changeant la valeur de *useValue* au sein de **app.module.ts** :

```
{ provide: MY_APP_CONFIG_TOKEN, useValue: MY_DEV_APP_CONFIG }
```

2.1. Appel Http/get et réponse Observable

compte.service.ts

```
import {Injectable,Inject} from '@angular/core';
import {Headers, Http, Response} from '@angular/http';
import {Observable} from 'rxjs/Observable'; // _http.get() return Observable<Response> !!!
import {Compte , Operation , Virement} from './compte';
import { AppConfig , MY_APP_CONFIG_TOKEN} from './app.config';

@Injectable()
export class CompteService {
  private _headers = new Headers({'Content-Type': 'application/json'});
  private _compteUrlPart :string = "/comptes" ; // + ?numClient=...'; // REST call
  constructor (private _http: Http ,
    @Inject(MY_APP_CONFIG_TOKEN) private myAppConfig: AppConfig ) {
  }

  public getComptesOfClientObservable(numCli: number) : Observable< Compte[] > {
    let comptesUrl : string = null;
    comptesUrl = this.myAppConfig.api_base_url + this._compteUrlPart
      + "?numClient=" + numCli;
    console.log( "comptesUrl = " + comptesUrl);
    return this._http.get(comptesUrl )
      .map(response => <Compte[]> response.json() )
      .catch(e => { return Observable.throw('error:' + e);});
  } ... }
```

Exemple d'appel :

```
export class ListeComptesComponent implements OnInit, AfterViewInit{ ...
  ngAfterViewInit() { this.fetchComptes();//refresh  }
  fetchComptes() {   this._compteService.getComptesOfClientObservable(this.clientId)
    .subscribe(comptes =>this.comptes = comptes ,
      error => console.log(error));
  }
  ...}
```

2.2. Modes post, ... avec Observable

```
//demande de Virement (POST)
export class Virement {
  constructor(
    public montant : number,
    public numCptDeb: number,
    public numCptCred : number,
    public ok: boolean
  ) {}
}
```

```
...
export class CompteService {
  ....
  private _virementUrlPart :string = "/virement" // POST REST call

  public postVirementObservable(virement: Virement): Observable<Virement> {
    let virementUrl :string = null;
    virementUrl = this.myAppConfig.api_base_url + this._virementUrlPart;
    console.log( "virementUrl = " + virementUrl);
    return this._http
      .post(virementUrl, JSON.stringify(virement), {headers: this._headers})
      .map(res => <Virement> res.json())
      .catch(e => {return Observable.throw('error:' + e)});
  }
  ...
}
```

Exemple appel :

param-virement.component.ts

```
import {Component , Output, EventEmitter} from '@angular/core';
import {Virement} from '../compte';
import {CompteService} from '../compte.service';
```

```

@Component({
  selector:'param-virement',
  template:` <div id="divVirement" style="..." >    <h3> parametrage virement </h3>
    ... <input id="montant" [(ngModel)]="transfert.montant" /> <br/>
    ...<input id="numCptDeb" [(ngModel)]="transfert.numCptDeb" /> <br/>
    ... <input id="numCptCred" [(ngModel)]="transfert.numCptCred" /> <br/>
    <button (click)="doVirementAndRefresh()">effectuer le virement</button>
  </div> ` })
export class ParamVirementComponent {
  @Output()
  public virementOk: EventEmitter<{value:string}> = new EventEmitter<{value:string}>();
  message : string ;
  transfert: Virement = { "montant": 0 , "numCptDeb": 1 , "numCptCred": 2 , "ok":false};
  constructor(private _compteService : CompteService){
  }

  private setAndLogMessage( virementOk : boolean){
    if(virementOk){ this.message = "le montant de " + this.transfert.montant +
      " a bien ete transfere du compte " + this.transfert.numCptDeb +
      " vers le compte " + this.transfert.numCptCred; }
    else {this.message = "echec virement"; }
    console.log(this.message);
  }

  doVirementAndRefresh(){
    console.log("doVirementAndRefresh() : " + this.transfert.montant );
    this._compteService.postVirementObservable(this.transfert)
      .subscribe(transfertEffectue =>{
        if(transfertEffectue.ok) { this.setAndLogMessage(true);
          this.virementOk.emit({value:this.message}); /*fire event with data*/
        } else { this.setAndLogMessage(false); } ,
        error => console.log(error));
      }
    }
}

```

3. Retransmission des éléments de sécurité

Un ensemble de Web services "REST" (ou "API Rest" ou "Restful Api") est généralement sécurisé sur les bases suivantes :

- HTTPS
- jeton (token) d'authentification véhiculé en mode "Bearer" ou autre et au format uuid ou jwt ou autre
- autre(s) champ(s) de l'entête HTTP (ex : `_csrf`, `X-XSRF-TOKEN`, ...)

La documentation officielle du framework Angular (v2, v4, ...) indique que certains champs de l'entête HTTP sont automatiquement / implicitement retransmis au sein des requêtes ultérieures.

Il peut cependant être nécessaire de retransmettre explicitement certaines informations reçues (ex : jeton/token).

Exemple (à adapter au contexte et peaufiner) :

A la réception de la réponse d'un WS d'authentification retournant ici un token dans une structure/classe "VerifAuth" spécifique à une certaine Api REST :

```
...
storeTokenInLocalStorage(va:VerifAuth){
  if(va && va.token){
    console.log('received token='+va.token);
    localStorage.setItem('token',va.token);
  }
}
```

Retransmission standardisé du jeton en mode "Bearer" dans le champ "Authorization" :

```
...
private _basicHeaders = new Headers({'Content-Type': 'application/json'});
private _headers = this._basicHeaders;

loadTokenFromLocalStorage(){
  var token = localStorage.getItem('token');
  if(token){
    this._headers= this._basicHeaders;
    this._headers.append('Authorization','Bearer ' + token);
  }
}

...
this.loadTokenFromLocalStorage();
return this._http.get(urlWsRest,{headers: this._headers})
  .map(response => response.json()).catch(e => Observable.throw(e));
...
```


IX - Contrôles de formulaires , composants GUI

1. Contrôle des formulaires

1.1. les différentes approches (template-driven , model-driven,...)

Approches	Caractéristiques
template-driven	Simple paramétrages , pas de code javascript
model-driven (alias reactive-forms)	Meilleure façon de tester le comportement (sans DOM dans navigateur) ,
via Form-builder API	Approche sophistiquée et complexe pour cas pointus

L'approche la **plus simple** et la **plus classique** est "**template-driven**".

Rappel (configuration nécessaire dans le module) :

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
...
@NgModule({
  imports: [ BrowserModule , FormsModule , ... ],
  declarations: [ AppComponent , ],
  providers: [ ... ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
}
```

1.2. Exemples de paramétrages (ngForm)

```
<form #formXy="ngForm" >
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control"
      [(ngModel)]="model.name" #spy required />
    <br/>temp class name: {{spy.className}}
  </div>
  ....
  <button type="submit" class="btn btn-default">Submit</button>
</form>
```

<form #formXy="ngForm" > piste les changements qui peuvent avoir lieu sur les champs d'un formulaire.

NB : lorsqu'un template est pris en charge par angular (v2.x) , la directive **ngForm** est automatiquement appliquée sur la balise **<form>**. Le seul intérêt d'écrire explicitement

<form #formXy="ngForm" > est de pouvoir écrire **<button type="submit" class="btn btn-default" [disabled]="!formXy.form.valid">Submit</button>**

Un des effets de **ngForm** sur un formulaire consiste à automatiquement associer des classes de styles css au champ de saisie :

Etat	flag (booléen)	Css class si true	Css class si false
Champ visité (souris entrée et sortie)	<i>touched</i>	ng-touched	ng-untouched
Valeur du champ modifiée	<i>dirty</i>	ng-dirty	ng-pristine
Valeur du champ valide	<i>valid</i>	ng-valid	ng-invalid

Exemples de **styles.css**

```
.ng-valid[required] {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid {
  border-left: 5px solid #a94442; /* red */
}
```

Name

Dr IQ

temp class name: form-control ng-pristine ng-valid ng-touched

(si visité)

temp class name: form-control ng-untouched ng-pristine ng-valid

(si pas visité)

Name

Hercule

temp class name: form-control ng-valid ng-touched ng-dirty

lorsque modifié

Name

temp class name: form-control ng-touched ng-dirty ng-invalid

si invalide

Dr IQ

Valid + Required

Chuck Overstreet

Valid + Optional

Invalid (required | optional)

Messages d'erreurs:

En déclarant une variable locale de référence associée à l'objet du champ de saisie via la syntaxe **#nameFormCtrl="ngModel"** , on peut afficher de façon conditionnée certains messages d'erreurs :

```
<input type="text" class="form-control"
[(ngModel)]="model.name" #nameFormCtrl="ngModel" required />

<div [hidden]="nameFormCtrl.valid" class="alert alert-danger">
  Name is required
</div>
```

nameFormCtrl.**valid** (true or false)nameFormCtrl.**dirty** (true or false)nameFormCtrl.**touched** (true or false)

Soumission d'un formulaire:

```

<div [hidden]="submitted">
<h1>Coords Form</h1>
<form (ngSubmit)="onSubmit()" #formXy="ngForm">
....
<button type="submit" class="btn btn-default"
        [disabled]="!formXy.form.valid">Submit</button>
</form>
</div>

<div [hidden]="!submitted">
... <!-- actions après la soumission du formulaire -->
</div>

```

```

import {Component} from '@angular/core';
import { Coords } from './coords';
@Component({
  moduleId: module.id,
  selector: 'coords-form',
  templateUrl: 'coords-form.component.html'
})
export class CoordsFormComponent {
  titles = ['Mr', 'Ms'];

  model = new Coords(1,this.titles[0], 'PowerUser', '1969-07-11', 'good level');
  submitted = false;
  onSubmit() { this.submitted = true; // ou autre }
  get diagnostic() { return JSON.stringify(this.model); }
}

```

NB: *moduleId: module.id* permet d'interpréter templateUrl en relatif par rapport au module courant (ici HeroFormComponent).

==> le bouton "submit" du formulaire est automatiquement désactivé lorsque le formulaire comporte au moins un champ invalide .

X - Aspects divers de angular (pipes, ...)

1. BehaviorSubject

NB: un objet de type **BehaviorSubject**<...> doit avoir une valeur initiale dès sa construction. C'est une chose "**Observable**" depuis plusieurs composants de l'application.

Dès que la valeur sera modifiée , tous les observateurs seront automatiquement synchronisés.

Exemple :

caddy.service.ts

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

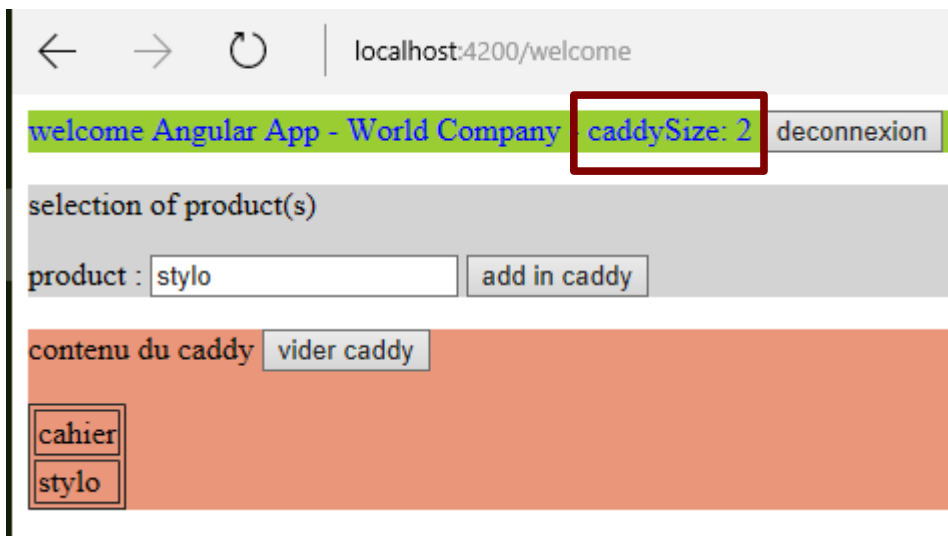
@Injectable()
export class CaddyService {
  private _compteur : number = 0;
  private _caddyContent : string[] = [];

  public bsCompteur : BehaviorSubject<number>
    = new BehaviorSubject<number>(this._compteur);
  public bsCaddyContent : BehaviorSubject<string[]>
    = new BehaviorSubject<string[]>(this._caddyContent);

  constructor() {
    //....subscribe(callBackOnNext(nextValue) , callBackOnError(err) , callBackOnCompleted());
    this.bsCompteur.subscribe(
      nextValueOfCompteur => this._compteur = nextValueOfCompteur);
  }

  public addElementInCaddy(productName : string){
    this._caddyContent.push(productName);
    this.bsCaddyContent.next(this._caddyContent);
    this._compteur++;
    this.bsCompteur.next(this._compteur);
  }

  public clearCaddy(){
    this._caddyContent = [];
    this.bsCaddyContent.next(this._caddyContent);
    this._compteur=0;
    this.bsCompteur.next(this._compteur);
  }
}
```



dans *my-header.component.html*

```
.... caddySize: {{caddySize}} ....
```

dans *my-header.component.ts*

```
...
export class MyHeaderComponent implements OnInit {
  caddySize : number = 0;

  constructor(private caddyService : CaddyService ){
    caddyService.bsCompteur.subscribe(
      nextValueOfCompteur => this.caddySize = nextValueOfCompteur);
  }
  ...
}
```

selection.component.html

```
<p> selection of product(s) </p>
product : <input [(ngModel)]="productName" />
<button (click)="onAddInCaddy($event)" >add in caddy</button>
```

selection.component.ts

```
import { Component, OnInit } from '@angular/core';
import { CaddyService } from "app/caddy.service";
```

```

...
export class SelectionComponent implements OnInit {
  productName : string = "?";

  constructor(private caddyService : CaddyService ){ }
  ngOnInit() { }

  onAddInCaddy(evt){
    //appel indirect de bsCompteur.next(++....); et donc déclenchement de tous les subscribe(...)
    this.caddyService.addElementInCaddy(this.productName);
  }
}

```

commande.component.html

```

<p>contenu du caddy <button (click)="onViderCaddy()">vider caddy</button></p>
<table border="1">
  <tr *ngFor="let e of contenuCaddy"><td>{{e}}</td></tr>
</table>

```

commande.component.ts

```

import { Component, OnInit } from '@angular/core';
import { CaddyService } from "app/caddy.service";

@Component({
  selector: 'app-commande',
  templateUrl: './commande.component.html',
  styleUrls: ['./commande.component.css']
})
export class CommandeComponent implements OnInit {
  private contenuCaddy : string[];

  constructor(private caddyService : CaddyService ){
    caddyService.bsCaddyContent.subscribe(
      caddyContent => this.contenuCaddy = caddyContent);
  }

  ngOnInit() { }

  onViderCaddy(){
    this.caddyService.clearCaddy();
  }
}

```

Eventuelle combinaison "BehaviorSubject + LocalStorage"

En cas de "refresh" déclenché (quelquefois involontairement) pas l'utilisateur (via F5 ou autre), tout le contenu en mémoire de l'application angular est réinitialisé et potentiellement perdu .

Pour ne pas perdre le contenu (caddy ou autre) dans un tel cas , le "localStorage" d'HTML5 peut éventuellement être une solution.

caddy.service.ts

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

@Injectable()
export class CaddyService {
  private _compteur : number = 0;    private _caddyContent : string[] = [];
  public bsCompteur : BehaviorSubject<number>
    = new BehaviorSubject<number>(this._compteur);
  public bsCaddyContent : BehaviorSubject<string[]>
    = new BehaviorSubject<string[]>(this._caddyContent);

  constructor() {
    this.bsCompteur.subscribe( nextValueOfCompteur => this._compteur = nextValueOfCompteur);
    this.tryReloadCaddyContentFromLocalStorage();
    this.subscribeCaddyStoringInLocalStorage();
  }
  ...

  //Attention: localStorage = moyennement sécurisé / confidentiel
  private subscribeCaddyStoringInLocalStorage() {
    this.bsCompteur.subscribe( nextValueOfCompteur =>
      localStorage.setItem("caddySize", ""+nextValueOfCompteur ));

    this.bsCaddyContent.subscribe( caddyContent =>
      localStorage.setItem("caddyContent", JSON.stringify(caddyContent) ));
  }

  private tryReloadCaddyContentFromLocalStorage() {
    // code à améliorer (en tenant compte des exceptions):
    let caddySizeAsString = localStorage.getItem("caddySize");
    if(caddySizeAsString) {
      this._compteur = Number(caddySizeAsString);
      this.bsCompteur.next( this._compteur );
    }
    let caddyContentAsString = localStorage.getItem("caddyContent");
    if(caddyContentAsString) {
      this._caddyContent = JSON.parse(caddyContentAsString);
      this.bsCaddyContent.next( this._caddyContent );
    }
  }
}
```


2. Autres aspects divers

2.1. Formatage des valeurs à afficher avec des "pipes"

Exemples:

```
<div> {{ birthday | date:"MM/dd/yy" }} </div>
```

```
<!-- pipe with configuration argument => "February 25, 1970" -->
```

```
<div>Birthdate: {{currentHero?.birthdate | date:'longDate'}}</div>
```

```
<div>{{ title | uppercase }}</div>
```

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

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

Autres pipes prédéfinis:

currency

percent

....

"json pipe" pour (temporairement) déboguer un "binding":

```
<div>{{ currentHero | json }}</div>
```

```
<!-- Output:
  { "firstName": "Hercules", "lastName": "Son of Zeus",
    "birthdate": "1970-02-25T08:00:00.000Z",
    "url": "http://www.imdb.com/title/tt0065832/",
    "rate": 325, "id": 1 }
-->
```

Custom pipe:*app/exponential-strength.pipe.ts*

```

import {Pipe} from 'angular2/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 {
  transform(value:number, args:string[]) : any {
    return Math.pow(value, parseInt(args[0] || '1', 10));
  }
}

```

Composant utilisant le pipe personnalisé :

app/power-booster.component.ts

```

import {Component} from 'angular2/core';
import {ExponentialStrengthPipe} from './exponential-strength.pipe';

@Component({
  selector: 'power-booster',
  template: ` <h2>Power Booster</h2>
    <p>    Super power boost: {{2 | exponentialStrength: 10}}    </p> `,
  pipes: [ExponentialStrengthPipe]
})
export class PowerBooster { }

```

Power Booster

Super power boost: 1024

---->

2.2. Syntaxes alternatives :

bind-target = "expression" est équivalent à **[target]** = "expression"
on-target = "expression" est équivalent à **(target)**="expression"
bindon-target = "expression" est équivalent à **[(target)]**="expression"

```
<hero-detail *ngFor="#hero of heroes" [hero]="hero"></hero-detail>
```

est équivalent à

```
<template ngFor #hero [ngForOf]="heroes">
  <hero-detail [hero]="hero"></hero-detail>
</template>
```

2.3. Divers

Rappel :

The null or not hero's name is **{{nullHero?.firstName}}**

{{a?.b?.c?.d}} is ok if a or b or c is null or undefined

Possibilité d'externaliser le template dans un fichier html à part :

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

2.4. logs , ...

XI - Tests unitaires (et ...) avec angular

1. Tests autour de angular

1.1. Vue d'ensemble / différents types et technologies de tests

Rappel du contexte: une application "Angular2" correspond avant tout à la partie "Interface graphique" d'une architecture n-tiers et s'exécute au sein d'un navigateur web avec interprétation d'un code "typescript" transformé en "javascript".

Les principales fonctionnalités à tester sont les suivantes :

- **test unitaire d'un service** (avec éventuel "mock" sur accès aux données)
- **test unitaire d'un composant graphique** (avec éventuel mock sur "service")
- **test d'intégration complet (end to end)** englobant un dialogue HTTP/ajax/XHR avec des web services REST en arrière plan et sans avoir à connaître la structure interne de l'application (vue comme un boîte noire , vue de la même façon que depuis l'utilisateur final).

Pour tester du code javascript , la technologie de référence est "**jasmine**". Avec quelques extensions pour angularJs ou Angular2, cette technologie pourrait suffire à mettre en place des tests unitaires simples .

Etant donné que l'on souhaite également tester unitairement des composants graphiques (en javascript) qui s'exécutent dans un navigateur, on a également besoin d'une technologie de test qui puisse interagir avec un navigateur (chrome, firefox, ...) et c'est là qu'intervient "**karma**".

Pour effectuer des tests globaux en mode "**end-to-end**" / "**boîte noire**" , on peut utiliser la technologie spécifique "**protractor**" qui permet d'intégrer ensemble "**selenium**" et "**angular**" à travers des tests faciles à lancer.

On a souvent besoin d'automatiser certaines étapes lors du lancement des tests. Une technologie annexe de script telle que "**Grunt**" ou "**gulp**" peut alors être intéressante (nb: dans le cas particulier de "angular CLI" , beaucoup de choses sont déjà automatisées derrière le lancement de "ng test" et "grunt" ou "gulp" n'est pas indispensable).

Dans la plupart des cas, le coeur de la configuration du projet est basé sur npm / package.json (avec éventuellement "angular_cli") et la configuration autour des tests est globalement la suivante :

package.json (npm)

- **grunt** ou **gulp** ou **angular_cli** (scripts)
- **karma** ou **protractor** (interaction navigateur)
- **jasmine** (tests codés en javascript)
et extensions pour angular

A titre de comparaison, dans le monde "java" :

l'équivalent de "npm" + "grunt" ou "gulp" correspond à "maven" , "ant" ou "gradle"

l'équivalent de "karma" ou "protractor" correspond à "selenium_driver" ou autre

l'équivalent de "jasmine" correspond à "JUnit" (+ extensions "mockito", "...") .

1.2. Configuration des tests "karma" / contexte "angular CLI"

package.json

```
{
  "name": "my-app",
  "version": "0.0.0",
  "license": "MIT",
  "angular-cli": {},
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "test": "ng test",
    "pree2e": "webdriver-manager update --standalone false --gecko false",
    "e2e": "protractor"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "^2.3.1",
    "@angular/compiler": "^2.3.1",
    "@angular/core": "^2.3.1",
    "@angular/forms": "^2.3.1",
    "@angular/http": "^2.3.1",
    "@angular/platform-browser": "^2.3.1",
    "@angular/platform-browser-dynamic": "^2.3.1",
    "@angular/router": "^3.3.1",
    "angular-in-memory-web-api": "latest",
    "core-js": "^2.4.1",
    "rxjs": "^5.0.1",
    "ts-helpers": "^1.1.1",
    "zone.js": "^0.7.2"
  },
  "devDependencies": {
    "@angular/compiler-cli": "^2.3.1",
    "@types/jasmine": "2.5.38",
    "@types/node": "^6.0.42",
    "angular-cli": "1.0.0-beta.26",
    "codemlizer": "~2.0.0-beta.1",
    "jasmine-core": "2.5.2",
    "jasmine-spec-reporter": "2.5.0",
    "karma": "1.2.0",
    "karma-chrome-launcher": "^2.0.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
    "karma-remap-istanbul": "^0.2.1",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~4.0.13",
    "ts-node": "1.2.1",
    "tslint": "^4.3.0",
    "typescript": "~2.0.3"
  }
}
```

karma.conf.js

```
// Karma configuration file, see link for more information
// https://karma-runner.github.io/0.13/config/configuration-file.html

module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', 'angular-cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-remap-istanbul'),
      require('angular-cli/plugins/karma'),
      require('karma-jasmine-html-reporter')
    ],
    files: [
      { pattern: './src/test.ts', watched: false }
    ],
    preprocessors: {
      './src/test.ts': ['angular-cli']
    },
    mime: {
      'text/x-typescript': ['ts', 'tsx']
    },
    remapIstanbulReporter: {
      reports: {
        html: 'coverage',
        lcovonly: './coverage/coverage.lcov'
      }
    },
    angularCli: {
      config: './angular-cli.json',
      environment: 'dev'
    },
    reporters: config.angularCli && config.angularCli.codeCoverage
      ? ['progress', 'karma-remap-istanbul', 'kjhtml']
      : ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false
  });
};
```

NB: le rapporteur 'karma-jasmine-html-reporter' ('kjhtml') permet d'afficher le résultat des tests en mode HTML en cliquant sur l'icône **DEBUG** de la page HTML principale de karma .

src/test.ts

```
// This file is required by karma.conf.js and loads recursively all the .spec and
// framework files

import './polyfills.ts';

import 'zone.js/dist/long-stack-trace-zone';
import 'zone.js/dist/proxy.js';
import 'zone.js/dist/sync-test';
import 'zone.js/dist/jasmine-patch';
import 'zone.js/dist/async-test';
import 'zone.js/dist/fake-async-test';
import { getTestBed } from '@angular/core/testing';
import {
  BrowserDynamicTestingModule,
  platformBrowserDynamicTesting
} from '@angular/platform-browser-dynamic/testing';

// Unfortunately there's no typing for the `__karma__` variable.
// Just declare it as any.
declare var __karma__: any;
declare var require: any;

// Prevent Karma from running prematurely.
__karma__.loaded = function () {};

// First, initialize the Angular testing environment.
getTestBed().initTestEnvironment(
  BrowserDynamicTestingModule,
  platformBrowserDynamicTesting()
);
// Then we find all the tests.
const context = require.context('./', true, /\.spec\.ts$/);
// And load the modules.
context.keys().map(context);
// Finally, start Karma to run the tests.
__karma__.start();
```

1.3. Test unitaire élémentaire

basicTest.spec.ts

```
describe('1st tests', () => {
  it('1+1=2', () => expect(1+1).toBe(2));
  it('2+2=4', () => expect(2+2).toBe(4));
});
```

Ces spécifications de tests au format "jasmine" correspondent à un fichier d'extension ".spec.ts" (ou

".spec.js") et chaque partie " () => expect(...).toBe(...)" correspond à une assertion à vérifier.

1.4. Rare test unitaire isolé (sans extension Angular)

Exemple :

Service élémentaire à tester (calcul de tva):

compute.service.ts

```
import { Injectable } from '@angular/core';
@Injectable()
export class ComputeService {
  public vat(excl_tax : number, vat_pct : number ) : number{
    return excl_tax * vat_pct / 100;
  }
}
```

Test unitaire :

compute.service.spec.ts

```
import { ComputeService } from './compute.service';
// Isolated unit test = Straight Jasmine test
// no imports from Angular test libraries
describe('ComputeService without the TestBed', () => {
  let service: ComputeService;

  beforeEach(() => { service = new ComputeService(); });

  it('20%tva sur 200 ht = 40', () => {
    expect(service.vat(200,20)).toBe(40);
  });
});
```

1.5. Test unitaire de composant angular avec TestBed

Exemple :

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <nav>
      <a routerLink="/dashboard">Dashboard</a>
      <a routerLink="/heroes">Heroes</a>
    </nav>
    <router-outlet></router-outlet>
  `
})
```



```

    })
    export class AppComponent {
      title = 'Tour of Heroes';
    }

```

Exemple de test :

app.component.spec.ts

```

import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

import { RouterTestingModule } from '@angular/router/testing';
// for <router-outlet></router-outlet> in template
import { AppComponent } from './app.component';

describe('AppComponent (inline template)', () => {

  let comp: AppComponent;
  let fixture: ComponentFixture<AppComponent>;
  let de: DebugElement;
  let el: HTMLElement;

  //initialisations (avant chaque test)
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ RouterTestingModule ], // for <router-outlet> in template
      declarations: [ AppComponent ], // declare the test component
    });

    fixture = TestBed.createComponent(AppComponent);

    comp = fixture.componentInstance; // AppComponent test instance

    // query for the title <h1> by CSS element selector
    de = fixture.debugElement.query(By.css('h1'));
    el = de.nativeElement;
  });

  it('should have a defined component', () => {
    expect(comp).toBeDefined();
  });

  it('no title in the DOM until manually call `detectChanges`', () => {
    expect(el.textContent).toEqual('');
  });

  it('should display original title', () => {
    fixture.detectChanges();
    console.log("title:" + el.textContent);
    expect(el.textContent).toContain(comp.title);
  });

  it('should display a different test title', () => {
    comp.title = 'Test Title';
    fixture.detectChanges();
    expect(el.textContent).toContain('Test Title');
  });
});

```

```
});
```

La librairie `@angular/core/testing` comporte entre autres la classe fondamentale **TestBed** qui permet de **créer un module/environnement de test** (de type `@NgModule`) à partir de la méthode **configureTestingModule()** qui admet des paramètres d'initialisation très semblables à ceux de la décoration `@NgModule` que l'on trouve par exemple dans `app.module.ts`.

Il est conseillé d'appeler cette méthode à l'intérieur de `beforeEach()` de façon à ce que chaque test soit indépendant des autres (avec un contexte ré-initialisé).

Seulement après une bonne et définitive configuration, la méthode **TestBed.createComponent()** permet de créer une chose technique de type **ComponentFixture<ComponentType>** sur laquelle on peut invoquer :

.componentInstance de façon à accéder à l'instance du composant à tester

.debugElement() de façon à accéder au nœud d'un arbre DOM lié au template du composant à tester.

NB : la méthode **fixture.detectChanges()** permet d'explicitement (re)synchroniser la vue HTML en fonction des changements effectués au niveau du modèle.

Il est éventuellement possible d'importer le service automatique

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

et de le déclarer à l'initialisation de TestBed via ce code :

```
TestBed.configureTestingModule({
  declarations: [ AppComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```

cependant cette invocation automatique et implicite de `detectChanges()` étant asynchrone, il y a beaucoup de cas où l'appel explicite sera nécessaire pour immédiatement tenir compte d'un changement au niveau des lignes de code séquentielles de l'écriture d'un test.

Attention : sur un vrai projet, il est déconseillé de tester le composant principal "`app.component`" car celui-ci comporte trop de dépendances vers les autres composants et services.

Un test unitaire devrait idéalement ne porter que sur un sous composant bien précis de l'application.

1.6. Ajustements à effectuer pour un template html externe

Dans le cas où le template html (ou les styles css) n'est (ou ne sont pas) pas en inline mais dans un fichier ".html" ou ".css" annexe, le code du test doit être ajusté de façon à ce que certains mécanismes asynchrones d'angular aient le temps de lire, compiler et intégrer les fichiers externes.

Ligne supplémentaire (dans le haut de app.component.spec.ts):

```
import { async } from '@angular/core/testing'; //si templateUrl externe
```

Deux beforeEach() au lieu d'un seul (dans app.component.spec.ts):

```
//first beforeEach with async() for template and css initialization :
beforeEach(async() => {
  TestBed.configureTestingModule({
    imports: [ RouterTestingModule ], // if necessary
    declarations: [ AppComponent ], // declare the test component
  })
  .compileComponents(); // compile asynchronously template and css , return promise
});

//second beforeEach without async() for fixture, component, debugElement:
beforeEach(() => {

  fixture = TestBed.createComponent(AppComponent);

  comp = fixture.componentInstance; // AppComponent test instance

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));
  el = de.nativeElement;
});
```

Explications autour de **async** :

Le framework jasmine lance automatiquement en boucle les fonctions *beforeEach()* et *it()* ;

le code interne des méthodes *beforeEach()* et/ou *it()* est :

- soit entièrement synchrone et aucune attente n'est à prévoir
- soit en partie asynchrone et certaines attentes sont à paramétrer

Le framework "**jasmine**" offre de façon standard une fonction **done()** permettant de l'avertir de la fin d'une fonction appelée *beforeEach()* ou *it()* :

```
beforeEach_or_it(... , done => {
  appel_synchrone1() ;
  appel_synchrone2() ;
  appel_asynchrone_retournant_promise().then( () => {
    instructions_de_la_callback_asynchrone ;
  });
  done();
});
```

```

        done();
    });
});

```

La fonction utilitaire **async()** de [@angular/core/testing](#) est une alternative simplifiée permettant de ne pas avoir à appeler explicitement *done()* du standard jasmine. Le *then / done* est caché à l'intérieur.

NB: dans le cas particulier de "Angular_CLI" et *webPack*, l'équivalent de *.compileComponents()* semble être automatiquement effectué lors du packaging et il semble être possible de tester un composant angular comportant un template externe sans avoir à appeler *.compileComponents()* au sein d'un *beforeEach(async(...))*.

1.7. Tester un composant angular utilisant un service simple

Exemple de composant à tester (avec service simple injecté) :

vat.component.ts

```

import { Component } from '@angular/core';
import { ComputeService } from '../compute.service';

@Component({
  selector: 'temp-test',
  template: `
    <h3>temp test / value added tax</h3>
    price excluded of tax: <input type='text'
                          id='price_excluded_of_tax_input'
                          [(ngModel)]='price_excluded_of_tax'
                          (input)='onRefresh($event)' /> <br/>
    value added tax rate (%): <input type='text'
                                  id='v_a_tax_rate_pct_input'
                                  [(ngModel)]='v_a_tax_rate_pct'
                                  (input)='onRefresh($event)' /> <br/>
    price inclusive of tax :
    <span id='price_inclusive_of_tax_span'>
      {{price_inclusive_of_tax}} </span><br/>
  `
})
export class VatComponent {
  price_excluded_of_tax: number;
  v_a_tax_rate_pct: number;
  price_inclusive_of_tax: number;

  constructor(private computeService: ComputeService) { }
}

```

```
onRefresh = function (evt : KeyboardEvent){
    this.price_inclusive_of_tax = Number(this.price_excluded_of_tax)
    + Number(this.computeService.vat(this.price_excluded_of_tax,
    this.v_a_tax_rate_pct));
}
}
```

temp test / value added tax

price excluded of tax:

value added tax rate (%):

price inclusive of tax : 240

Exemple de test :

vat.component.spec.ts

```
import { ComponentFixture, TestBed, async } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

import { VatComponent } from './vat.component';
import { ComputeService } from './compute.service';
import { FormsModule } from '@angular/forms';

describe('VatComponent (using simple service)', () => {

    let comp: VatComponent;
    let fixture: ComponentFixture<VatComponent>;
    let deEot, deIot, deRate: DebugElement;
    let elIot: HTMLElement;
    let elEot, elRate: HTMLInputElement;

    let computeServiceWithinTest : ComputeService;

    beforeEach(async(() => {
        //stub Service for test purposes (will be cloned and injected)
        let computeServiceStub = {
            vat(excl_tax : number, vat_pct : number ) : number{
                return excl_tax * vat_pct / 100;
            }
        };

        TestBed.configureTestingModule({
            imports: [ FormsModule ], // FormsModule is for [(ngModel)]
            providers: [ {provide: ComputeService,
                useValue: computeServiceStub } ],
            declarations: [ VatComponent ],
        }).compileComponents();
```

```

    }));

    beforeEach( () => {
        fixture = TestBed.createComponent(VatComponent);
        computeServiceWithinTest =
            fixture.debugElement.injector.get(ComputeService);

        comp = fixture.componentInstance; // VatComponent test instance

        deEot = fixture.debugElement.query(
            By.css('#price_excluded_of_tax_input'));
        elEot = deEot.nativeElement;
        deRate = fixture.debugElement.query(
            By.css('#v_a_tax_rate_pct_input'));
        elRate = deRate.nativeElement;
        deIot = fixture.debugElement.query(
            By.css('#price_inclusive_of_tax_span'));
        elIot = deIot.nativeElement;
    });

    it('20% , 200 -> 240 from model', () => {
        comp.price_excluded_of_tax=200;
        comp.v_a_tax_rate_pct=20; //20%
        comp.onRefresh(null /*not used event*/);
        fixture.detectChanges();
        console.log("from model, price_inclusive_of_vat:"
            +elIot.textContent);
        expect(elIot.textContent).toContain('240');
    });

    it('test computeServiceWithinTest', () => {
        expect(computeServiceWithinTest).toBeDefined();
        expect(computeServiceWithinTest.vat(100, 20)).toBe(20);
    });
});

```

Remarque importante :

Bien que dans cet exemple extra-simple, on aurait pu utiliser en direct le réel service de calcul via `TestBed.configureTestingModule({`

```

    providers:    [ ComputeService ] ,
...} ;)
```

il est en général conseillé de demander à injecter un service de type "stub" ou "mock" (simulant le comportement du réel service et permettant de se focaliser sur le composant angular à tester) :

```

//stub Service for test purposes (will be cloned and injected)
let    computeServiceStub = {
        vat(excl_tax : number, vat_pct : number ) : number{
            return excl_tax * vat_pct / 100;
        }
    };

```

```
TestBed.configureTestingModule({
  imports: [ FormsModule ], // FormsModule is for [(ngModel)]
  providers: [ {provide: ComputeService,
                useValue: computeServiceStub } ],
  declarations: [ VatComponent ],
}).compileComponents();
```

1.8. Tester un composant angular utilisant un service asynchrone

Exemple de composant à tester (utilisant un service asynchrone) :

identification.component.ts

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { ClientAuth } from './client';
import { ClientService } from './client.service';

@Component({
  template: `
    <div> <h3> {{title}} </h3>
      numClient:<input id="numClientInput" type="text" [(ngModel)]="numClient"/> <br/>
      password:<input id="passwordInput" type="text" [(ngModel)]="password"/><br/>
      <button (click)="onVerifPassword()" > verif. password </button> <br/>
      <button (click)="onNavigate()" [hidden]="!resVerifPwd" > espace client identifie </button>
    </div> ` })
export class IdentificationComponent {
  title : string = "identification client minibank";
  numClient : number;
  password : string;
  resVerifPwd : boolean = false;

  constructor(private _router: Router , private _clientService : ClientService){ }

  onVerifPassword() : void {
    let clientAuth : ClientAuth = {
      "numClient": this.numClient,
      "password": this.password,
      "ok" : null};

    this._clientService.verifyClientAuthObservableWithAlternativeTry(clientAuth)
      .subscribe(verifiedClientAuth =>{
        if(verifiedClientAuth.ok) { this.resVerifPwd=true; console.log("verifyAuth ok") }
        else { this.resVerifPwd=false; console.log("verifyAuth failed") } } ,
        error => console.log(error));
  }
  ... }
```

identification client minibank

numClient:

password:

Mock du service asynchrone (*spyOn(...).and(...)*) :

En règle générale un service asynchrone effectue un dialogue HTTP/XHR avec un web service REST distant (fonctionnant sur un autre ordinateur en Php , java , nodeJs ou autre).

Cette dépendance externe n'étant pas facile à gérer durant les tests unitaires, la stratégie préconisée dans la plupart des cas consiste à préparer/initialiser le test en :

- **injectant le réel service dans le composant à tester** (paramétrage *providers* : [*ServiceXy*] de TestBed)
- **redéfinissant ponctuellement/localement la fonction du service qui sera appelée** (via *spyOn(...).and.returnValue()* ou bien *spyOn(...).and.callFake(function(...) { })*).

Exemple :

identification.component.spec.ts

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { async, tick, fakeAsync } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

import { FormsModule } from '@angular/forms';
import { RouterTestingModule } from '@angular/router/testing';
import { HttpModule } from '@angular/http';
import './rxjs-extensions';
import { Observable } from 'rxjs/Observable'; // _http.get() return Observable<Response> !!!

import { IdentificationComponent } from './identification.component';
import { AppConfig, MY_APP_CONFIG_TOKEN, MY_DEV_APP_CONFIG } from './app.config';

import { ClientAuth } from './client';
import { ClientService } from './client.service';
import Spy = jasmine.Spy;

describe('IdentificationComponent (using async compte service)', () => {
  let comp: IdentificationComponent;
  let fixture: ComponentFixture<IdentificationComponent>;
  let deNumClient, dePassword: DebugElement;
  let elNumClient, elPassword: HTMLInputElement;

  let clientServiceWithinTest : ClientService;

  let stubClientAuth : ClientAuth;
  let spy : Spy; //let spy : any;
```



```
beforeEach(async() => {
  TestBed.configureTestingModule({
    imports: [ RouterTestingModule , FormsModule , HttpModule ],
    providers:  [ ClientService , { provide: MY_APP_CONFIG_TOKEN,
                                   useValue: MY_DEV_APP_CONFIG }],
    declarations: [ IdentificationComponent ],
  }).compileComponents();
});
```

```
beforeEach( () => {
  fixture = TestBed.createComponent(IdentificationComponent);
```

```
  clientServiceWithinTest = fixture.debugElement.injector.get(ClientService);
```

```
// Setup spy on the `verifyClientAuthObservableWithAlternativeTry` method :
```

```
/*//v1
  subClientAuth = { "numClient": 1, "password": 'pwd1', "ok" : true};
  spy = spyOn(clientServiceWithinTest, 'verifyClientAuthObservableWithAlternativeTry')
    .and.returnValue(Observable.of(subClientAuth));
*/
spy = spyOn(clientServiceWithinTest, 'verifyClientAuthObservableWithAlternativeTry')
  .and.callFake(function(clientAuth){
    if(clientAuth.password == "pwd" + clientAuth.numClient) {
      clientAuth.ok = true;    }
    else { clientAuth.ok = false;  }
    return Observable.of(clientAuth);
  });
```

```
comp = fixture.componentInstance; // VatComponent test instance
```

```
deNumClient = fixture.debugElement.query(By.css('#numClientInput'));
elNumClient = deNumClient.nativeElement;
dePassword = fixture.debugElement.query(By.css('#passwordInput'));
elPassword = dePassword.nativeElement;
});
```

```
it('test bonne identification', async() => {
  comp.numClient=1;    comp.password='pwd1'; // good password
  comp.onVerifPassword();
  // wait for async activities (observable/promise/event/...)
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(spy.calls.any()).toBe(true, 'verifyClientAuth.... should be called');
    console.log("1/pwd1 - comp.resVerifPwd:"+comp.resVerifPwd);
    expect(comp.resVerifPwd).toBe(true);
  });
});
```

```
it('test mauvaise identification',fakeAsync( () => {
```

```

comp.numClient=1;    comp.password='pwdXy'; //wrong password
comp.onVerifPassword();
tick() ; //waiting inside fakeAsync
fixture.detectChanges();
expect(spy.calls.any()).toBe(true, 'verifyClientAuth.... should be called');;
console.log("1/pwdXy - comp.resVerifPwd:"+comp.resVerifPwd);
expect(comp.resVerifPwd).toBe(false);
  });
});

```

NB: En théorie ,

```

async ( () => {
    appel_synchrone_1() ;
    appel_synchrone_2() ;
    appel_asynchrone_Xy_retournant_promise_ou_observable() ;
    fixture.whenStable().then() => {
        //zone de test (attente automatique de toute promise ou observable
        //indirectement déclenchée)
        fixture.detectChanges();
        expect(...).toBe(...);
    } ;
})

```

et

```

fakeAsync( () => {
    appel_synchrone_1() ;
    appel_synchrone_2() ;
    appel_asynchrone_Xy_retournant_promise_ou_observable() ;
    tick() ; // attente au sein de fakeAsync()
    fixture.detectChanges();
    expect(...).toBe(...);
})

```

sont censés être **à peu près équivalent** et **permettre d'attendre la fin d'une opération asynchrone** indirectement déclenchée par un composant angular et un service.

En pratique, certains bugs, limitations techniques et autres problèmes font qu'un test d'appel asynchrone est assez délicat à mettre au point.

1.9. Test "end-to-end / e2e"

Les tests e2e de angular sont exécutés via la technologie "**protractor**".
 Cette technologie (spécifique à angular) est en interne basée sur "**selenium**" et "jasmine".

e2e/app.po.ts

```
import { browser, element, by } from 'protractor';

export class ENgPage {
  navigateTo() {
    return browser.get('/');
  }

  getParagraphText() {
    //return element(by.css('app-root h1')).getText();
    return element(by.css('app-root #mainHeader h3')).getText();
  }
}
```

La méthode `getParagraphText()` de l'exemple ci-dessus permet d'accéder à l'élément suivant du template html du composant principal de l'application :

```
...
@Component({
  selector: 'app-root',
  template:`
    <header id="mainHeader" role="banner">
      <h3>Minibank App</h3>
    </header>
  `
})
...`
```

e2e/app-e2e-spec.ts

```
import { ENgPage } from './app.po';

describe('e-ng App', function() {
  let page: ENgPage;

  beforeEach(() => {
    page = new ENgPage();
  });

  it('should display title Minibank App', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('Minibank App');
  });
});
```

Lancement d'un test e2e (angular 2) :

- 0) vérifier la connexion réseau (pour téléchargement automatique de certains pilotes)
- 1) se placer dans le répertoire my-app (ou ...) contenant package.json
- 2) lancer d'abord l'application via **ng serve** (ou un équivalent)
- 3) lancer (éventuellement dans une autre console) **ng e2e**

```
> protractor "./protractor.conf.js"
[19:28:53] I/direct - Using ChromeDriver directly...
[19:28:53] I/launcher - Running 1 instances of WebDriver
Spec started
e-ng App
✓ should display title Minibank App
Executed 1 of 1 spec SUCCESS in 4 secs.
[19:29:12] I/launcher - 0 instance(s) of WebDriver still running
[19:29:12] I/launcher - chrome #01 passed
All end-to-end tests pass
```

Autres fonctionnalités de protractor :

...

XII - Packaging et déploiement d'appli. angular

1. Vue d'ensemble sur technologies "javascript"

1.1. Version de javascript (ES5 ou ES6/es2015)

Les versions standardisées/normalisées de javascript sont appelées **ES** (EcmaScript) .

Les versions modernes sont :

- **ES5** (de 2009) – supporté par quasiment tous les navigateurs actuels ("mobiles" ou "desktop")
- **ES6** (renommé **ES2015** car normalisé en 2015) . ES6/es2015 n'est pour l'instant supporté que par quelques navigateurs "desktop" récents.
ES6/ES2015 apporte quelques nouvelles syntaxes et mots clefs (**class** , **let**, ...) et gère des modules dits "statics" via "**import** { ComponentName } **from** 'moduleName' ;" et **export** .

En 2016, 2017, 2018, ... , une application "angular" doit être compilée/transpilée en ES5 de façon à pouvoir s'exécuter sur n'importe quel navigateur.

1.2. Types de modules (cjs , amd , es2015 , umd , ...)

Beaucoup de technologies javascript modernes s'exécutent dans un environnement prenant en charge des modules (bien délimités) de code (avec import/export) . Le développement d'une application "Angular(2)" s'effectue à fond dans ce contexte.

Les principales technologies de "modules javascript" sont les suivantes :

- **CommonJS (cjs)** – modules "synchrone", **syntaxe** "var xyz = **requires**('xyz')"
NB : node (nodeJs) utilise partiellement les idées et syntaxes de CommonJS .
- **AMD** (Asynchronous Module Definition) avec chargements asynchrones
- **ES2015 Modules** : syntaxiquement standardisé , mots clef "import {...} from '...'" et export pour la gestion statique des modules et System.import("...") possible pour liaisons dynamiques.
- **SystemJS** (très récent et pas encore complètement stabilisé) supporte en théorie les 3 technologies de modules précédentes (cjs , amd, es2015) et c'est pour cette raison que le tutorial "Angular 2 / Tour of Heroes" s'appuie dessus pour charger le code en mémoire au niveau du navigateur et de la page d'accueil index.html.
SystemJS est à priori capable de gérer une compilation/transpilation "typescript → javascript" au dernier moment (juste à temps) mais l'opération est lente et donc rarement retenue.

Il existe aussi les formats de modules suivants :

- **umd** (universal module definition) – fichiers *xyz.umd.js*
- **iife** (immediately-invoked function expression) – fonctions anonymes auto-exécutées

1.3. Technologies de "packaging" (webpack, rollup, ...) et autres

De façon à éviter le téléchargement d'une multitude de petits fichiers, il est possible de créer des gros paquets appelés "**bundles**".

Les principales technologies de packaging "javascript" sont les suivantes :

- **webpack** (mature et supportant les modules "**csj**", "**amd**", ...)
- **rollup** (récent et pour modules "**es2015**") . Rollup est une fusion intelligente de n fichiers en 1 (remplacement des imports/exports par sous contenu ajustés, prise en compte de la chaîne des dépendances en partant par exemple de *main.js*)
- **SystemJs-builder** (technologie très récente par encore mature)

Autres technologies annexes (proches) :

- **browserify** : technologie déjà assez ancienne permettant de faire fonctionner un module "*nodeJs*" dans un navigateur après transformation.
- **babel** : transformation (par exemple *es2015* vers *es5*)
- **uglify** : minification (enlever tous les espaces et commentaires inutiles, simplifier noms des variables, ...) → code beaucoup plus compact (*xyz.min.js*) .
- **gzip** : compression .Les fichiers *bundlexy.min.js.gz* sont automatiquement traités par quasiment tous les serveurs HTTP et les navigateurs : décompression automatique après transfert réseau).

Spécifiquement pour Angular2 :

- **ngc** (Angular compiler)
→ fichiers "**AoT**" (Ahead-of-Time compilation)
optimisation des templates HTML
- **material2** : librairie de composants graphiques évolués (au format Angular2 / *.ts*)

2. Précisions sur contexte développement "Angular"

Le développement d'une application "Angular(2)" est basée sur *nodeJs* . La plupart des bundles d'angular (*@angular/core*, ...) sont au format "*umd.js*" et en *es5*. Il existe également une distribution parallèle au format "*es2015*" des librairies d'angular et *RxJs* dans le répertoire *node-modules* récupéré par *npm*..

Les fichiers typescript (".ts") de l'application peuvent être compilés/transpilés dans les formats "es5" ou bien "es2015" selon les paramètres du fichier tsconfig.json . Les modules générés par tsc peuvent être au format csj ou es2015 ou autre.

3. Packaging et déploiement avec ou sans bundle(s)

3.1. Déploiement (rare) sans bundle

Il est possible d'**extraire (par copie) les librairies fondamentales de angular2 et de RxJs** de la branche **node_modules** (de nodeJs) dans un répertoire "**dist/lib**".

Ce répertoire "**dist**" comportera également :

- les templates html et les styles css de l'application.
- le résultat des transpilations (".ts → .js") des composants de notre application.

Au final , ce répertoire "**dist**" comportera tous les fichiers nécessaires de l'application "angular2" et son contenu pourra être recopié "tel quel" vers n'importe quel serveur HTTP (ex : httdocs de apache2) . On pourra également recopier cet ensemble de fichiers "angular2" pour les intégrer en tant que sous partie "**html+js+css**" d'une application Java/JEE , Php ou autre .

Bien que possible, un déploiement sans bundle va induire un très grand nombre de micro-téléchargements lorsqu'un utilisateur va utiliser l'application depuis son navigateur.

Ceci dit , identifier et extraire les librairies nécessaires de "angular2 + RxJs" c'est un premier pas vers la constitution d'un futur bundle plus optimisé.

Voici la liste des librairies à récupérer (par extraction/copie) :

'reflect-metadata/Reflect.js'

'zone.js/dist/zone.js'

'core-js/client/shim.min.js',
 'systemjs/dist/system-polyfills.js',
 'systemjs/dist/system.src.js',
 'rxjs/**',
 '@angular/core/bundles/core.umd.js',
 '@angular/common/bundles/common.umd.js',
 '@angular/compiler/bundles/compiler.umd.js',
 '@angular/platform-browser/bundles/platform-browser.umd.js',
 '@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',
 '@angular/http/bundles/http.umd.js',
 '@angular/router/bundles/router.umd.js',
 '@angular/forms/bundles/forms.umd.js',
 '@angular/upgrade/bundles/upgrade.umd.js',
 'angular-in-memory-web-api/bundles/in-memory-web-api.umd.js'

3.2. Déploiement via webpack et angular-cli

Le projet "**Angular-CLI**" utilise maintenant en interne la technologie "**webpack**" pour générer les bundles à déployer .

On peut ainsi en **quelques lignes de commandes** :

- **créer un nouveau projet** ayant la *structure et la configuration attendues par webpack et angular-cli*.
- créer des débuts de composants (Component , Service , ...)
- *générer des bundles* en mode "**dev**" (développement)
ou bien en mode "**prod**" (production) : avec uglify / compression en plus.

Avantages et inconvénients de cette approche :

- + pleins de choses sont automatisées
- + c'est à court terme la solution la plus simple/efficace
- + depuis début 2017, Angular-cli est préconisé sur le site officiel de Angular(2)
- + le projet "material2" s'appuie sur Angular-cli
- structure du projet imposée (*plus gênant si ça devient la méthode préconisée/standardisée*).

URL du projet Angular-CLI :

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

3.3. Déploiement via rollup et npm , grunt ou gulp

c'est techniquement possible (voir détails dans ancienne annexe) mais à court terme cette solution n'est pas encore complètement mature.

Les scripts d'automatisation sont assez difficiles à écrire et maintenir .

3.4. Déploiement via SystemJs-builder ou autres

c'est également possible mais à court terme cette solution n'est pas encore complètement mature.

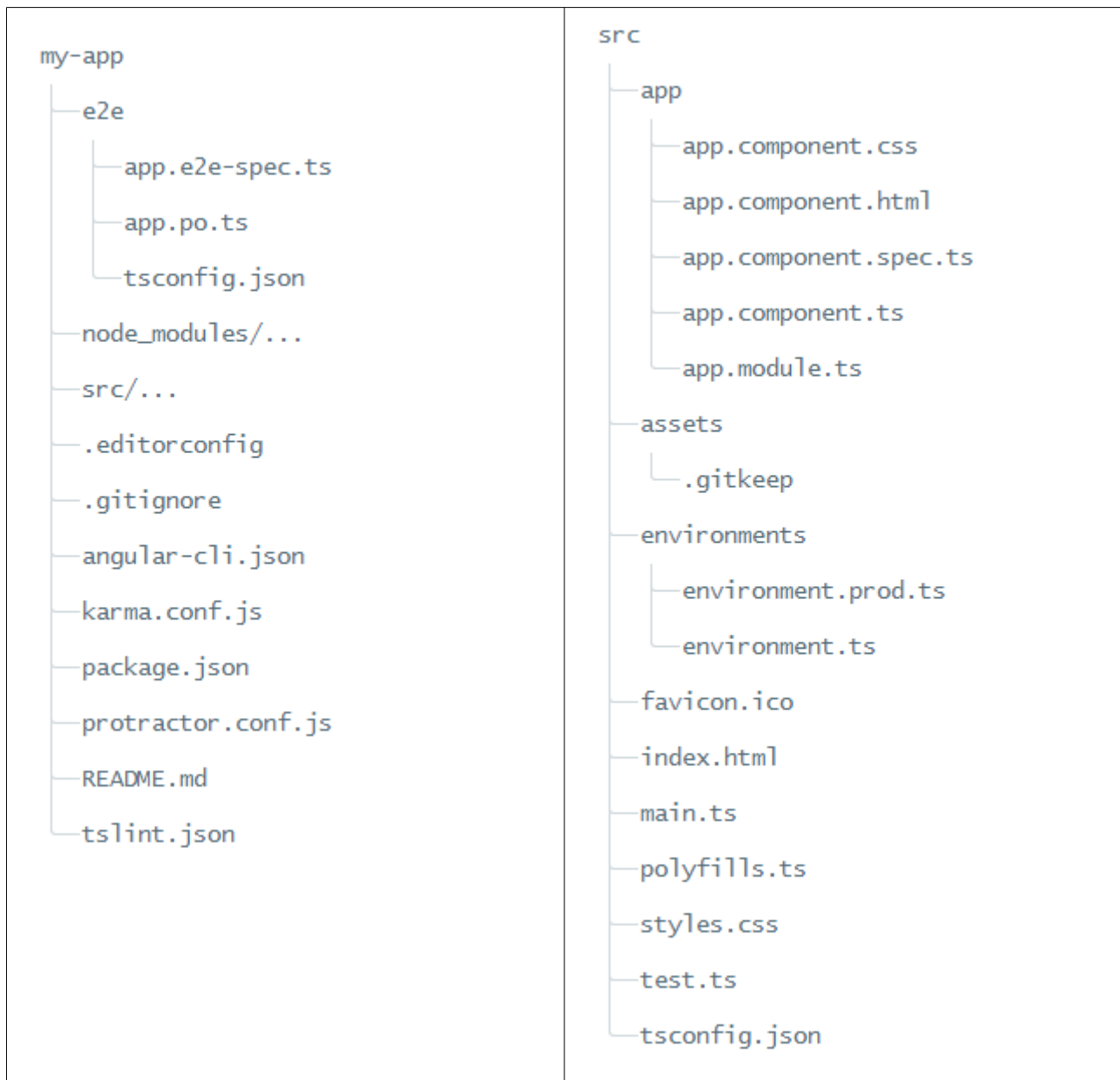
4. Angular-CLI

Angular-CLI est maintenant officiellement préconisé sur le site officiel de Angular2. Autant faire comme tout le monde et utiliser cette façon de structurer et construire une application angular.

Installation (en mode global) de angular-cli via npm : **npm install -g @angular/cli**

NB : si besoin , upgrade préalable de npm via `npm install npm@latest -g` ou bien carrément désinstaller nodeJs et réinstaller une version plus récente.

La création d'une nouvelle application s'effectue via la ligne de commande "**ng new my-app**". Cette commande met pas mal de temps à s'exécuter (beaucoup de fichiers sont téléchargés). L'arborescence des répertoires et fichiers créés est la suivante :



* **src/assets** est prévu pour contenir des ressources annexes (images ,) qui seront automatiquement recopiées/packagées avec l'application construite.

* **e2e** correspond à "**end to end tests**".

NB : il faut ajouter `<link rel="stylesheet" href="styles.css">` dans **index.html**

Selon les spécificités du projet, le fichier **package.json** devra éventuellement être ajusté (ex : ajout de `"angular-in-memory-web-api": "latest"` pour quickstart / Tour of Heroes).

Principales lignes de commandes de **ng** (angular-cli) :

ng new <i>my-app</i> , <i>cd my-app</i>	Création d'une nouvelle application " <i>my-app</i> ".
ng serve	Lancement de l'application en mode développement (watch & compile file , launch server,) → URL par défaut : <i>http://localhost:4200</i>
ng build ng build --prod (ou <code>--target=production</code> <code>--environment=prod</code>)	Construction de l'application (par défaut en mode --dev)
ng help	Affiche les commandes et options possibles
ng generate ... (ou ng g ...)	Génère un début de code pour un composant , un service ou autre (selon argument précisé)
ng test	Lance les tests unitaires (via karma)
ng e2e	Lance les tests "end to end" / "intégration" (après un ng server à lancer au préalable)
...	...

Type de composants que l'on peut générer (début de code) :

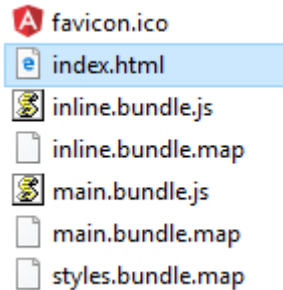
Scaffold (échafaudage)	Usage (ligne de commande)
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-module</code>

NB : le nom du composant à générer est interprété comme un chemin relatif (*my-new-component* ou *../my-new-component* ou ...)

NB : **ng serve** construit l'application entièrement en mémoire pour des raisons d'efficacité / performance (on ne voit aucun fichier temporaire écrit sur le disque) .

ng build génère quant à lui des fichiers dans le répertoire **my-app/dist** .

Contenu du répertoire **my-app/dist** après la commande "**ng build**" (par défaut en mode **--dev**) :



ng build --prod est quelquefois accompagné de quelques "bugs" avec certaines versions de angular-cli (encore en version bêta) .

Lorsque le mode "**--prod**" fonctionne , les fichiers "bundle" générés sont compressés au format ".gz".

ng build --watch existe (au cas où) mais c'est généralement ng serve qui déclenche automatiquement l'option **--watch** (pour recompiler automatiquement dès qu'un fichier a changé)

XIII - Sécurité – application "Angular2"

1. Sécurisation d'une application "angular2"

1.1.

ANNEXES

XIV - Annexe – Détails Typescript et javascript

1. Détails du langage typescript (ts)

1.1. Modules internes et externes

Le langage typescript gère deux sortes de modules "internes/logiques" et "externes" :

internes/logiques	Via mot clef module <i>ModuleName</i> { ... } englobant plusieurs export (sémantique de "namespace" et utilisation via préfixe " ModuleName ".)	Dans un seul fichier ou réparti dans plusieurs fichiers (à regrouper) , peu importe.
externes	Via (au moins un) mot clef export au premier niveau d'un fichier et utilisation via mot clef import (associé à requires ('./moduleName')) .	Toujours un fichier par module externe (nom du module = nom du fichier) Selon contexte (nodeJs ou ...)

1.2. Modules internes (sémantique de "namespace")

Module (avec utilisation locale dans même fichier):

```

module Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }

  var lettersRegex = /^[A-Za-z]+$/;    // volontairement non exporté (détail interne)
  var numberRegex = /^[0-9]+$/;      // volontairement non exporté (détail interne)

  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
      return lettersRegex.test(s);
    }
  }

  export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
      return s.length === 5 && numberRegex.test(s);
    }
  }
}

// échantillon de valeurs :
var strings = ['Hello', '98052', '101'];
// tableau de validateurs

```

```

var validators: { [s: string]: Validation.StringValidator; } = {};
validators['ZIP code'] = new Validation.ZipCodeValidator();
validators['Letters only'] = new Validation.LettersOnlyValidator();
// Validation de chaque échantillon par chaque valideur :
strings.forEach(s => {
  for (var name in validators) {
    console.log("'" + s + "' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ' )
                    + name);
  }
});

```

```

"Hello"   does not match ZIP code
"Hello"   matches Letters only
"98052"   matches ZIP code
"98052"   does not match Letters only
"101"     does not match ZIP code
"101"     does not match Letters only

```

1.3. Modules externes et organisation en fichiers

Exemple :

validateurs.ts

```

export interface StringValidator {
  isAcceptable(s: string): boolean;
}

var lettersRegexp = /^[A-Za-z]+$/;
var numberRegexp = /^[0-9]+$/;

export class LettersOnlyValidator implements StringValidator {
  isAcceptable(s: string) {
    return lettersRegexp.test(s);
  }
}

export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}

```

Soit l'alias "tsc:m": "**tsc --module *commonjs***" défini dans package.json (sachant que *commonjs* correspond à la technologie des modules de nodeJs)

```
npm run tsc:m validateurs.ts test-validateurs.ts
```


test-validateurs.ts

```
import validateurs = require('./validateurs');

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: validateurs.StringValidator; } = {};
validators['ZIP code'] = new validateurs.ZipCodeValidator();
validators['Letters only'] = new validateurs.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log("'" + s + "' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ') +
name);
    }
});
```

1.4. Aspects divers de TypeScript

Fonctions à retour variable ou bien surchargées (overload) :

La prise en charge des fonctions surchargées (avec différents types de paramètres d'entrées) est assez limitée en "typescript" .

Exemple (seulement intéressant pour la syntaxe) :

```
function displayColor( tabRgb : number[] ) : void;
function displayColor(c: string) : void;

function displayColor(p:any) : void{
    if(typeof p == "string" ){
        console.log("c:" + p);
    }
    else if(typeof p == "object" ){
        console.log("r:" + p[0] + ",g:" + p[1] + ",b=" + p[1]);
    }
    else{
        console.log("unknown color , typeof =" + (typeof p));
    }
}

displayColor([125,250,30]);
displayColor("red");
```

1.5. Ambient ts

On appelle "**ambient ts**" une déclaration de choses "js" externes potentiellement appelables (ex : jQuery) .

Ceci permet d'appeler des fonctions "jQuery" (ou autres) depuis le code typescript de notre application tout en ayant la possibilité d'utiliser un typage fort.

Intérêts potentiels : debug plus aisé , auto-complétion, ...

Les parties

"lib": ["es6", "es2015", "dom"],

et

"typeRoots": [
 "./node_modules/@types/"
]

de **tsconfig.json** vont dans ce sens et il vaut mieux éviter les doublons de déclaration.

Pour infos , @types est plus récent que "definitivedTyped" (quasi obsolète aujourd'hui) et les librairies "es6" ou "es2015" sont souvent plus précises que @types/...

XV - Annexe – Web Services REST (coté serveur)

1. Généralités sur Web-Services REST

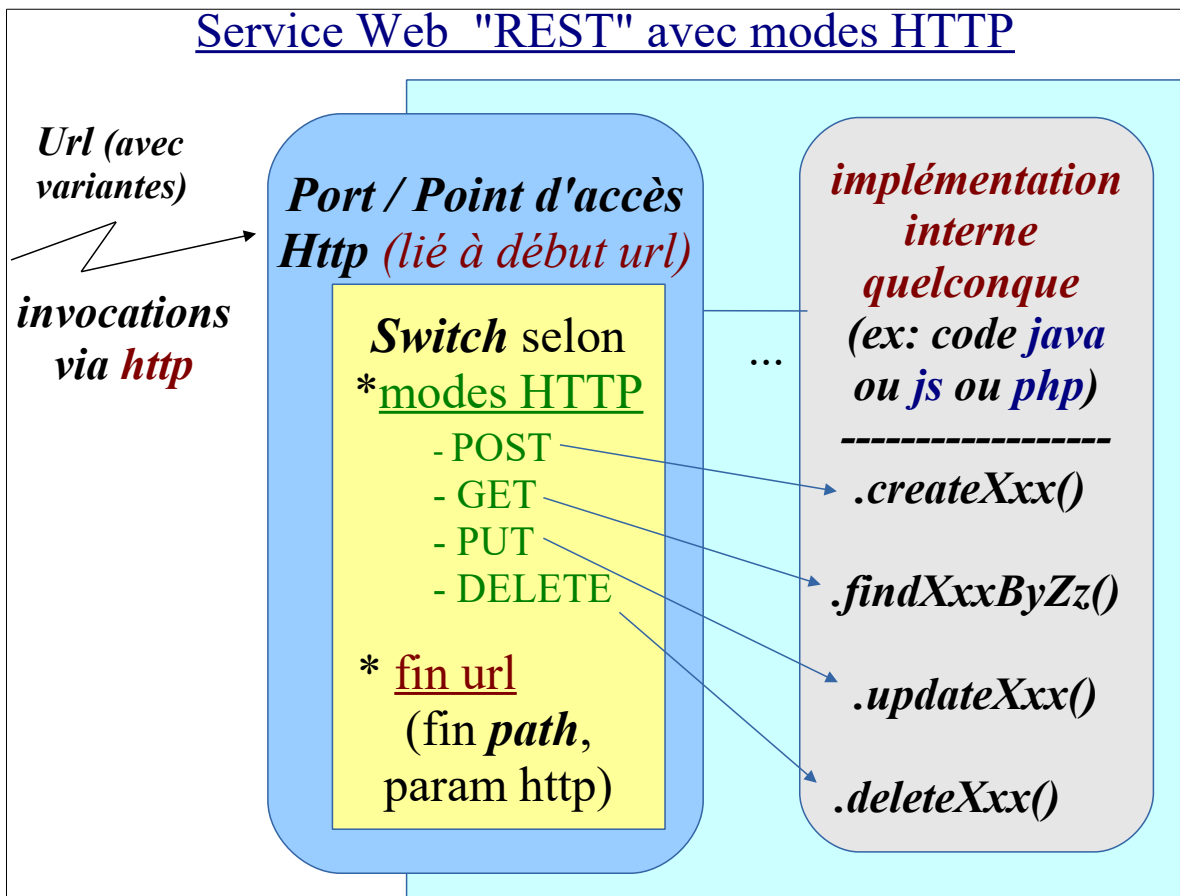
2 grands types de services WEB: SOAP/XML et REST/HTTP

WS-* (SOAP / XML)

- "Payload" systématiquement en **XML** (*sauf pièces attachées / HTTP*)
- **Enveloppe SOAP** en XML (*header facultatif pour extensions*)
- **Protocole de transport au choix (HTTP, JMS, ...)**
- Sémantique quelconque (*appels méthodes*) , **description WSDL**
- **Plutôt** orienté Middleware SOA (*arrière plan*)

REST (HTTP)

- "Payload" au choix (XML , HTML , **JSON**, ...)
- Pas d'enveloppe imposée
- **Protocole de transport = toujours HTTP.**
- Sémantique "CRUD" (*modes http PUT,GET,POST,DELETE*)
- **Plutôt** orienté IHM Web/Web2 (*avant plan*)



Points clefs des Web services "REST"

Retournant des données dans un format quelconque ("**XML**", "**JSON**" et éventuellement "**txt**" ou "**html**") les web-services "REST" offrent des **résultats qui nécessitent généralement peu de re-traitements** pour être mis en forme au sein d'une IHM web.

Le format "**au cas par cas**" des données retournées par les services REST permet peu d'automatisme(s) sur les niveaux intermédiaires.

Souvent associés au format "**JSON**" les web-services "REST" **conviennent parfaitement** à des appels (ou implémentations) au sein du **langage javascript** .

La **relative simplicité des URLs d'invocation des services "REST"** permet des **appels plus immédiats** (un simple *href*="**...**" suffit en mode **GET** pour les recherches de données) .

La **compacité/simplicité des messages "JSON"** souvent **associés à "REST"** permet d'obtenir **d'assez bonnes performances** .

REST = style d'architecture (conventions)

REST est l'acronyme de **R**epresentational **S**tate **T**ransfert.

C'est un **style d'architecture** qui a été décrit par *Roy Thomas Fielding* dans sa thèse «*Architectural Styles and the Design of Network-based Software Architectures*».

L'information de base, dans une architecture REST, est appelée **ressource**.
Toute information (à sémantique stable) qui peut être nommée est une ressource: un article, une photo, une personne, un service ou n'importe quel concept.

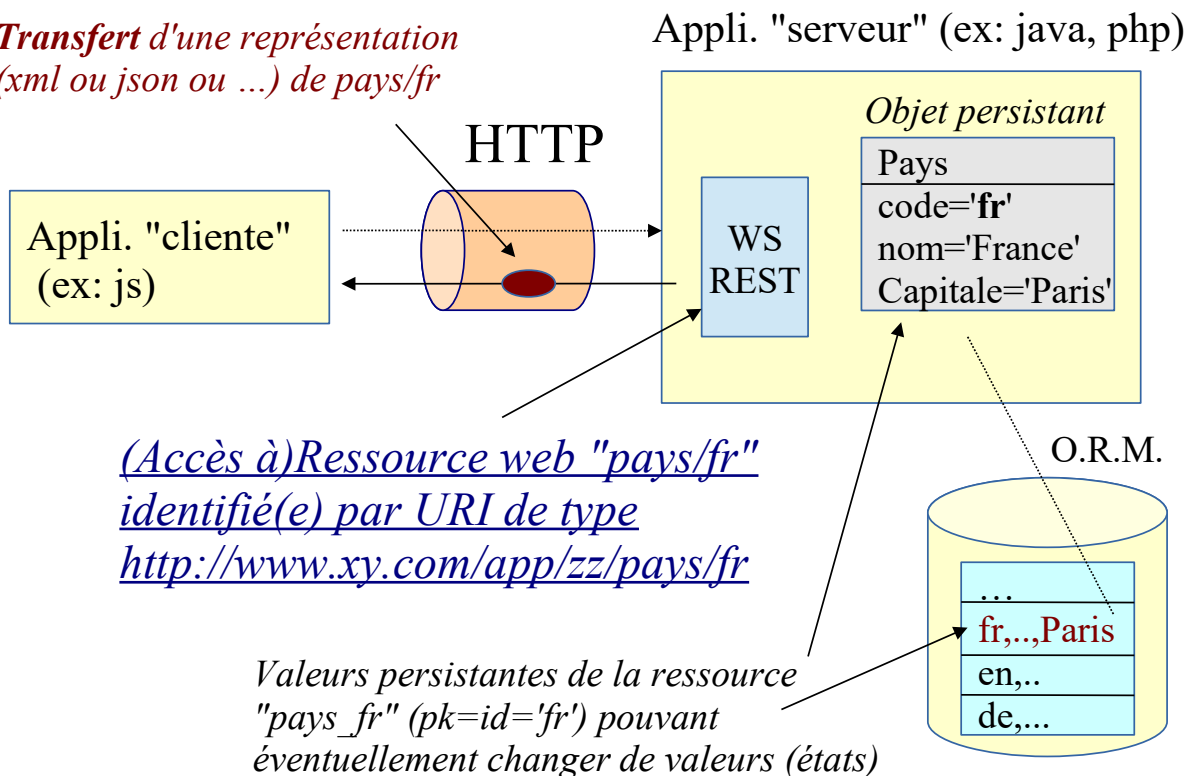
Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

NB: dans la plupart des cas, une ressource REST correspond indirectement à un enregistrement en base (avec la *clef primaire* comme partie finale de l'uri "identifiant").

Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.
Sur le web, on trouve aujourd'hui le plus souvent des représentations au format **HTML, XML ou JSON**.

REST : transferts de représentations de ressources

*Transfert d'une représentation
(xml ou json ou ...) de pays/fr*



REST et principaux formats (xml,json)

Une invocation d'URL de service REST peut être accompagnée de données (en entrée ou en sortie) pouvant prendre des formats quelconques :

text/plain , text/html , application/xml , application/json , ...

Dans le cas d'une lecture/recherche d'informations , le format du résultat retourné pourra (selon les cas) être :

- **imposé (en dur) par le code du service REST .**
- **au choix (xml , json) et précisé par une partie de l'url**
- **au choix (xml , json) et précisé par le champ "Accept :" de l'entête HTTP de la requête. (exemple: Accept: application/json) .**

Dans tous les cas, la réponse HTTP devra avoir son format précisé via le champ habituel ***Content-Type: application/json*** de l'entête.

Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques de JSON sont :

- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

```
[
  {
    "nom": "article a",
    "prix": 3.05,
    "disponible": false,
    "descriptif": "article1"
  },
  {
    "nom": "article b",
    "prix": 13.05,
    "disponible": true,
    "descriptif": null
  }
]
```

Les types de données valables sont :

- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

une liste d'articles

une personne

```
{
  "nom": "xxxx",
  "prenom": "yyyy",
  "age": 25
}
```

REST et méthodes HTTP (verbes)

Les méthodes HTTP sont utilisées pour indiquer la sémantique des actions demandées :

- **GET** : **lecture/recherche** d'information
- **POST** : **envoi** d'information
- **PUT** : **mise à jour** d'information
- **DELETE** : **suppression** d'information

Par exemple, pour récupérer la liste des adhérents d'un club, on peut effectuer une requête de type **GET** vers la ressource **<http://monsite.com/adherents>**

Pour obtenir que les adhérents ayant plus de 20 ans, la requête devient **<http://monsite.com/adherents?ageMinimum=20>**

Pour supprimer numéro 4, on peut employer une requête de type **DELETE** telle que **<http://monsite.com/adherents/4>**

Pour envoyer des informations, on utilise **POST** ou **PUT** en passant les informations dans le corps (invisible) du message HTTP avec comme URL celle de la ressource web que l'on veut créer ou mettre à jour.

Exemple concret de service REST : "Elevation API"

L'entreprise "**Google**" fournit gratuitement certains services WEB de type REST. "**Elevation API**" est un service REST de Google qui renvoie l'altitude d'un point de la planète selon ses coordonnées (latitude, longitude).

La documentation complète se trouve au bout de l'URL suivante :

<https://developers.google.com/maps/documentation/elevation/?hl=fr>

Sachant que les coordonnées du Mont blanc sont :

Lat/Lon : 45.8325 N / 6.86417 E (GPS : 32T 334120 5077656)

Les invocations suivantes (du service web rest "api/elevation")

<http://maps.googleapis.com/maps/api/elevation/json?locations=45.8325,6.86417>

<http://maps.googleapis.com/maps/api/elevation/xml?locations=45.8325,6.86417>

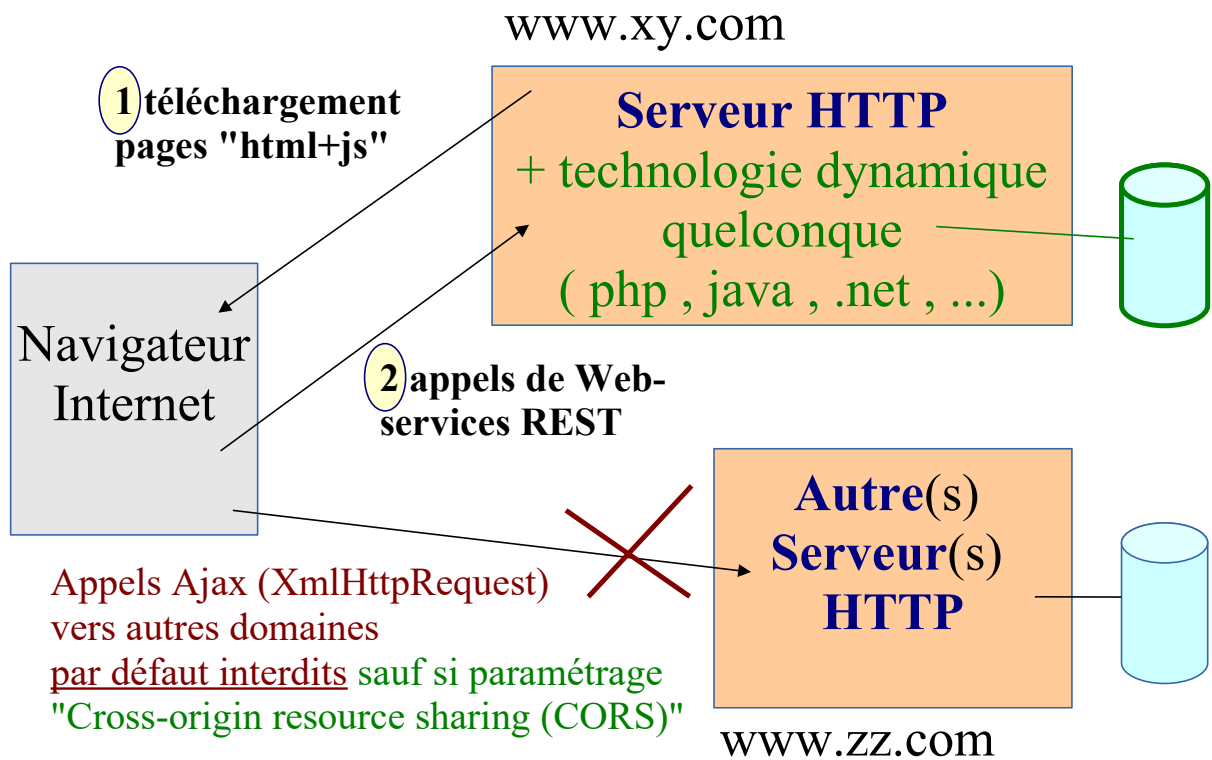
donne les résultats suivants "json" ou "xml":

```
{ "results" : [
  {
    "elevation" : 4766.466796875,
    "location" : {
      "lat" : 45.8325,
      "lng" : 6.86417
    },
    "resolution" : 152.7032318115234
  }
], "status" : "OK"
}
```

```
?xml version="1.0" encoding="UTF-8"?>
<ElevationResponse>
  <status>OK</status>
  <result>
    <location>
      <lat>45.8325000</lat>
      <lng>6.8641700</lng>
    </location>
    <elevation>4766.4667969</elevation>
    <resolution>152.7032318</resolution>
  </result>
</ElevationResponse>
```


2. Limitations Ajax sans CORS

Cadre des appels "html/js/ajax" vers services REST



3. CORS (Cross Origin Resource Sharing)

CORS=Cross Origin Resource Sharing

CORS est une **norme du W3C** qui précise certains **champs** à placer dans une **entête HTTP** qui serviront à échanger entre le navigateur et le serveur des informations qui serviront à décider si une requête sera ou pas acceptée.

(utile si domaines différents) , dans requête simple ou bien dans pré-échange préliminaire quelquefois déclenché en plus :

Au sein d'une requête "demande autorisation" envoyée du client vers le serveur :

Origin: <http://www.xy.com>

Dans la "réponse à demande d'autorisation" renvoyée par le serveur :

Access-Control-Allow-Origin: <http://www.xy.com>

Ou bien

Access-Control-Allow-Origin: * *(si public)*

→ requête acceptée

*Si absence de "Access-Control-Allow-Origin :" ou bien valeur différente
---> requête refusée*

CORS=Cross Origin Resource Sharing (2)

NB1: toute requête "CORS" valide doit absolument comporter le champ "**Origin** :" dans l'entête http. Ce champ est toujours construit automatiquement par le navigateur et jamais renseigné par programmation javascript.

Ceci ne protège que partiellement l'accès à certains serveurs car un "méchant hacker" utilise un "navigateur trafiqué".

Les mécanismes "CORS" protège un peu le client ordinaire (utilisant un vrai navigateur) que dans la mesure où la page d'origine n'a pas été interceptée ni trafiquée (l'utilisation conjointe de "https" est primordiale) .

NB2 : Dans le cas (très classique/fréquent) , où la requête comporte "**Content-Type: application/json**" (ou **application/xml** ou ...) , la norme "CORS" (considérant la requête comme étant "pas si simple") impose un pré-échange préliminaire appelé "**Preflighted request/response**" .

Paramétrages CORS à effectuer coté serveur

L'application qui coté serveur, fourni quelques Web Services REST , peut (et généralement doit) autoriser les requêtes "Ajax / CORS" issues d'autres domaines ("*" ou "www.xy.com").

Attention: ce n'est pas une "sécurité coté serveur" mais juste **un paramétrage autorisant ou pas à rendre service à d'autres domaines et en devant gérer la charge induite (taille du cluster, consommation électrique, ...)** .

// Exemple : CORS enabled with express/node-js :

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*"); // "*" ou "xy.com , ..."
  res.header("Access-Control-Allow-Methods",
    "POST, GET, PUT, DELETE, OPTIONS"); //default: GET, ...
  res.header("Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept , Authorization");
  next();
});
```

Paramétrages CORS avec CXF et JAX-RS

```

<bean id="corsFilter" class="org.apache.cxf.rs.security.cors.
    CrossOriginResourceSharingFilter">
    <!-- <property name="allowCredentials" value="true"/> -->
</bean>

...
<jaxrs:server id="myRestServices" address="/rest">
    <jaxrs:providers>
        <ref bean='jacksonJsonProvider' />
        <ref bean='corsFilter' />
    </jaxrs:providers>
    <jaxrs:serviceBeans> ...
        <ref bean="serviceClientsRest" />
    </jaxrs:serviceBeans> ...

```

config
spring/cxf

```

@Path("/json/gestionclients")
@Produces("application/json")
@Consumes("application/json")
@CrossOriginResourceSharing(allowAllOrigins = true)
    // ou bien autorisations plus fines
public class ClientRestJsonService {
    ...}

```

code java

XVI - Annexe – Essentiel HTML , CSS

...

...

XVII - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP