

Angular

v2,v4,v5,v6,v7,v8
(avec npm et typescript)

Table des matières

I - Présentation de Angular.....	4
1. Présentation du framework web Angular.....	4
II - Environnement de développement Angular.....	11
1. Environnement de développement pour Angular.....	11
III - Langage typescript.....	22
1. Bases syntaxiques du langage typescript (ts).....	22
2. Programmation objet avec typescript (ts).....	27
3. Lambda et generics et modules.....	36
4. Précautions/pièges "js" (et "ts").....	40
IV - Essentiel sur templates , bindings , events.....	41
1. Templates bindings (property , event).....	41

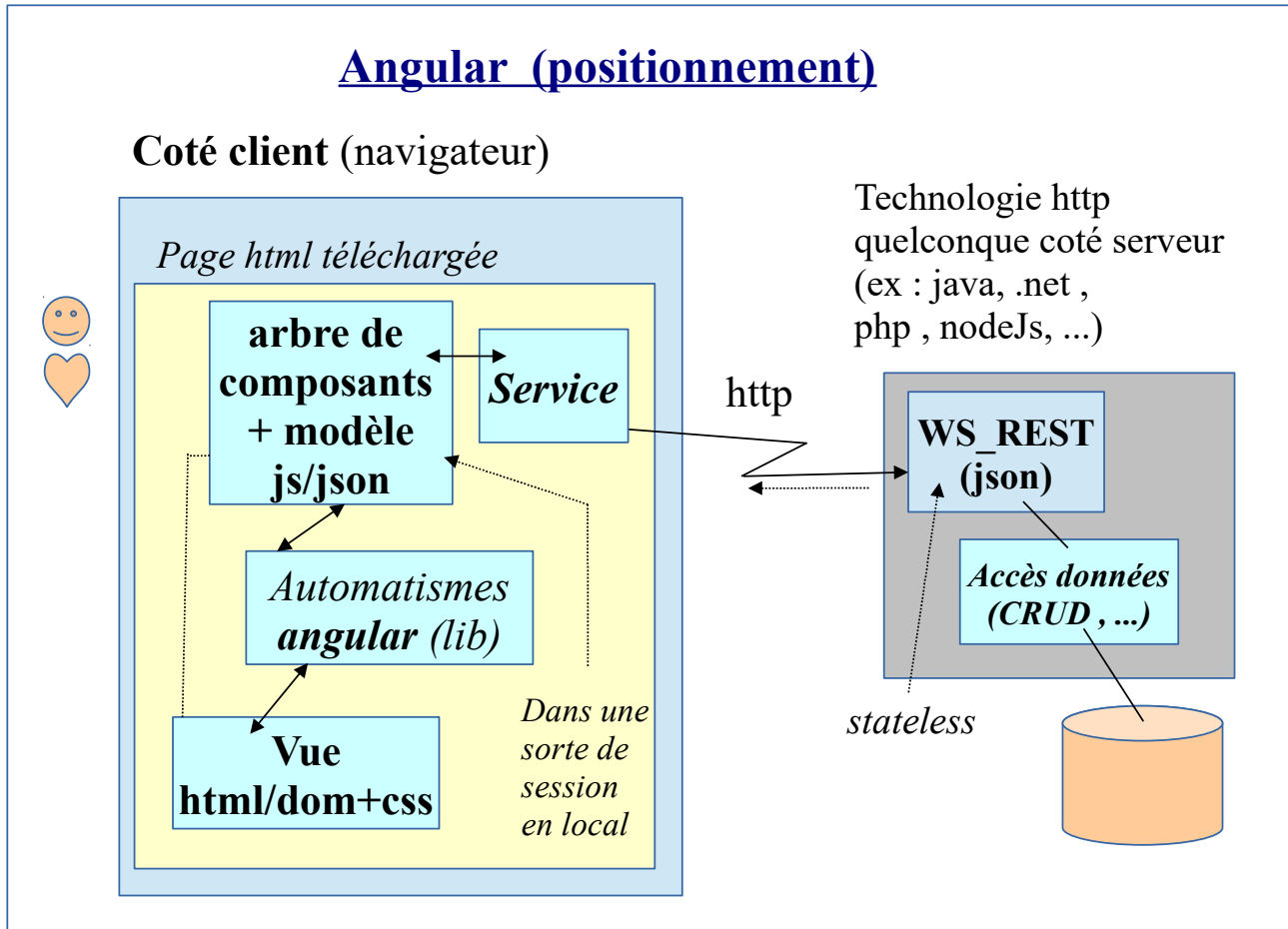
V - Contrôles de formulaires , composants GUI.....	49
1. Contrôle des formulaires.....	49
VI - Components (angular).....	56
1. Vue d'ensemble sur la structure du code Angular2.....	56
2. Les modules applicatifs.....	58
3. Les composants de l'application (@Component).....	59
4. Cycle de vie sur composants (et directives).....	65
VII - Directives , services et injections.....	67
1. Service.....	67
NB : Observable<...> ressemble un peu à Promise<...> et se consomme via ...	67
compteService.getComptesOfClientObservable(this.clientId).....	67
.subscribe(comptes =>this.comptes = comptes ,	67
error => console.log(error));.....	67
Observable (de rxjs) sera étudié de façon plus détaillée au sein d'un chapitre ultérieur (HTTP , ...)	67
2. Injection de dépendances.....	68
3. Les directives (angular2).....	70
VIII - Switch et routing (navigations).....	76
1. Navigation via ngSwitch et router de angular.....	76
2. Lazy loading.....	83
IX - Appels de W.S. REST (Observable, ...).....	85
1. Angular et dialogues HTTP/REST.....	85
2. Nouvelle Api HttpClient (depuis Angular 4.3).....	87
X - Aspects divers de angular (pipes, ...).....	90
1. BehaviorSubject.....	90
2. Autres aspects divers.....	94
XI - Packaging et déploiement d'appli. angular.....	97
1. déploiement avec ou sans "bundle".....	97
2. Angular-CLI en mode développement.....	98
3. Mise en production d'une application angular.....	100

XII - Tests unitaires (et ...) avec angular.....	103
1. Tests autour de angular.....	103
XIII - Sécurité – application "Angular2".....	120
1. Sécurisation d'une application "angular".....	120
2. Sécurisation des appels aux Web-services REST.....	121
3. Migration "AngularJs/v1.x" et Angular 2+.....	127
4. Utilisation du service Http avec Promise.....	128
5. Simulation d'appels HTTP via angular-in-memory-web-api.....	132
XIV - Annexe – Ancien Http (avant HttpClient).....	134
1. Utilisation du service Http avec Observable (rxjs).....	134
2. Retransmission des éléments de sécurité (v2,v4,v5).....	139
XV - Annexe – RxJs.....	140
1. introduction à RxJs.....	140
2. Fonctionnement (sources et consommations).....	141
3. Réorganisation de RxJs (avant et après v5,v6).....	141
4. Sources classiques générant des "Observables".....	143
5. Principaux opérateurs (à enchaîner via pipe).....	145
6. Passerelles entre "Observable" et "Promise".....	147
XVI - Annexe – Web Services REST (coté serveur).....	148
1. Généralités sur Web-Services REST.....	148
2. Limitations Ajax sans CORS.....	155
3. CORS (Cross Origin Resource Sharing).....	156

I - Présentation de Angular

1. Présentation du framework web Angular

1.1. Positionnement du framework "Angular"



Angular est un **framework web** de **Google** qui **s'exécute entièrement du coté navigateur** et dont la programmation est basée sur les langages **typescript** et **javascript** (es6 , es5).

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 :

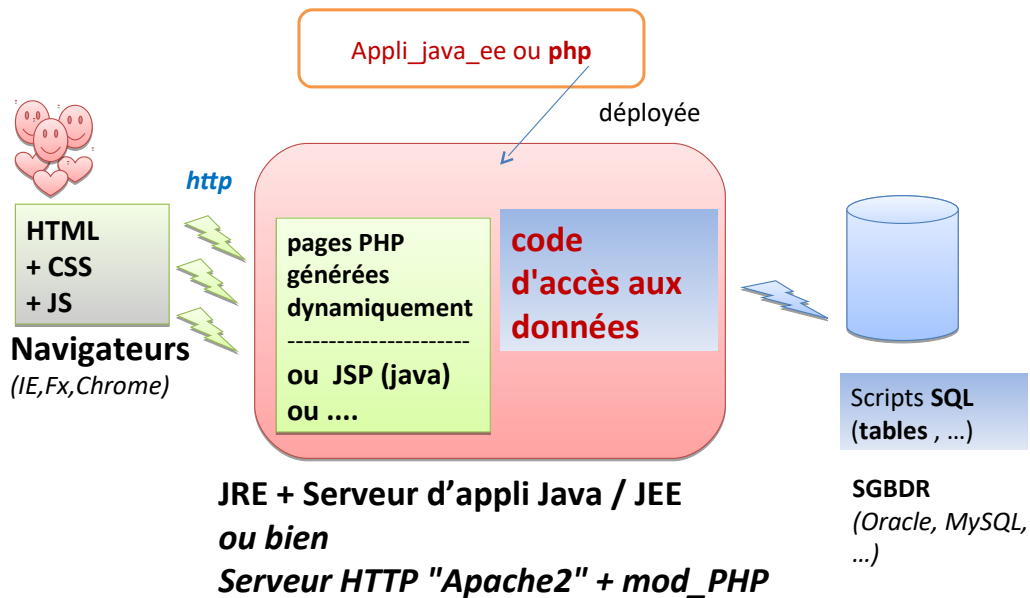
- une grande partie des traitements web s'effectue coté client (dans le navigateur) et le serveur se voit alors déchargé d'une lourde tâche (refabriquer des pages , gérer les sessions utilisateurs, ...) ---> bien pour tenir la charge .
- meilleurs performances/réactivités du coté affichage/présentation 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. Contexte architectural

Ancienne architecture web prédominante (années 1995-2015) :

Pages HTML générées coté serveur (ex : java/JEE , php, asp, ...) et sessions HTTP coté serveur .

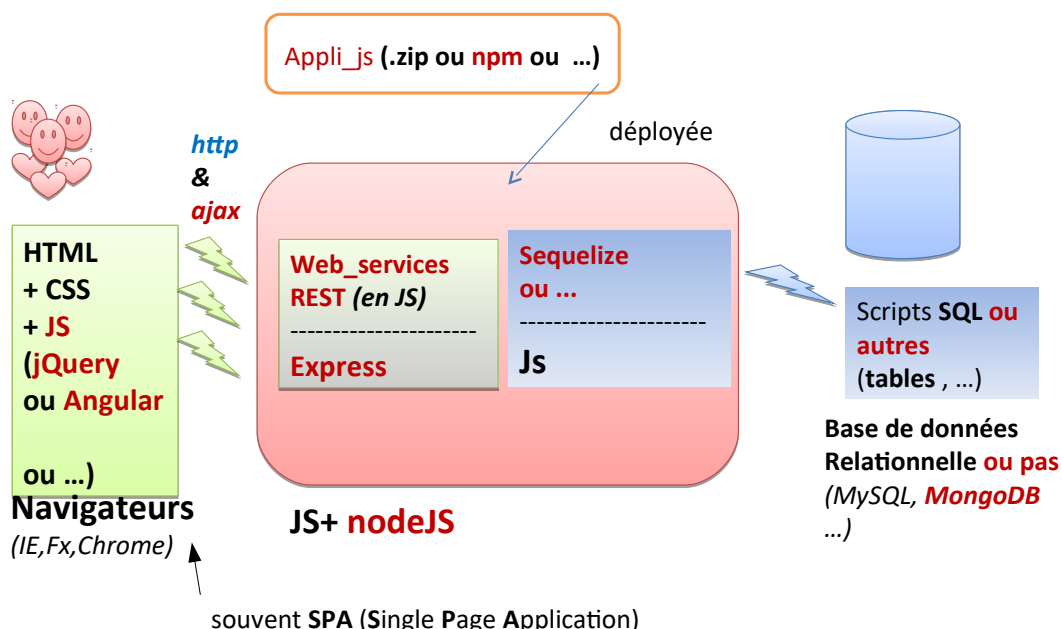
Ancienne architecture web



Nouvelle architecture web prédominante (depuis 2015 environ) :

Front-end (ex : Angular, Vue-Js , react, ...) en HTML5/CSS3/JS invoquant (via ajax) des Web services REST d'un "backend" serveur quelconque (nodeJs, php , java/JEE/spring , ...) .

Env exécution NodeJs



--> avantages : meilleurs performances (si grand nombre de clients simultanés) et meilleur séparation front-end (affichage standard HTML5/CSS3) / back-end (api rest) .

1.3. Binding angular

Composant angular avec binding automatique (*)

(*) *m-v-vm (proche mvc)
en javascript / navigateur*

product.component.html

label:	<input type="text" value="produit xyz"/>
prix ht:	<input type="text" value="200"/>
taux tva:	<input type="text" value="20"/>
<input type="button" value="actualiser prix ttc"/>	
prix ttc:	240

product.component.ts

```
@Component({ ... })
export class ProductComponent {
  product = new Product();
  ....
  OnMajTtc = function(evt) { ... }
}
```

Product (class)

```
.label produit xyz
.ht 200
.tva 20
.ttc 240
```

[(ngModel)]
="product.tva"

{{product.ttc}}

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** .

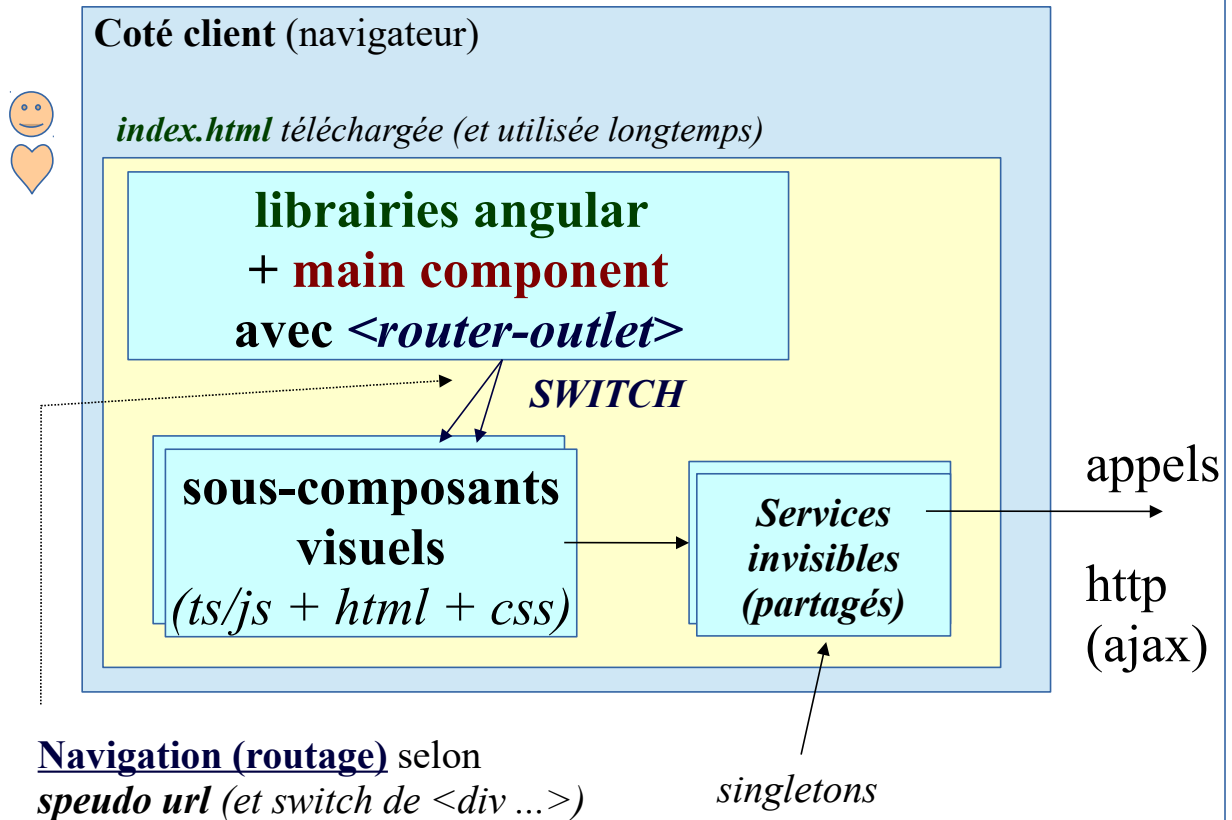
Quelque soit la version d'angular (1 , 2, 4 ou +) , le **binding automatique** entre valeurs saisies ou affichées et les valeurs des objets "javascript" constitue **la principale valeur ajoutée du framework**.

C'est le principal apport d'Angular par rapport à une application simplement basé sur jquery .

NB : AngularJs (v1.x) utilisait un binding systématiquement bidirectionnel et assez peu performant. Angular (v2, v4, ...) utilise maintenant un binding mieux contrôlé (soit unidirectionnel , soit bidirectionnel) et est plus performant.

1.4. Structure "Single Page" et routage angular

Single Page Application et switch de sous parties



De façon à ce que le code javascript (librairies angular + code de l'application) soit converti en mémoire sur le long terme, une application Angular est constituée d'**une seule grande page "index.html"** qui est **elle-même décomposée en une hiérarchie de composants** (ex : header, footer, content, ...). On parle généralement en terme de "**SPA : Single Page Application**" pour désigner cette architecture web (très classique).

Le **composant principal** ("main.component", ".ts", ".html") **comporte** très souvent une balise spéciale `<router-outlet></router-outlet>` (fonctionnellement proche de `<div />`) dont le **contenu (interchangeable)** sera automatiquement remplacé par un des sous composants importants de l'application.

Le **switch de sous composants** sera associé à des **navigations** généralement **paramétrées dans un module de routage**.

En pouvant associer une pseudo-URL relative à l'affichage contrôlé d'un certain sous composant précis, il est ainsi possible de mémoriser des "bookmarks / favoris / marques-pages" dans un navigateur.

1.5. Orientation "composants"

Contrairement à l'ancienne version 1.x , les nouvelles versions 2,4,6,... du framework angular sont clairement orientées "composants" .

Il s'agit là d'une évolution récente des technologies web .

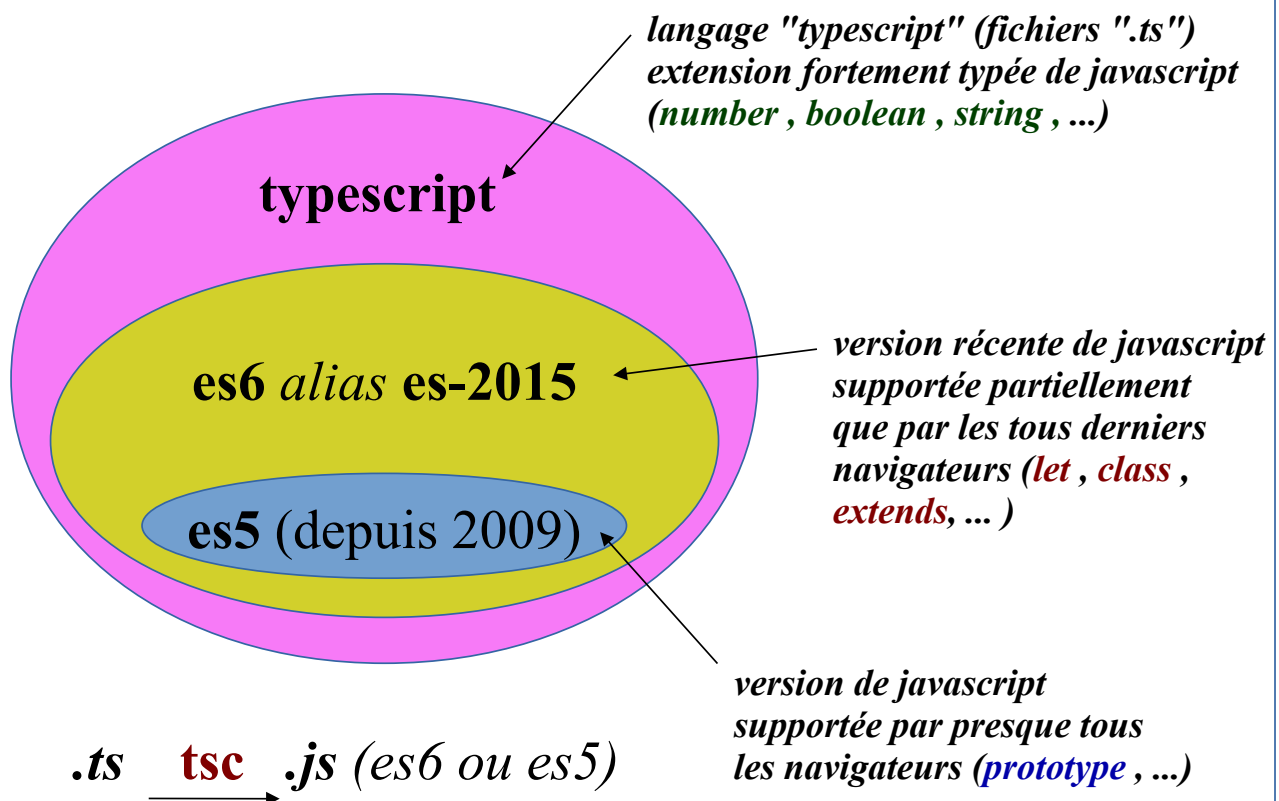
"Web Component"

"..."

1.6. Particularités du framework "Angular" (v2, v4, ... , v6, ...)

- le code d'une application angular est bien structuré (orienté objet / syntaxe rigoureuse grâce à typescript) et est "à peu près maintenable" .
- le framework "angular" est dès le départ très complet (rendu , binding , routage , appels ajax/http , ...) , ce qui n'est volontairement pas le cas de certains autres frameworks concurrents (backbone , react , knockout-js, ...)
- l'environnement de développement est basé (depuis la v2) sur **npm** et **@angular/cli** et est maintenant très complet (tests , génération de bundles , ...)

Fonctionnalités "es5" , "es6" et "typescript"



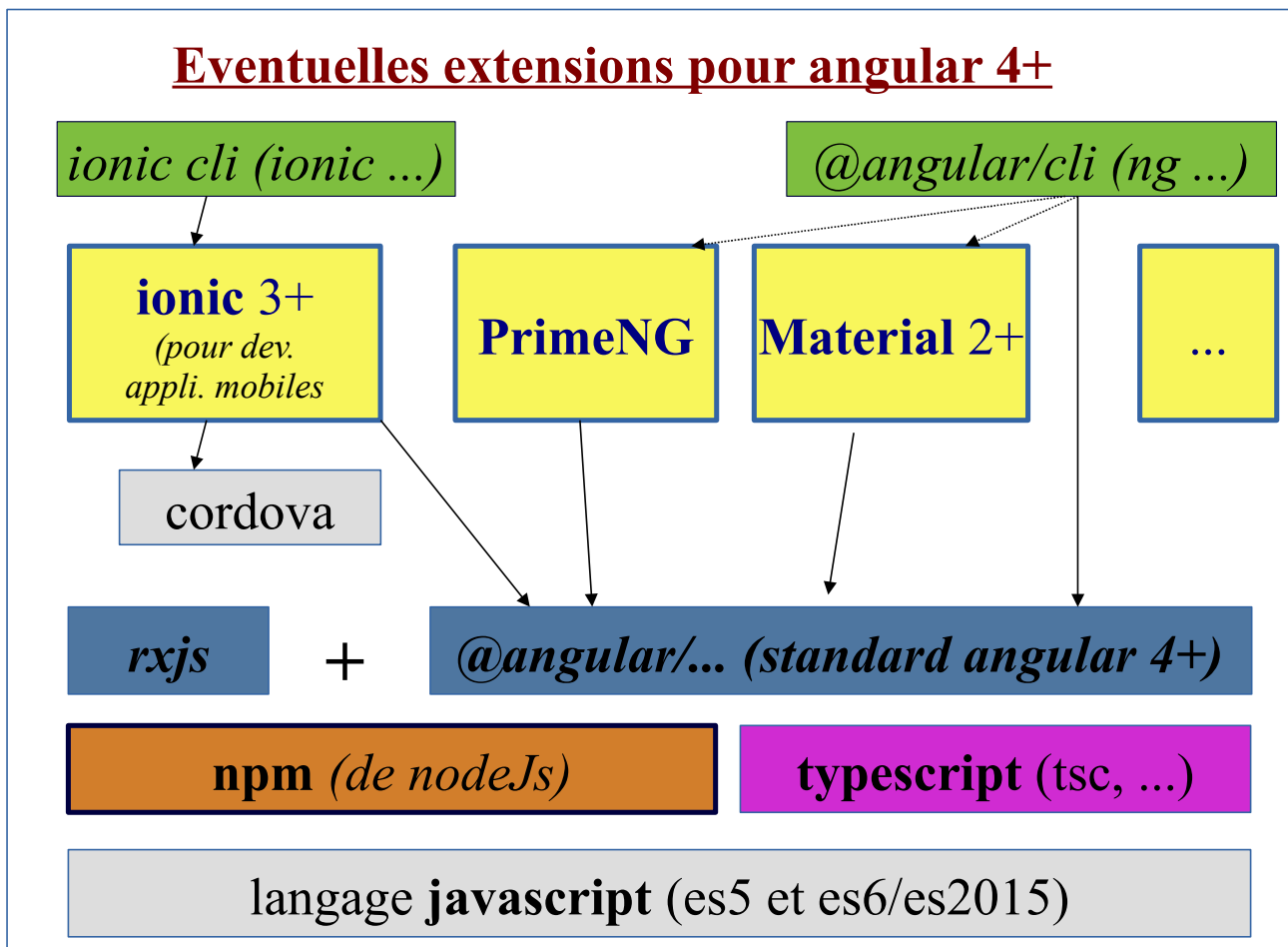
NB:

- Angular Js (1.x) n'était basé que sur javascript/es5 et ne nécessitait aucun environnement de développement sophistiqué (un simple "notepad++" suffisait).
En contre partie de cet environnement de développement simpliste, le code d'une application

"Angular Js / 1.x" était assez rapidement complexe à maintenir (pas adapté aux applications de grandes tailles).

- Depuis la v2 , "Angular" n'est plus qualifié de "Js" et s'appuie sur un environnement de développement beaucoup plus sophistiqué (npm + @angular/cli + typescript) et très complet (tests , générations de "bundles" ,).
- Depuis la V2 d'angular les composants sont codés en typescript "typé et orienté objet" (.ts)
- A court terme les fichiers ".ts" sont traduits en ".js" (es5) de façon à pouvoir être interprétés par presque tous les navigateurs des années 2010-2018 .
- La V2 a été très éphémère (fin 2016 / début 2017) , la V3 n'a jamais existé et la **version 4** (fin 2017) de Angular est enfin à considérée comme une version mature et stable .

1.7. Eventuelles extensions pour angular



PrimeNG et **Material 2** sont deux extensions concurrentes qui sont constituées d'un ensemble homogène de **nouveaux composants graphiques réutilisables** (ex : tabs/onglets , menus déroulants , panels , ...)

En utilisant une bibliothèque additionnelle (PrimeNG ou Material 2) , le développement concret d'une application angular 4+ s'appuie sur de nouvelles balises prêtes à l'emploi et facilement paramétrables .

--> avantage : code plus compact et intégration naturelle dans le reste du code applicatif angular.

--> inconvénient : balisages et paramétrages très spécifiques (moins portables que du classique "html5+css3" ou que de l'habituel "bootstrap-css + réflexes jquery") .

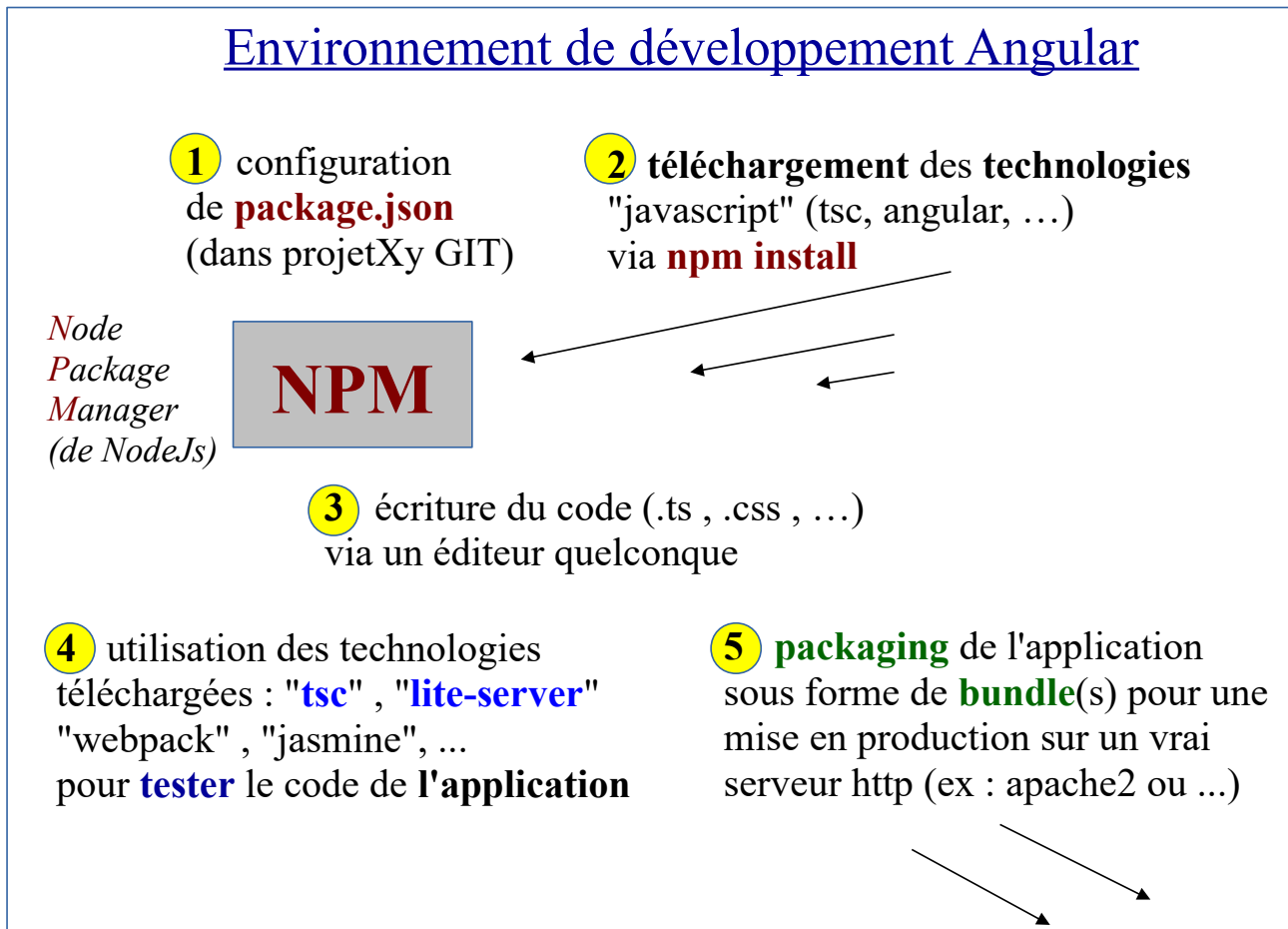
NB : Angular 1.x s'appuyait en interne sur "jquery lite" . Depuis la v2 , Angular ne s'appuie plus du tout sur jquery. Il est éventuellement possible d'utiliser "jquery" en plus d'angular 4 (par exemple pour accompagner bootstrap-css) mais avec quelques limitations (compatibilité / complémentarité au cas par cas selon versions et fonctionnalités utilisées) .

ionic 3+ est une **extension** de angular 4+ qui *s'appuie en interne sur "apache cordova"* et qui **permet de développer des applications mobiles hybrides (en partie "web" , en partie native) pour les smartphones "ios/iphone" , "android" , "windows" , ...**

II - Environnement de développement Angular

1. Environnement de développement pour Angular

1.1. Environnement de développement minimum (depuis v2)



Bien que Angular soit une technologie qui s'exécute "coté navigateur", l'environnement de développement s'appuie sur la sous partie "**npm**" de nodeJs.

L' **éco-système "npm"** sert essentiellement à télécharger et exécuter les technologies de développement nécessaires pour angular ("tsc", "@angular/cli", ...).

A l'époque des premières "v2" de Angular, le mode de développement préconisé consistait à directement partir d'un fichier "package.json" récupéré par copier/coller et éventuellement adapté pour ensuite lancer "npm install", etc ...

Bien qu'encore possible actuellement, ce mode de développement basique et direct est de moins en moins utilisé au profit de l'utilitaire en ligne de commande "@angular/cli" (exposé au sein du prochain paragraphe).

1.2. Développement Angular basé sur @angular/cli (ng)

incontournable @angular/cli

S'installant via ***npm install -g @angular/cli*** , **angular CLI** est un *utilitaire en ligne de commandes* (s'appuyant sur *npm* et *webpack*) permettant de gérer toutes les phases d'un projet angular :

ng new my-app -- création d'une nouvelle appli angular4+

ng g component cxy -- génération d'un nouveau composant

ng g service sa -- génération d'un nouveau service

ng g ...

ng serve -- build en mémoire + démarrage serveur de test

ng build --prod -- construction de bundles (pour déploiement)

ng ...

Angular-CLI est maintenant officiellement préconisé sur le site officiel de Angular. 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**

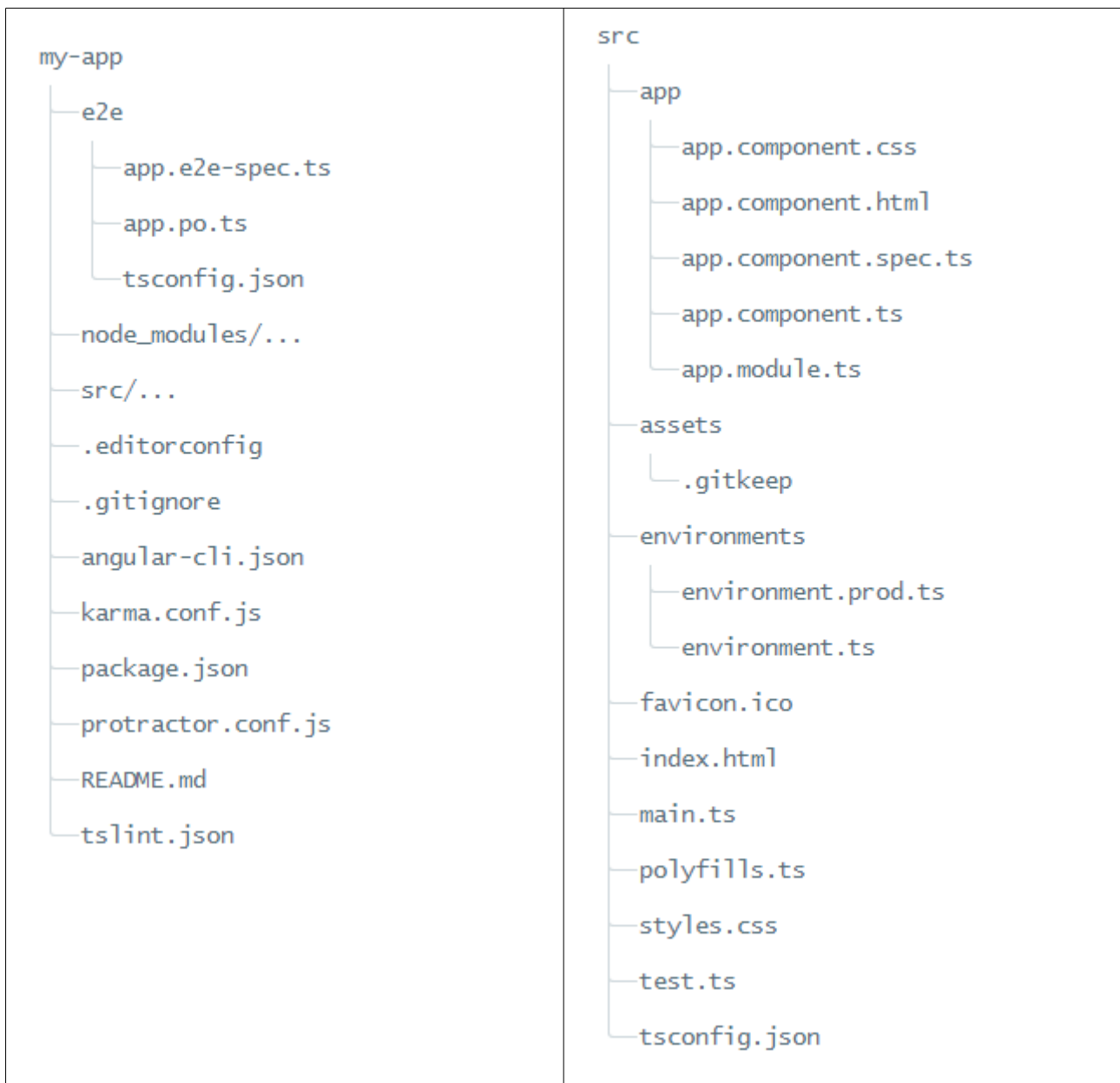
NB : si besoin , upgrade préalable de npm via **npm install npm@latest -g** ou bien carrément désinstaller nodejs et réinstaller une version plus récente.

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).

Au sein de l'arborescence des répertoires et fichiers créés (voir ci-après) :

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

* **e2e** correspond à "**end to end tests**" (lancés via **ng e2e** et *protractor*) .



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

ng new my-app , cd my-app	Création d'une nouvelle application "my-app" .
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 --target=production --environment=prod)	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) :

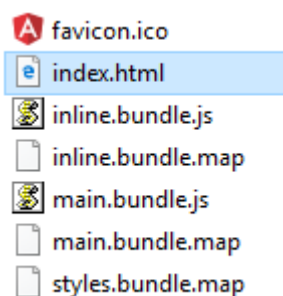
Scaffold (échafaudage)	Usage (ligne de commande)
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Interface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

Migration locale du seul projet courant vers version plus récente :

```
ng update @angular/cli @angular/core [ --to=7.2.0 ]
```

Migration globale de angular-cli vers version plus récente :

```
npm uninstall -g angular-cli  
npm cache clean or npm cache verify (if npm > 5)  
npm install -g @angular/cli@latest
```

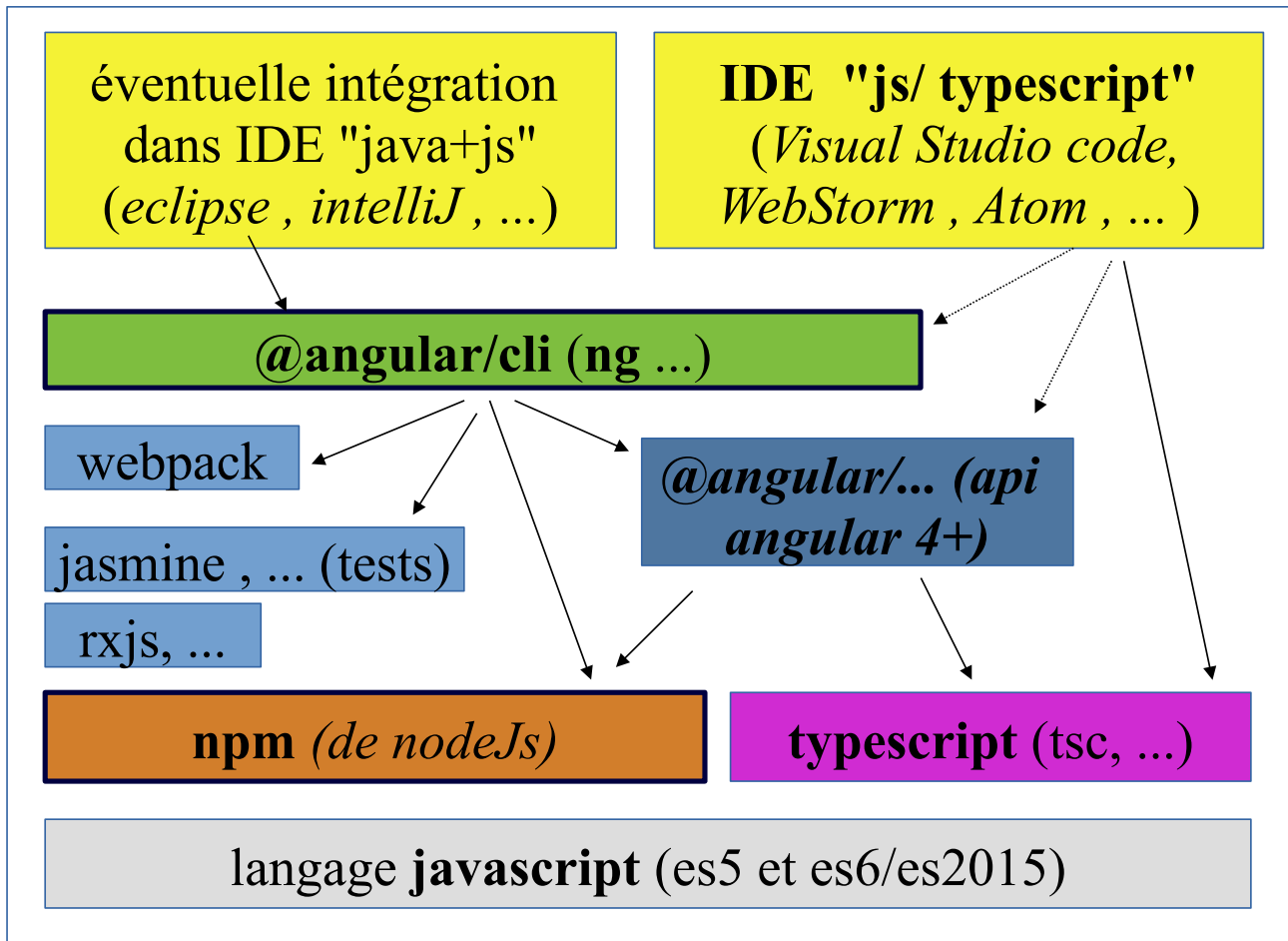
Ajout de bibliothèque(s) javascript :

NB : si l'on souhaite utiliser conjointement certains fichiers javascripts complémentaires (exemple pas forcément conseillé mais quelquefois compatible : "jquery...js" et "bootstrap.min.js") on peut :

- 1) placer un répertoire "js" a coté de node-modules (ou bien utiliser un jquery récupéré par npm)
- 2) adapter le fichier **angular-cli.json** ou **angular.json** de la façon suivante :

```
"scripts": [ "../js/jquery-3.2.1.min.js" ,  
              "../js/bootstrap.min.js"],
```

1.3. IDE et éditeurs de code pour angular



1.4. Configuration "npm" et "@angular/cli" pour Angular

Voici un exemple de fichier **"package.json"** généré par **"ng new my-app"** :

```
{
  "name": "my-app",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^4.2.4",
    "@angular/common": "^4.2.4",
    "@angular/compiler": "^4.2.4",
    "@angular/core": "^4.2.4",
    "@angular/forms": "^4.2.4",
    "@angular/http": "^4.2.4",
    "@angular/platform-browser": "^4.2.4",
    "@angular/platform-browser-dynamic": "^4.2.4",
    "@angular/router": "^4.2.4",
    "core-js": "^2.4.1",
    "rxjs": "^5.4.2",
    "zone.js": "^0.8.14"
  },
  "devDependencies": {
    "@angular/cli": "1.4.2",
    "@angular/compiler-cli": "^4.2.4",
    "@angular/language-service": "^4.2.4",
    "@types/jasmine": "~2.5.53",
    "@types/jasminewd2": "~2.0.2",
    "@types/node": "~6.0.60",
    "codemlizer": "~3.1.1",
    "jasmine-core": "~2.6.2",
    "jasmine-spec-reporter": "~4.1.0",
    "karma": "~1.7.0",
    "karma-chrome-launcher": "~2.1.1",
    "karma-cli": "~1.0.1",
    "karma-coverage-istanbul-reporter": "^1.2.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.1.2",
    "ts-node": "~3.2.0",
    "tslint": "~5.3.2",
    "typescript": "~2.3.3"
  }
}
```

Voici un exemple de fichier **tsconfig.json** généré :

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [ "es2017", "dom" ]
  }
}
```

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

Fichier **"main.ts"** (généré et ré-exploité par @angular/cli) pour charger et démarrer le code de l'application :

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

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

Page principale **index.html** (qui sera retraitée/enrichie via **ng serve** ou **ng build**)

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>my Angular4+ app</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

NB : les fichiers index.html et styles.css peuvent être un petit peu personnalisés mais le gros du code de l'application sera placé dans les sous-répertoires "app" et autres .

1.5. Exemple de code élémentaire pour une application angular

app/app.component.ts

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

@Component({
  selector: 'app-root',
  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";
  }
}
```

My First Angular 2 App ..

i	i*i
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81

NB : dans @Component() ,

- soit **templateUrl**: 'app.component.html' (template html dans fichier annexe)
- soit **template**: `<h1>... {{message}} </h1> ...` (contenu direct d'un petit template html entre quotes inverses)

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 ],
  providers: [ ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

à lancer et tester via **ng serve**
et **http://localhost:4200**

...

1.6. Styles css globaux et styles spécifiques à un composant .

xyz.component.ts

```
...
@Component({
  selector: 'xyz',
  templateUrl: './xyz.component.html',
  styleUrls: ['./xyz.component.css']
})
export class XyzComponent implements OnInit {
  ...
}
```

xyz.component.css

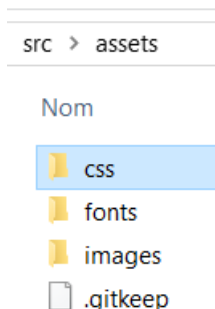
```
input.ng-valid[required] {
  border-left: 5px solid #42A948; /* green */
}
input.ng-invalid {
  border-left: 5px solid #a94442; /* red */
}
.errMsg{
  font-style: italic;
}
```

Ces classes de styles css ne sont utilisées que par le composant "xyz" . Il n'y aura pas d'effet de bord (pas d'éventuels conflits ou perturbations) avec d'autres composants .

Pour utiliser des styles css au niveau global (toute l'application) , on peut dans un contexte "angular-cli" les référencer dans la partie "styles" de *.angular-cli.json*

```
...
"styles": [
  "styles.css" , "assets/css/bootstrap.css"
],
...
```

NB : Les chemins sont à exprimer de façon relative à *src* (la où est placé *index.html*) .



1.7. Lien classique entre angular 4+ et bootstrap-css 3 ou 4

Avertissement préalable :

Bootstrap-css vient assez récemment de basculer de la version 3.x à la toute nouvelle version 4.0.0
Il y a beaucoup de changements ("panel" --> "card" , ...)

Etant donné que bootstrap4 n'est plus livré/accompagné de glyphicons, on peut utiliser les styles *font-awesome* à la place (restant partiellement gratuit/open-source) .

Téléchargement (et installation dans l'appli) de bootstrap-css via npm :

```
npm install bootstrap --save
npm install font-awesome --save
```

NB : par défaut , la dernière version stable de bootstrap-css est téléchargée (actuellement 4).
on peut (si besoin) préciser la version de bootstrap souhaitée lors du "npm install" .

dans .angular-cli.json (.angular.json) :

```
"styles": [
  "src/styles.css" ,
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "node_modules/font-awesome/css/font-awesome.min.css"
],
```

NB :

- Dans anciennes versions 4 et 5 : **angular-cli.json** et chemins relatifs (de src) vers des styles préalablement récupérés via npm : **"../node_modules/.../bootstrap.min.css"** , **"../node_modules/..."** .
- Depuis angular6 , **angular.json** et chemins relatifs depuis la racine du projet .

Petit test dans un ...component.html:

```
...
<input type="button" value="ok" class="btn btn-primary" />
<i class="fa fa-heart" aria-hidden="true" style="color: red;"></i>
...
```

III - Langage typescript

1. Bases syntaxiques du langage typescript (ts)

1.1. Rappel: 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 basée sur "typescript" doit être compilée/transpilée en ES5 de façon à pouvoir s'exécuter sur n'importe quel navigateur.

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

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

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.

La version **2.3.2** est l'une des versions les plus récentes de **typescript** (mi 2017) .

1.3. utilisation concrète de tsc

Après une installation globale effectuée par "`npm install -g typescript`", une ligne de commande en "`tsc`" ou "`tsc -w`" lancée depuis le répertoire d'un projet où est présent le fichier **tsconfig.json** suffit à lancer toute une série de compilations "typescript" :

<p>tsconfig.json</p> <pre>{ "compileOnSave": true, "compilerOptions": { "baseUrl": "", "declaration": false, "emitDecoratorMetadata": false, "experimentalDecorators": false, "outDir": "dist/out-tsc", "sourceMap": true, "target": "es5", "noEmitOnError" : false }, "include": ["src/**/*"], "exclude": ["node_modules", "**/*.spec.ts", "dist"] }</pre>	
--	--

Lorsque l'option "**noEmitOnError** : **true**" est fixée, le fichier ".js" n'est pas généré en cas d'erreur (pourtant ordinairement non bloquante) soulevée par tsc.

Avec l'option "**sourceMap** : **true**", le compilateur/transpilateur "tsc" génère des fichiers ".map" à côté des fichiers ".js".

Ces fichiers dénommés "**source map**" servent à effectuer des correspondances entre le code source original ".ts" et le code transformé ".js".

Les fichiers ".map" sont quelquefois chargés et interprétés par des débogueurs sophistiqués (ex : debug de certains IDE, ...). Ces fichiers ".map" ne sont pas indispensables pour l'interprétation du code ".js" généré et peuvent (éventuellement) être omis en production.

Quelques Bons éditeurs de code (pour langage "typescript") :

- * *Visual Studio Code*
- * *Atom*
- * *WebStorm*

1.4. 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];

1.5. Tableaux (construction et parcours)

```
var tableau : string[] = new Array<string>();
```

```
//tableau.push("abc");
```

```
//tableau.push("def");
```

```
tableau[0] = "abc";
```

```
tableau[1] = "def";
```

Au moins 3 parcours possibles:

```
var n : number = tableau.length;  
for(let i = 0; i < n; i++) {  
    console.log(">> at index " + i + " value = " + tableau[i] );  
}
```

```
for(let i in tableau) {  
    console.log("** at index " + i + " value = " + tableau[i] );  
}
```

```
for( let s of tableau){  
    console.log("## val = " + s );  
}
```

1.6. Portées (var , let) et constantes (const)

Depuis longtemps (en javascript) , le mot clef "**var**" permet de déclarer explicitement une variable dont la portée dépend de l'endroit de sa déclaration (globale ou dans une fonction).

Sans aucune déclaration, une variable (affectée à la volée) est globale et cela risque d'engendrer des effets de bords (incontrôlés) . Ceci est maintenant interdit en "typescript".

Introduits depuis es6/es2015 et typescript 1.4 , les mots clefs **let** et **const** apportent de nouveaux comportements :

- Une variable déclarée via le mot clef **let** a une *portée limité au bloc local* (exemple boucle for) . Il n'y a alors pas de collision avec une éventuelle autre variable de même nom déclarée quelques ligne au dessus du bloc d'instructions (entre {} , de la boucle).
- Une variable déclarée via le mot clef **const** *ne peut plus changer de valeur après la première affectation*. Il s'agit d'une **constante** .

Exemple :

```
const PISur2 = Math.PI / 2;
//PISur2=2; // Error, can't assign to a `const`
console.log("PISur2 = " + PISur2);

var tableau : string[] = new Array<string>();
tableau[0] = "abc";
tableau[1] = "def";

var i : number = 5;
var j : number = 5;

//for(let i in tableau) {
for(let i=0; i<tableau.length; i++) {
    console.log("*** at index " + i + " value = " + tableau[i] );
}

//for(j=0; j<tableau.length; j++) {
for(var j=0; j<tableau.length; j++) {
    console.log("### at index " + j + " value = " + tableau[j] );
}

console.log("i=" + i); //affiche i=5
console.log("j=" + j); //affiche j=2
```

2. Programmation objet avec typescript (ts)

2.1. Classe et instances

```
class Compte{
  numero : number;
  label: string;
  solde : number;

  debiter(montant : number) : void {
    this.solde -= montant; // this.solde = this.solde - montant;
  }

  crediter(montant : number) : void {
    this.solde += montant; // this.solde = this.solde + montant;
  }
}
```

```
var c1 = new Compte(); //instance (exemplaire) 1
console.log("numero et label de c1: " + c1.numero + " " + c1.label);
console.log("solde de c1: " + c1.solde);

var c2 = new Compte(); //instance (exemplaire) 2
c2.solde = 100.0;
c2.crediter(50.0);
console.log("solde de c2: " + c2.solde); //150.0
```

NB: Sans initialisation explicite (via constructeur ou autre) , les propriétés internes d'un objet sont par défaut à la valeur **"undefined"** .

2.2. constructor

Un constructeur est une méthode qui sert à initialiser les valeurs internes d'une instance dès sa construction (dès l'appel à new) .

En langage typescript le constructeur se programme comme la méthode spéciale "**constructor**" (mot clef du langage) :

```
class Compte{
    numero : number;
    label: string;
    solde : number;

    constructor(numero:number, libelle:string, soldeInitial:number){
        this.numero = numero;
        this.label = libelle;
        this.solde = soldeInitial;
    }

    //...
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
c1.crediter(50.0);
console.log("solde de c1: " + c1.solde);
```

NB: il n'est pas possible d'écrire plusieurs versions du constructeur :

```
constructor(numero:number, libelle:string, soldeInitial:number){
    this.numero = numero;
    this.label = libelle;
    this.solde = soldeInitial;
}
```

```
constructor(){
    this.=0;
    this.label="?";
    this.=0.0;
}
```

Il faut donc quasi systématiquement utiliser la syntaxe = *valeur_par_defaut* sur les arguments d'un constructeur pour pouvoir créer une nouvelle instance en précisant plus ou moins d'informations lors de la construction :

```
class Compte{
    numero : number;
    label: string;
    solde : number;

    constructor(numero:number=0, libelle:string="?", soldeInitial:number=0.0){
        this.numero = numero;
```

```

        this.label = libelle;
        this.solde = soldeInitial;
    } //...
}

```

```

var c1 = new Compte(1,"compte 1",100.0);
var c2 = new Compte(2,"compte 2");
var c3 = new Compte(3);
var c4 = new Compte();

```

Remarque : en plaçant le mot clef (récent) **"readonly"** devant un attribut (propriété) , la valeur de celui ci doit absolument être initialisée dès le constructeur et ne pourra plus changer par la suite.

2.3. 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) !!!

2.4. Accesseurs automatiques `get xxx()` / `set xxx()`

```
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()
console.log("size=" + a1.size) ; // calling get size() → affiche size=120
```

NB: `get ...` et `set ...` nécessitent l'option `-t ES5` de `tsc`

2.5. Mixage "structure & constructor" avec `public` ou `private`

```
class Compte{
  numero : number;
  label : string;
  solde : number;

  constructor(public numero : number=0,
    public label : string="?",
    public solde : number=0.0){
    this.numero = numero;
    this.label = label; this.solde = solde;
  } //...
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
c1.solde = 250.0; console.log(c1.numero + ' ' + c1.label + ' ' + c1.solde );
```

Remarque importante : via le mot clef "**public**" ou "**private**" ou "**protected**" (au niveau des paramètres du constructeur) , certains paramètres passés au niveau du constructeur sont automatiquement transformés en attributs/propriétés de la classe .

Autrement dit, toutes les lignes "barrées" de l'exemple précédent sont alors générées implicitement (automatiquement) .

2.6. mot clef "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());
```

2.7. 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.

```

NB: depuis la version 1.3 de typescript , le mot clef "**protected**" peut être utilisé dans une classe de base à la place de private et les méthodes des sous classes (qui hériteront de la classe de base) pourront alors accéder directement au attributs/propriétés "*protected*" .

2.8. Classes abstraites (avec opérations abstraites)

```

...
// classe abstraite (avec au moins une méthode abstraite / sans code):
abstract class Fig2D {
  constructor(public lineColor : string = "black",
    public lineWidth : number = 1,
    public fillColor : string = null){
  }
  performVisit(visitor : FigVisitor) : void {}
  abstract performVisit(visitor : FigVisitor) : void ;
}

// classe concrète (avec du code d'implémentation pour chaque opération):
class Line extends Fig2D{
  constructor(public x1:number = 0 , public y1:number = 0 ,
    public x2:number = 0 , public y2:number = 0,
    lineColor : string = "black",
    lineWidth : number = 1){
    super(lineColor,lineWidth);
  }
  performVisit(visitor : FigVisitor) : void {
    visitor.doActionForLine(this);
  }
}

```

```

var tabFig : Fig2D[] = new Array<Fig2D>();
tabFig.push( new Fig2D("blue") ); //impossible d'instancier une classe abstraite
tabFig.push( new Line(20,20,180,200,"red") ); //on ne peut instancier que des classes concrètes

```


2.9. Interfaces

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);
```

Rappel important : 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;
  }
  //...
}
```

3. Lambda et generics et modules

3.1. 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;

```

NB: la technologie "**RxJs**" utilisée par angular2 utilise beaucoup de "lambda expressions" .

3.2. 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;
}
```

3.3. Modules "typescript"

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

internes/logiques (rarement utilisés)	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 .	Toujours un fichier par module externe (nom du module = nom du fichier) Selon contexte (nodeJs ou ...)

Précision importante :

Beaucoup de technologies javascript modernes s'exécutent dans un environnement prenant en charge des modules (bien délimités) de code (avec import/export) .

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

- **CommonJS (cjs)** – modules "synchrones" , **syntaxe** "var xyz = **requires**('xyz')"
NB : node (nodeJs) utilise partiellement les idées et syntaxes de CommonJS .
- **AMD** (Asynchronous **M**odule **D**efinition) 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 **m**odule **d**efinition) – *fichiers xyz.umd.js*
- **iife** (immediately-invoked **f**unction **e**xpression) – *fonctions anonymes auto-exécutées*

3.4. Modules (externes) "es2015" vs modules "cjs" :

L'un des principaux atouts de la structure des "modules es2015" tient dans les imports statiques et précis qui peuvent ainsi être analysés pour une génération optimisée des bundles à déployer en production.

Au lieu d'écrire `var/const xyModule = requires('xyModule')` comme en "cjs", la syntaxe plus précise de es2015 permet d'écrire

```
import { Composant1 , ... , ComposantN } from 'xyModule' .
```

Ainsi l'optimisation dite "**Tree-Shaking**" (de webpack ou rollup ou ...) permet d'exclure tous les composants jamais utilisés de certaines librairies et la taille des bundles générés est ainsi beaucoup plus petite.

4. Précautions/pièges "js" (et "ts")

La programmation en javascript (ou typescript) peut quelquefois être déroutante (non intuitive , piègeuse, ...) et essentiellement sur les points suivants :

- exécution en différé (en mode asynchrone) des callbacks
- variables javascripts référençant quelquefois des éléments "mutable" ou "immutable" .

NB :

* une valeur atomique (number , string , boolean , ...) sera re-crée en tant que nouvelle instance ailleurs en mémoire (avec nouvelle valeur) à la suite d'une demande de modification.

--> sémantique proche d'un passage par valeur (au niveau d'un paramètre de fonction)
comportement de type "copy on write" .

* un objet javascript évolué (avec sous parties) pourra généralement être modifié sans changer de position en mémoire

--> sémantique d'un véritable passage par référence (au niveau d'un paramètre de fonction)
comportement de type copie d'un "accès" partagé en mode lecture/écriture

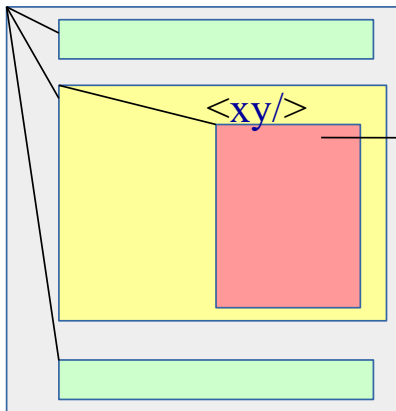
*à comprendre, expérimenter et ne pas perdre de vue ...
sinon ... beaucoup de bugs non maîtrisés ...*

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 <code>readRainbow()</code> 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> <i>possible mais très rare</i>	"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`

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

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 "angular" 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>
```

1.9. Bindings particuliers

Cas particulier **[ngValue]** pour une liste de sélection d'objet :

```
...  
<select [(ngModel)]="selectedPublication" size="8" style="width:100%"  
  (change)="onChangeSelectedPublication($event)">  
  <option *ngFor="let publication of tabPublications"  
    [ngValue]="publication" >{{essentielPublicationString(publication)}} </option>  
</select>  
...
```


V - Contrôles de formulaires , composants GUI

1. Contrôle des formulaires

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

Approches	Caractéristiques
template-driven	Simple paramétrage dans le templates HTML, pas ou très peu de code typescript/ javascript
model-driven (alias reactive-forms)	Meilleure façon de paramétrer le comportement (moins de paramétrage côté HTML) , plus de code typescript
via Form-builder API	Variante sophistiquée de model-driven / reactive-forms

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 HTML (ngForm)

```
<form #formXy="ngForm" >
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" name="name"
      [(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 , 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>**

NB : dans cet exemple *btn* , *btn-default* , *form-group* , *form-control* sont des *classes de styles "bootstrap-css"* et sont *facultatives* (pas du tout obligatoire pour les mécanismes d'angular) .

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 au cas par cas 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 .

Configuration de contrôles de saisies spécifiques:

NB : pour configurer des contrôles de saisies plus sophistiqués/spécifiques , on peut (entre autres solutions) adopter ce style de code (qui deviendra plus limpide après avoir lu le paragraphe suivant).

```
...
import { NgForm, Validators } from "@angular/forms";
...

export class XyzComponent implements OnInit {
  ...
  @ViewChild('formXy') form : NgForm ; //pour accéder/manipuler <form #formXy="ngForm"

  /*
  <form #formXy="ngForm" [hidden]="submitted"
  (mouseenter)="onFormInit()" ....>
  */
  onFormInit(){
    console.log("onFormInit() called")
    //NB: dans ngOnInit() : trop tôt , this.form.controls['...'] undefined
    // l'événement (mouseenter) peut convenir (entre autres solutions).
    this.form.controls['age'].setValidators(
      [Validators.required ,
       Validators.min(0),
       Validators.max(150)]);

    this.form.controls['email'].setValidators(
      [Validators.email]);

  }
}
```

1.3. Approche "model-driven" / "reactive-form"

dans composant angular :

```
import { FormGroup, FormControl , Validators } from '@angular/forms';
....
class ModelFormComponent implements OnInit {
  myform: FormGroup;

  ngOnInit() {
    myform = new FormGroup({
      name: new FormGroup({
        firstName: new FormControl('', Validators.required ),
        lastName: new FormControl('default_name', Validators.required),
      }),
      email: new FormControl('', [ Validators.required,
        Validators.pattern("^[ @]*@[^ @]*") ] ),
      password: new FormControl( '',[ Validators.required, Validators.minLength(8) ] ),
      language: new FormControl()
    });
  }
}
```

dans module :

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

dans template HTML :

```
<form novalidate [formGroup]="myform">
  <fieldset formGroupName="name">
    <div class="form-group">
      <label>First Name</label>
      <input type="text" class="form-control"
        formControlName="firstName" >
    </div>
    <div class="form-group">
      <label>Last Name</label>
      <input type="text" class="form-control"
        formControlName="lastName" >
    </div>
  </fieldset>
  <div class="form-group">
    <label>Email</label>
    <input type="email" class="form-control"
      formControlName="email" >
  </div>  ...
</form>
```

NB :

- novalidate (dans <form ...>) signifie pas de validation HTML5 automatiquement effectuée par le navigateur mais simplement par l'application angular .
- en mode "model-driven" / "reactive-form" , pas besoin de [(ngModel)]="xxx.yyy" mais on récupère (dans onSubmit() ou ...) les données saisies au sein de **myform.value** .

Accès aux détails d'un champ d'un formulaire contrôlé par angular :

myForm.**get**(*formControlName*).**errors** // **.dirty** , **.valid** , ...

1.4. Avec l'aide de FormBuilder

En mode *model-driven* / *reactiveForm* ,
de façon à construire plus simplement le paramétrage d'un FormGroup avec FormControl et
Validateurs imbriqués , on peut éventuellement s'appuyer sur FormBuilder :

Exemple :

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

@Component({ ... })
export class AppComponent implements OnInit {
  myForm: FormGroup;
  constructor(private _formBuilder: FormBuilder) {}

  ngOnInit() {
    this.myForm = this._formBuilder.group({
      name: ['default_name', Validators.required],
      email: [' ', [Validators.required, Validators.pattern('[a-z0-9.@]*')]],
      message: [' ', [Validators.required, Validators.minLength(15)]]
    });
  }
}
```

1.5. Exemple de Validateur personnalisé / spécifique :

url.validator.ts

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

export function ValidateUrl(control: AbstractControl) {
  if (!control.value.startsWith('https') || !control.value.includes('.io')) {
    return { invalidUrl: true }; //return { errorKeyname : true } if invalid
  }
  return null; //return null if ok (no error)
}
```

Utilisation :

```
this.myForm = this._formBuilder.group({
  userName: [' ', Validators.required],
  websiteUrl: [' ', [Validators.required, ValidateUrl ]],
});
```

VI - Components (angular)

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

1.1. Deux niveaux de modularité : fichiers/classes et modules

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

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

Dans certains cas techniques , un **module (secondaire ou annexe)** correspond à un seul fichier (comportant une décoration `@NgModule`) : module auxiliaire pour configurer le routage (`app-routing.module.ts` , ...)

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

Une grosse application peut comporter plusieurs grands modules complémentaires (ex : partie principale publique "app" , partie réservée à l'administrateur "admin" , partie/espace réservé(e) à un utilisateur connecté et ayant tel rôle .

Un module peut également permettre de rassembler un ensemble de services communs spécifiques à l'application (ex : Authentification , Session , SharedData , ...)

Finalement, un module peut rassembler un ensemble de composants réutilisables (à la manière des extensions "material" , "ionic" ou "primeNG") .

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

Selon une logique proche de celle de nodeJs , une application "Angular" 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 angular 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'**angular** est codé à l'intérieur de **librairies de composants prédéfinis regroupés en modules**.

Les modules des "**librairies**" prédéfinies d'angular 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';
...
```

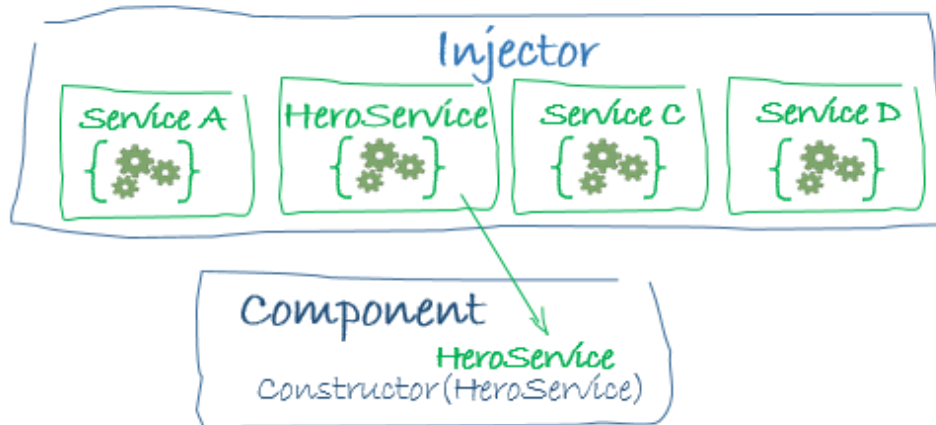
1.3. Principaux types de composants "angular":

Module fonctionnel (répertoire)	Ensemble de composants et/ou de services (généralement placés dans un même répertoire) et chapeauté par une classe décorée via @NgModule .
Module auxiliaire (paramétrages / config.)	Simple fichier auxiliaire de paramétrage (ex : app-routing.module.ts) regroupant certains éléments de configuration.
Component (composant visuel)	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 (invisible , potentiellement partagé)	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 .

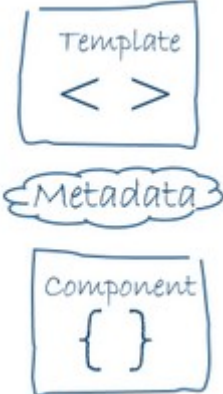
La **classe principale** d'un module Angular doit être (par convention) placée dans un fichier `xyz/xyz.module.ts` où `xyz` est le **nom du module** (ex : `"app"`).

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 : @Component). <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" (*ngFor , { { } }) .</p>
---	---

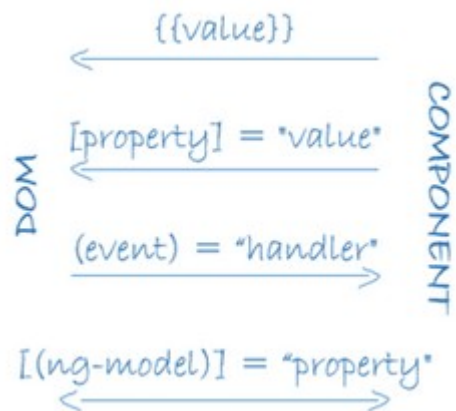
Les liaisons/correspondances gérées par le framework angular2 entre les éléments du composant "orienté objet" et les éléments de la vue "web/HTML/DOM" issue du template sont appelées "**data binding**".

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;
  }
}
```

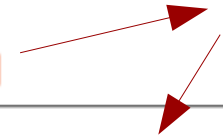
MyHeader titre1 .. **

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 event

3.3. Projection des éléments imbriqués

Un composant angular peut (en tant que nouvelle balise) , incorporer à son tour certains sous éléments (ex : <div ...> ou autre sous-sous composant angular).

Au sein du template HTML d'un composant le (ou le paquet de) sous-composant(s)/sous-balises sera vu via la balise spéciale <ng-content></ng-content> .

Cette fonctionnalité était appelée "transclusion" au sein des directives "angular Js 1.x" , elle est maintenant appelée "projection" au sein des composants angular 2+ .

Exemple concret : composant réutilisable "togglePanel" basé sur des styles et fontes *bootstrap-css* :

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'toggle-panel',
  templateUrl: './toggle-panel.component.html',
  styleUrls: ['./toggle-panel.component.css']
})
export class TogglePanelComponent {
  toggleP : boolean =false;
  @Input()
  title : string = 'default panel title';
  constructor() { }
}
```

```
<div class="card">
  <!-- <div class="card-header bg-info" -->
  <h4 class="card-header my-bg-primary">
    <a class="text-light" (click)="toggleP = !toggleP" >{{title}}
      <span class="fa" [class.fa-chevron-circle-down]="!toggleP"
        [class.fa-chevron-circle-up]="toggleP"></span>
    </a></h4>
  <!-- </div> -->
  <div class="card-body collapse " [class.show]="toggleP">
    <ng-content></ng-content>
  </div>
</div>
```

Exemple d'utilisation :

```
<toggle-panel [title]=" 'panel1' " >
  <app-part1></app-part1> <!-- ou ... , vu comme ng-content dans toggle-panel -->
</toggle-panel>

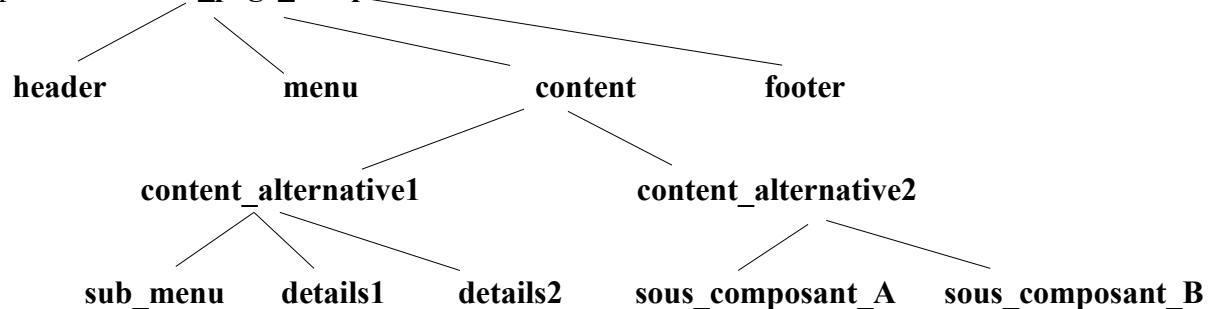
<toggle-panel [title]=" 'panel2' " >
  <div>contenu du panneau 2</div> <!-- vu comme ng-content dans toggle-panel....html -->
</toggle-panel>
```

<div>part 1 (basic.module) ▼</div>	<div>part 1 (basic.module) ▲</div> <div> binding @Input/@Output forms pipes service </div> <div> raw date: Sun Jan 21 2018 20:34:08 GMT+0100 date with date : Jan 21, 2018 </div>
------------------------------------	--

3.4. 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 (*finalment très classique pour une technologie d'interface graphique – comme DOM , comme JSF*) **permet au framework angular 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 binding" (exemple : "propriété initialisée selon niveau parent")
OnInit	ngOnInit()	A l'initialisation du composant et après le premier éventuel ngOnChanges() et après constructeur et injections
DoCheck	ngDoCheck()	Détection d'un changement personnalisé (selon développeur)
AfterContentInit	ngAfterContentInit()	Après que le contenu d'un composant soit initialisé en mémoire (après ngOnInit() et avant ngAfterViewInit())
AfterContentChecked	ngAfterContentChecked()	Après chaque vérification du contenu d'un composant
AfterViewInit	ngAfterViewInit()	Après que la vue associée à un composant soit (ré-)initialisée. (cas rare et après coup donc assez peu intéressant)
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>
  `
})
export class ListeComptesComponent {
  // ...
}
```

```

        </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
        }
    }

```

VII - 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 (*indirectement via ajax/http et ws rest/json*)
- des mises à jours de données (*indirectement via ajax/http et ws rest/json*)
- données partagées (par plusieurs composants) en mémoire *dans l'appli angular (coté navigateur)*
- gestion de la session utilisateur (authentification , ...)
- traitements réutilisables (calculs , traductions , ...)
- ...

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 Observable.of(this.sampleComptes); //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 } ];
}
```

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" .

Angular 2+ 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. Très 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; }
}
...
```

VIII - Switch et routing (navigations)

1. Navigation via ngSwitch et router de angular

1.1. Paramétrage dans index.html

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

1.2. Switch (local) de sous-templates (sous-composants):

```
<div [ngSwitch]="variableXy">
  <composant1 *ngSwitchCase="'case1Exp'">...</composant1>
  <composant2 *ngSwitchCase="'case2LiteralString'">...</composant2>
  <div_ou_composant3 *ngSwitchDefault>...</div_ou_composant3>
</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 parent principal . Pas de changement d'URL .

Attention : Le switch/basculement de composant s'effectue par remplacement d'instance (et éventuelle perte de l'état de l'ancien composant remplacé) .

--> pas de show/hide ni display none/block mais rechargement complet d'un nouveau composant !!!

1.3. pseudo switch visuel (en apparence)

On peut éventuellement se bricoler facilement un "*pseudo switch visuel*" en jouant sur le style *display* de plusieurs `<div ...>` dont une seule est active à l'instant *t* :

```
<div class="panel panel-info" >
  <div class="panel-body" [style.display]="subpart=='c1'?'block':'none'" >
    <composant1></composant1>
  </div>
  <div class="panel-body" [style.display]="subpart=='c2'?'block':'none'" >
    <composant2></composant2>
  </div>
  ...
</div>
```

--> dans ce cas l'instance développée ou pas (visible ou pas) d'un sous composant est conservée (ainsi que l'état de ses variables d'instance) .

1.4. 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'angular 4+ 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 , HttpClientModule,
    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';
...
const routes: Routes = [
  { path: 'welcome', component: WelcomeComponent },
  { path: '', redirectTo: '/welcome', pathMatch: 'full'},
  { path: 'identification', component: IdentificationComponent },
  { path: 'clientIdentification/:clientId', component: IdentificationComponent }
];
@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 , HttpClientModule, AppRoutingModule , ... ],
  ...})
export class AppModule { }
```

1.5. 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" >
        <router-outlet></router-outlet>
      </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.

1.6. 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>`).

1.7. Déclenchement d'une navigation avec ou sans paramètre

```
// dans app-routing.module.ts
const routes: Routes = [...
  { path: 'xx', component: XxComponent }
  { 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]; //ou link = ['/xx'] ; sans paramètre
    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>
```

```
<a [routerLink]='["/xx"]' ...> vers xx </a>
```

1.8. 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 = "...";
  yyId: number ;
  y : Yy;
  constructor(private _yyService : YyService,
               private _route: ActivatedRoute,
               private _location: Location){
  }
  ngOnInit() {
    this._route.params.subscribe(
      (params: Params) => {
        this.yyId = 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.yyId).subscribe(yy=>this.y = yy ,
                                                         error => this.message = <any>error);
  }
  goBack(): void { this._location.back(); /* retour arrière dans historique des navigations */ }
}
```


Remarque très importante (pour comprendre et ne pas devenir fou) :

- (Cas "a") Suite à la série de navigation suivante :
aaa/yyy , **aaa/xxx** , **aaa/yyy** , des composants des classes Yyy , Xxx et Yxx seront ré-instanciés à chaque changement d'URL (et l'état des variables d'instance sera perdu) .
- (Cas "b") Suite à la série de navigation suivante :
bbb/detail_produit/1 , **bbb/detail_produit/2** , **bbb/detail_produit/3** (où seule change la valeur du paramètre en fin d'url) , l'instance de la classe DetailProduit est conservée et la *callback enregistrée (via subscribe) sur this._route.params est automatiquement ré-appelée pour prendre en compte le changement de numéro de produit à détailler* .

NB :

Dans le cas "a" , où les instances ne sont pas conservées , on peut simplifier l'unique récupération d'un paramètre en fin de route activée via la notion de snapshot (valeurs à l'instant t):

Exemple :

```
class MyComponent {
  constructor(route: ActivatedRoute) {
    const yyId: number = Number(route.snapshot.params['yyId']);
    ...
  }
}
```

1.9. Sous niveau de routage (children)

Une application de grande taille peut comporter plusieurs niveaux de composants et sous composants.

Un <router-outlet> dans un composant main permet par exemple de switcher de sous composants Xxx , Yyy , Zzz et un de ces sous composants (par exemple Yyy) peut à son tour comporter une balise <router-outlet> pour switcher de sous-sous-composants .

Exemple d'url avec sous niveaux :

```
xxx/
yyy/1
yyy/1/details_prod/3
yyy/1/details_prod/4
zzz/
```

yyy sera intitulé *subspace-main* (sous espace principal) dans l'exemple qui suit .

Exemple de configuration de routes avec sous niveau:

```

...
const routes: Routes = [
  { path: 'welcome', component: WelcomeComponent },
  { path: '', redirectTo: '/welcome', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'subspace-main/:id', component: SubspaceMainComponent
  children: [
    { path: 'prodList', component: ProdListComponent },
    { path: 'details/:num', component: ProdDetailComponent },
    { path: '', redirectTo: 'prodList', pathMatch: 'prefix' }
  ]
}
];

```

1.10. Routes conditionnées (via gardiens)

Il est possible de créer (et enregistrer dans un module) des services techniques (ex : CanActivateRouteGuard) et implémentant une interface "Can...." (ex : CanActivate) pour contrôler si une route peut ou pas être activée selon (par exemple) une authentification utilisateur effectuée ou pas :

```

import { Injectable } from '@angular/core';
import { CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot } from '@angular/router';

import { AuthService } from './auth.service';

@Injectable()
export class CanActivateRouteGuard implements CanActivate {
  constructor(private auth: AuthService) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    return this.auth.isUserAuthenticated();
  }
}

```

Dans la définition des routes , on pourra déclarer une liste de gardiens à utiliser :

```

...
{ path: 'dashboard',
  component: DashboardComponent,
  canActivate: [CanActivateRouteGuard]
}, ...

```

2. Lazy loading

Au lieu de charger dès le démarrage (dans le navigateur) tous les composants de l'application (ce qui peut quelquefois être long/lent) , on peut organiser la structure du code en différents modules qui ne seront chargés (en différé) que lors d'une navigation (ex : activation d'un lien hypertexte associé au routage).

2.1. Exemple du (sous-)module en mode lazy :

```
ng g module myLazy
cd src/app/my-lazy
ng g component xx
ng g component yy
```

my-lazy.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
...
import { Routes, RouterModule } from '@angular/router';
const childRoutes: Routes = [
  { path: 'xx', component: XxComponent },
  { path: 'yy', component: YyComponent },
  { path: '', redirectTo: 'xx', pathMatch: 'prefix' }
];
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(childRoutes)
  ],
  declarations: [ XxComponent, YyComponent ]
})
export class MyLazyModule { }
```

Et part exemple pour naviguer relativement de xx vers yy (et vice versa) :

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

2.2. Référencement d'un sous module en mode lazy

Référencement du ou des sous module(s) "lazy" depuis le module principal de l'application :

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
const routes: Routes = [
  { path: 'mylazy-part1', loadChildren: './my-lazy/my-lazy.module#MyLazyModule' },
  { path: 'mylazy-part2', loadChildren: './my-lazy/my-lazy2.module#MyLazy2Module' }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Point clef: pas d'import explicite de sous module mais chemin **loadChildren** exprimé avec un suffixe en **"#LazySubModuleName"** .

Dans un des composants du module principal :

```
<a [routerLink]="['/mylazy-part1']"> lazy loading part 1 ... </a> <br/>
```

Pour visualiser l'effet lazy-loading on peut éventuellement surveiller le **chargement différé** (au moment de la navigation) de ***-chunk.js** dans la partie **"network"** de **"Chrome Dev Tools / F12"**.

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

1. Angular et dialogues HTTP/REST

1.1. Contexte des appels HTTP/REST et CORS

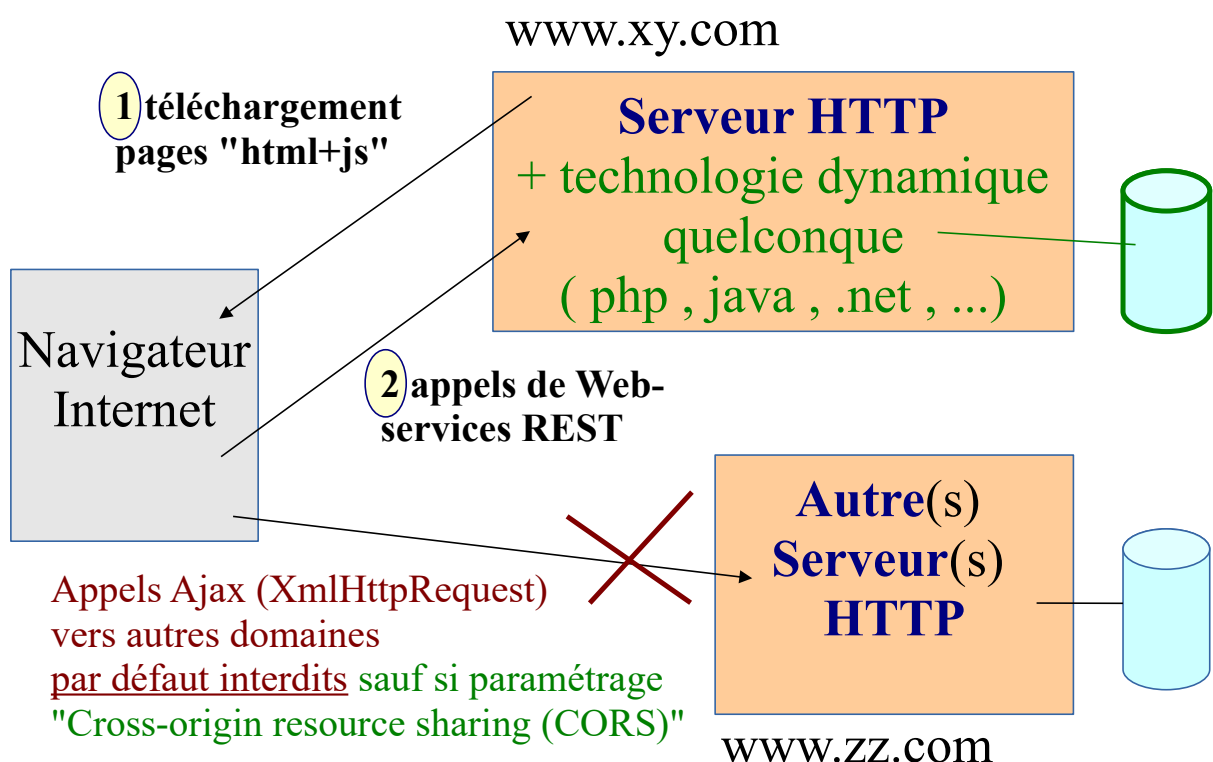
Une application Angular s'exécute au sein d'un navigateur Web . Lorsque celle-ci doit effectuer un appel HTTP/ajax , elle est soumise aux restrictions habituelles du navigateur :

- appels en file:/ pas toujours autorisés (ok avec firefox , par défaut refusés avec chrome)
- **les URLs des WS-REST appelés doivent commencer par le même nom de domaine que celui qui a servi à télécharger index.html** (ex : `http://localhost:4200` ou `http://www.xyz.com` ou ...). Dans le cas contraire des autorisations "CORS" sont nécessaire du coté serveur (java ou php ou nodeJs/express ou ...) .

Rappel important : si les appels HTTP/ajax sont effectués vers un autre nom de domaine il faudra prévoir des autorisations "CORS" du coté des web-services REST coté 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();
});
```

1.2. Configuration d'un reverse-proxy http

De façon à écrire et tester du code simple , stable et portable (de la phase de développement/test jusqu'à la phase de production) , on aura intérêt à :

- **utiliser des URLs relatives**
- **configurer un reverse proxy http** (option *--proxy-conf* de *ng serve* développée dans le chapitre "packaging / déploiement")

1.3. Promise ou Observable ?

Les habitués de AngularJs (1.x) connaissent bien les "Promise" . Celles-ci sont encore utilisables avec Angular 2 , 4 ou 5 (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 Angular 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()** .

1.4. Simulation d'invocation de WS REST via of() de rxjs

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

D'autres solutions sont disponibles pour simuler des appels HTTP ... (voir chapitre/annexe sur test unitaire)

1.5. Ancien service Http de Angular 2 à 4.2

NB : de la version 2.0 à 4.2 le framework angular ne proposait que le service Http pour appeler des Web Services REST .

Depuis la version 4.3 le nouveau service HttpClient rend obsolète l'ancien service Http .
D'autre part , les nouveautés syntaxiques apportées par RxJs 6 et Angular 6 augmente encore l'écart entre ancien et nouveau style concernant les appels http .

L'ancienne façon d'appeler les WS-REST via Http (et pas HttpClient) a maintenant été déplacé dans une annexe de ce support de cours .

2. Nouvelle Api HttpClient (depuis Angular 4.3)

La nouvelle api **HttpClient** de la partie `@angular/common/http` de **Angular** `>= 4.3` est une **version améliorée** (avec meilleur typage , généricité , code d'utilisation plus compacte, ...) du service technique **Http** disponible depuis Angular 2 .

Pour utiliser HttpClient à la place de Http , il faut commencer par importer le module technique **"HttpClientModule"** à la place de l'ancien "HttpModule" dans un module fonctionnel de l'application (ex : app.module.ts) :

```
import { HttpClientModule } from '@angular/common/http';
//à la place de import { HttpModule } from "@angular/http";
...

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

Utilisation du service technique HttpClient dans un service :

(ou éventuellement directement depuis un composant) :

```
import { HttpClient } from '@angular/common/http';
// à la place de l'ancien import { Http } from '@angular/http';

...
constructor(private http: HttpClient){ } //injection de dépendance
...
```

//utilisation rare (non typée) / ici directement depuis un composant :

```
ngOnInit(): void {
  this.http.get('https://api.github.com/users/didier-tp')
    .subscribe(data => { console.log(data); });
}
```

==> En mode "HttpClient" l'appel à `http.get()` retourne directement les données de la réponse Http au format `Observable<any>` et non pas un `Observable<Response>` brute à analyser/décortiquer .

Autrement dit , les anciennes lignes de code

```
.map(response => <Client[]> response.json() )  
.catch(e => { return Observable.throw('error:' + e);});
```

qui suivaient aussitôt les anciens appels à `http.get()` ne sont donc plus systématiquement nécessaires .

Récupération fine des causes d'erreurs via second paramètre facultatif de subscribe :

```
this.http.get('https://api.github.com/users/didier-tp')  
  .subscribe(  
    data => { console.log(data); } ,  
    (err: HttpErrorResponse) => {  
      if (err.error instanceof Error) {  
        console.log("Client-side error occured.");  
      } else {  
        console.log("Server-side error occured.");  
      }  
    }  
  );
```

Appels typés :

Plus rigoureusement , un appel à `http.get<DataClass>(url , ...)` retourne des données au format `Observable<DataClass>` plutôt qu'au format `Observable<any>`

Exemple :

```
public getTabInscriptionsObservable() : Observable< Client[] > {  
  let inscriptionUrl : string = null;  
  inscriptionUrl = this._inscriptionUrlBase ;  
  console.log( "inscriptionUrl = " + inscriptionUrl);  
  return this.http.get<Client[]>(inscriptionUrl );  
}
```

Appels typés en mode post avec header http personnalisé :

```
import { HttpClient , HttpHeaders } from "@angular/common/http";  
import { Observable } from "rxjs/Observable";  
import { Client } from "../client";  
  
@Injectable()  
export class InscriptionService {  
  constructor(private _http : HttpClient) { }  
  
  private _headers = new HttpHeaders({'Content-Type': 'application/json'});  
  
  public postInscriptionsObservable(cli : Client):Observable<Client> {
```



```

let inscriptionUrl : string = './tp/inscription' ; //avec ng serve --proxy-config proxy.conf.json
return this._http.post<Client>(inscriptionUrl ,
                             cli,
                             {headers: this._headers} );
} ...

```

Autre nouveauté annexe de la version 4.3 , les intercepteurs :

```

import { Injectable } from '@angular/core';
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest }
                                from '@angular/common/http';
import { Observable } from 'rxjs/observable';

@Injectable()
export class MyAuthInterceptor implements HttpInterceptor {
  intercept (req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    let token = localStorage.getItem('token');
    const authReq = req.clone({
      headers: req.headers.set('Authorization', 'Bearer ' + token)
    });
    return next.handle(authReq);
  }
}

```

avec la déclaration suivante dans app.module.ts :

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { HTTP_INTERCEPTORS } from '@angular/common/http';

import { AppComponent } from './app.component';
import { MyAuthInterceptor } from './myauth.interceptor';

@NgModule({
  declarations: [    AppComponent    ],
  imports: [
    BrowserModule,    HttpClientModule    ],
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: MyAuthInterceptor,
    multi: true
  }],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

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';

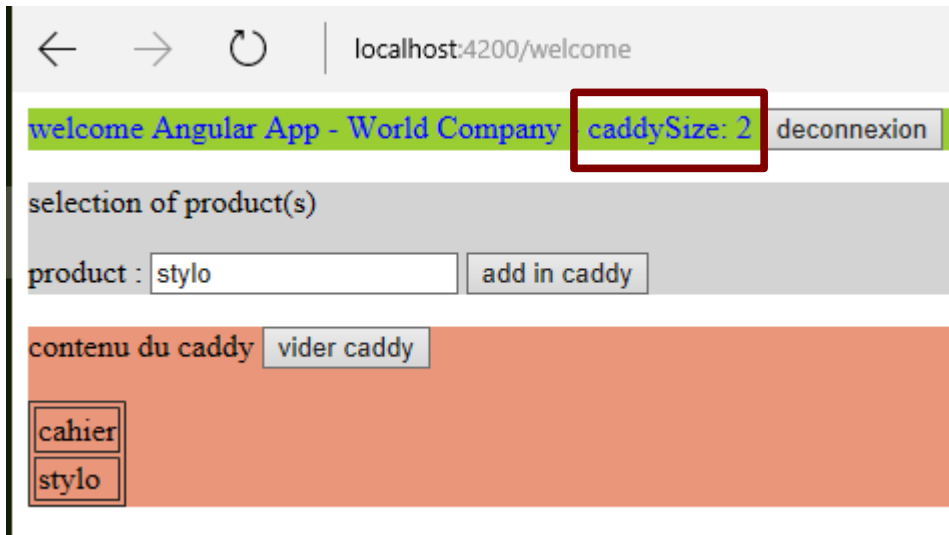
@Injectable()
export class CaddyService {
  private _compteur : number = 0;
  private _caddyContent : string[] = [];

  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 , ...

console.log("...") standard de javascript

XI - Packaging et déploiement d'appli. angular

1. déploiement avec ou sans "bundle"

1.1. Rappel du contexte d'un développement angular

- Le code source d'une application angular est en typescript (.ts) , lui même une évolution de es2015 .
- Les modules d'angular sont basés sur la technologie des modules es2015 (syntaxiquement standardisé , mots clef "import {...} from '...'" et export pour la gestion statique des modules) .
- Le code compilé/transpilé doit idéalement être généré au format .js (es5) de façon à être interprété de façon compréhensible par presque tous les navigateurs.
- Le développement d'une application "Angular+ " est basée sur npm et 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 ou +" des librairies d'angular et RxJs dans le répertoire node-modules récupéré par npm..

1.2. Déploiement (théorique ou rare) sans bundle

Il serait possible d'**extraire (par copie) les librairies fondamentales de angular 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 "angular" et son contenu pourra être recopié "tel quel" vers n'importe quel serveur HTTP (ex : htdocs de apache2) . On pourra également recopier cet ensemble de fichiers "angular" 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.

--> pas rapide , pas performant !!!!

Voici la liste des principales 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',
```

1.3. 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.

2. Angular-CLI en mode développement

L'essentiel de @angular/cli a déjà été présenté en début du cours.

Rappel des principales lignes de commandes de **ng** (@angular/cli) :

ng new my-app , cd my-app	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 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)

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) .

2.1. proxy http pour appels ajax/WS-REST en mode dev

Il serait possible (durant la phase de développement) de :

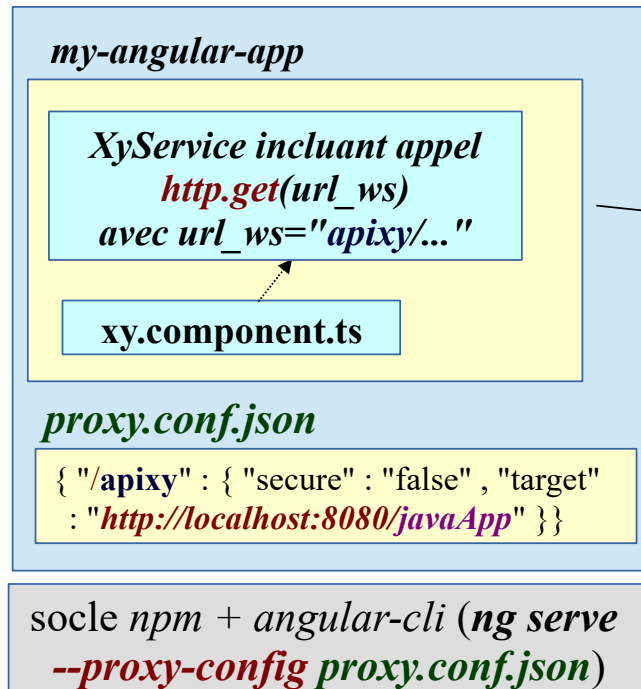
- utiliser des URLs absolues (ex : <http://localhost:8282/xxx/yyy>) pour appeler des Web services "REST" (java ou php ou nodeJs/express ou ...) via angular et ajax
- paramétrer des autorisations "CORS" du coté serveur (code java ou js/express ou ...)
- configurer des switchs d'URL coté angular

De façon à éviter toutes ces choses (aujourd'hui déconseillées) , on peut :

- toujours utiliser des URLs relatives (ex : xxx/yyy) pour appeler des Web services "REST" (java ou php ou nodeJs/express ou ...) via angular et ajax dès la phase de développement
- ne pas systématiquement avoir besoin de configurer des autorisations "CORS" du côté serveur
- configurer un proxy http au sein d'angular-CLI (uniquement exploitable avec ng serve).

Environnement de développement Angular (v2,v4,...)

partie cliente "Angular"
(<http://localhost:4200/...>)

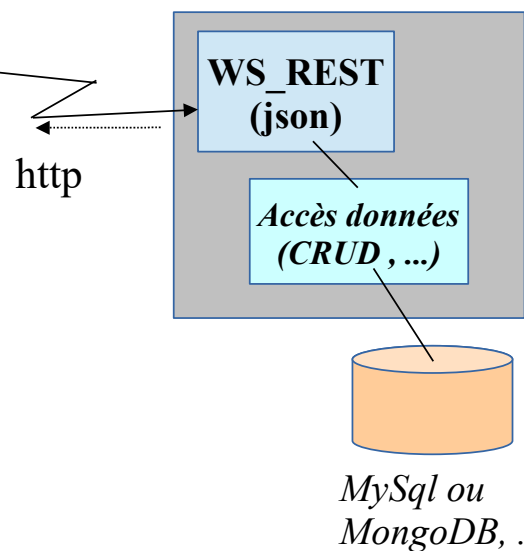


Partie "back-end" / WS-REST

<http://localhost:8080/javaApp/apixy>

ou bien

<http://localhost:8082/nodeApp/apixy>



ng serve --proxy-config proxy.conf.json

avec **proxy.conf.json**

```
{ "/apixy" : { "secure" : "false" ,  
              "target" : "http://localhost:8080/javaApp" }  
}
```

pour que les requêtes d'urls relatives **apixy/..** soient redirigées vers une application java/tomcat.

ou bien

avec **proxy.conf.json**

```
{ "/apixy" : { "secure" : "false" ,  
              "target" : "http://localhost:8282/nodeApp" }  
}
```

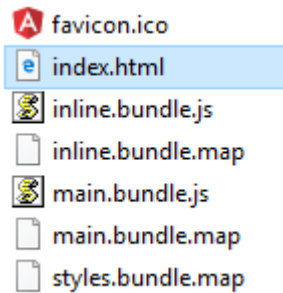
pour que les requêtes d'urls relatives **apixy/..** soient redirigées vers une appli. nodeJs/express .

Ne fonctionnant qu'en mode "développement" (via *ng serve*) , certains équivalents de *-proxy-config* en mode "production" seront exposés au sein des paragraphes ci-après.

3. Mise en production d'une application angular

ng build (et **ng build --prod**) génère des fichiers dans le répertoire **my-app/dist** .

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



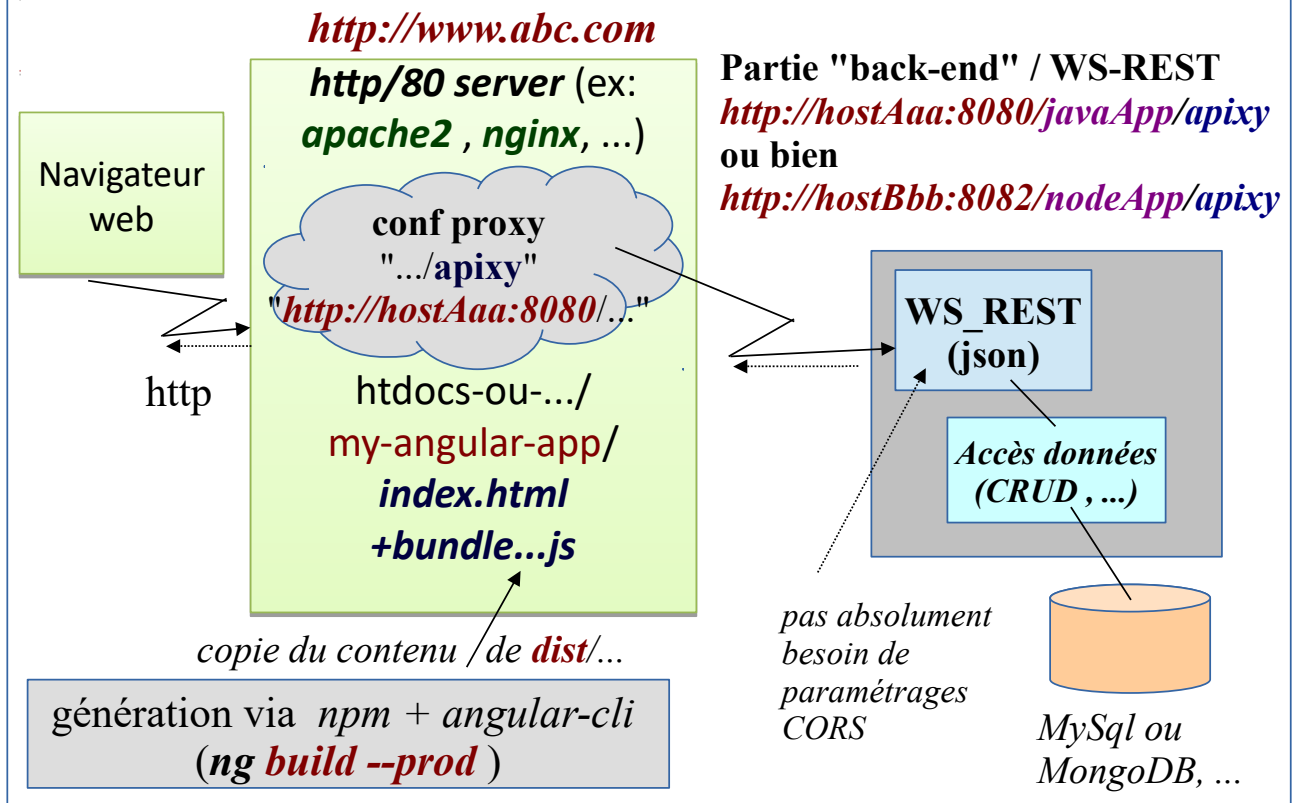
ng build --prod est quelquefois accompagné de quelques "bugs" avec certaines anciennes versions de angular-cli .

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

NB :

- le code généré dans le répertoire **dist** ne fonctionne qu'avec un accès "**http**" (pas **file:**) .
- il est possible de recopier le code du répertoire "**dist**" vers le répertoire d'une application simple **nodeJs/express** pour effectuer une sorte de **mixage compatible** (code **nodeJs/express** pour **WS-REST** et code "angular" recopié dans sous répertoire "**front-end**" servi statiquement par **nodeJs/express** via **app.use(express.static('front-end'))** ;
- Une mise en production évoluée passera souvent par l'utilisateur d'un véritable serveur **http** (tel que **apache 2.x** ou **nginx**) . On pourra déposer le code "static" angular à cet endroit et configurer des "reverse-proxy" vers des **WS-REST** java ou php ou **nodeJs/express** .

Environnement de prod compatible Angular (v2,v4,...)



3.1. Configuration nginx pour angular et redirection WS-REST .

Le démarrage de **nginx** sous windows s'effectue sans aucune option (*double click sur nginx.exe*)

L'arrêt du serveur s'effectue via **nginx -s stop**

répertoire par défaut des pages statiques : **html** (ou l'on peut placer un sous répertoire my-angular-app)

configuration par défaut (sous windows) : **conf/nginx.conf** (avec copie de sauvegarde conseillée dans *original.nginx.conf.txt*)

Configuration importante au sein de **conf/nginx.conf** :

```
server {
    listen    80;
    server_name localhost;

    # NB1 : dans nginx.conf, l'ordre des règles est important
    # et selon ~ ou = ou ^~ les autres règles sont utilisées ou pas
    # NB2 : les "location" sont exprimés avec des expressions
    # régulières ^AuDebut , aLaFin$ , contenu de() récupéré par $1,...
```

```
# REMARQUE IMPORTANTE pour APPLI ANGULAR:
# on peut déposer le code d'une appli angular (contenu du répertoire "dist")
# dans un sous répertoire "my-angular-app" ou "appxy" de nginx/html
# que si la valeur de <base href="/"> est ajustée en <base href="/appxy/"> ou ...

# proxy mvc/api part of my-java-app to tomcat on 127.0.0.1:8080
# virtualy seen as a "api" subpart of /my-angular-app (in html)

location ~ ^/my-angular-app/api/(.*){
    proxy_pass http://127.0.0.1:8080/my-java-app/mvc/api/$1?$args;
}

# proxy minibank api part of nodeJs app to 127.0.0.1:8282
# virtualy seen as a "minibank" subpart of /appxy (in html)

location ~ ^/appxy/minibank/(.*){
    proxy_pass http://127.0.0.1:8282/minibank/$1?$args;
}

location / {
    root html;
    index index.html index.htm;
}
}
```

Attention, Attention : il faut absolument placer 127.0.0.1 (ou autre ip ou) dans la config mais pas localhost pour que ça fonctionne partout (sous windows , sous linux, ...)!!!!

XII - 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. 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 :

...

1.3. 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",
    "pre2e": "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();
```

1.4. 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.

1.5. 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);
  });
});
```

1.6. 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.

1.7. 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 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();  
  });  
});
```

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(...))* .

1.8. 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();

```

1.9. 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 { HttpClientModule } 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.

XIII - Sécurité – application "Angular2"

1. Sécurisation d'une application "angular"

1.1. Quelques considérations générales sur la sécurité

Même transformé de ".ts" en ".js", une application Angular n'est pas véritablement compilée, son code source est accessible et éventuellement sujet à interception / modification.

--> jamais d'éléments confidentiels dans le code de l'application (pas de "salt", pas de "default_password", ...)

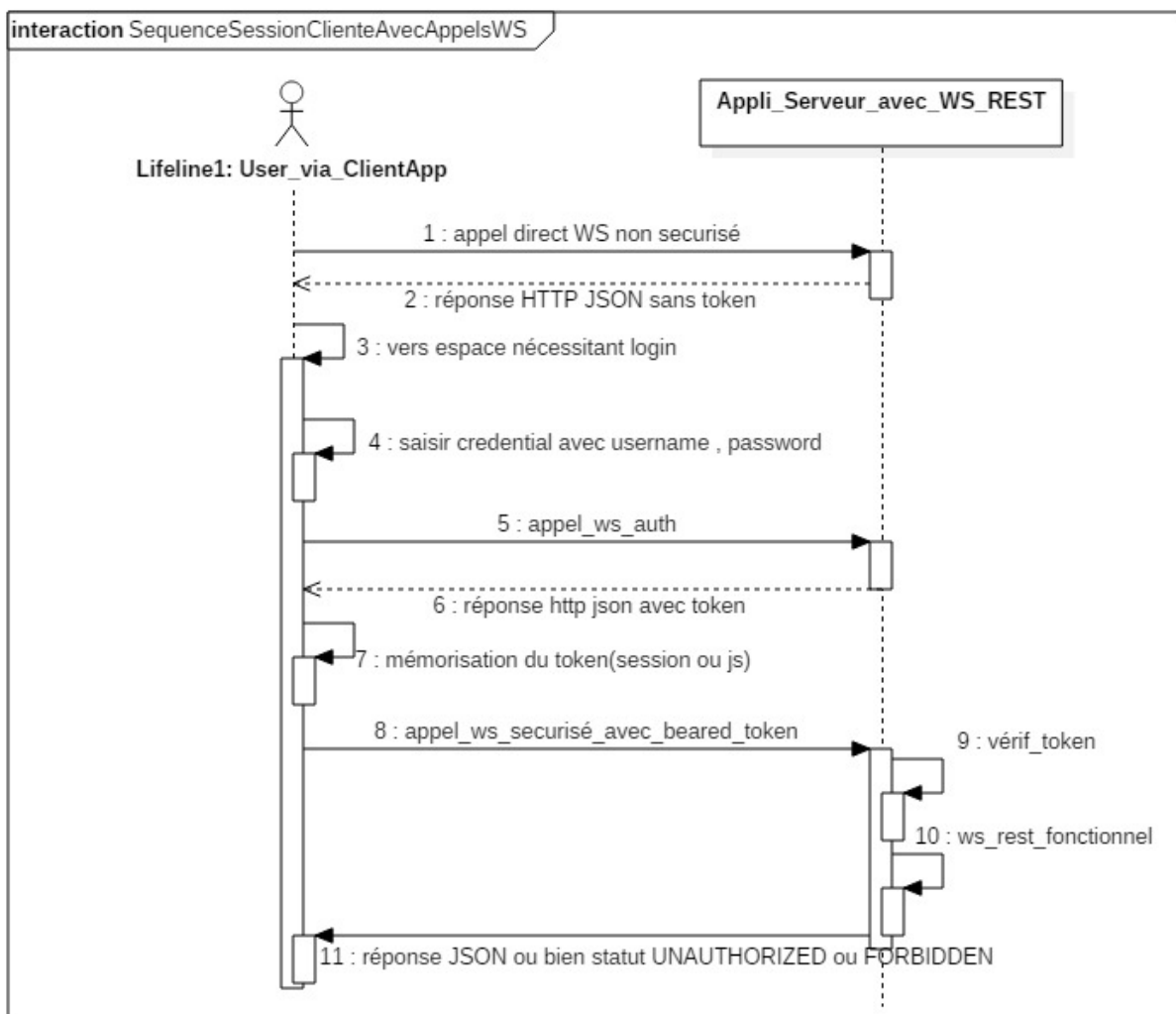
--> le côté serveur ne doit avoir qu'une confiance limitée aux requêtes angular.
Il est bon de vérifier fréquemment "token" et autres.

1.2. Conseils sur la structure d'une application angular sécurisée

- HTTPS / SSL dès qu'il faut échanger des informations confidentielles (username, password), ...
- Un service d'authentification
- Des gardiens pour certaines routes

2. Sécurisation des appels aux Web-services REST

2.1. Pseudo session avec "token" plutôt que cookie :



Une application cliente qui appelle une série de WS-REST peut être développée avec des technologies très diverses :

- java standalone (swing, java-fx ,)
- java/jee (JSF+....) ou (Spring-web-mvc +)
- HTML + js (jquery ou angular ou react ou) au sein d'un navigateur avec appels ajax
- PHP , C++ , .net/C# , ...

De même , l'application serveur qui gère le WS-REST ne gère pas systématiquement une session HTTP.

Des jetons de sécurité ("token") sont généralement employés pour gérer l'authentification d'un utilisateur et d'une application dans le cadre d'une communication sous forme de WS-REST.

Le jeton de sécurité est généré si le couple (username,password) transmis est correctement vérifié côté serveur.

Ce "token" (véhiculé au format "string") pourra prendre la forme d'un uuid (universal unique id , exemple: e51cd176-a522-454c-9c0a-36ca74cdb2d0) ou bien être conforme au format JWT (Json Web Token).

Dans le cas d'un token de type uuid , le coté serveur doit maintenir une liste ou une map des "tokens générés et valides" (éventuellement associés à certaines infos (username, role ,).

Dans le cas d'un token sophistiqué de type jwt , le token généré comporte déjà en lui (de manière cryptée/extractible) certaines informations utiles (subject , roles ,) et donc pas besoin de map côté serveur.

Le protocole HTTP a normalisé la façon dont le token doit être retransmis au sein des requêtes émises du client vers le serveur (après l'authentification préalablement effectuée) :

Il faut pour cela utiliser le champ "Authorization :" de l'entête HTTP pas en mode "Basic" mais en mode "Bearer" (signifiant "au porteur" en français).

exemple (postman) :

Example (postman):

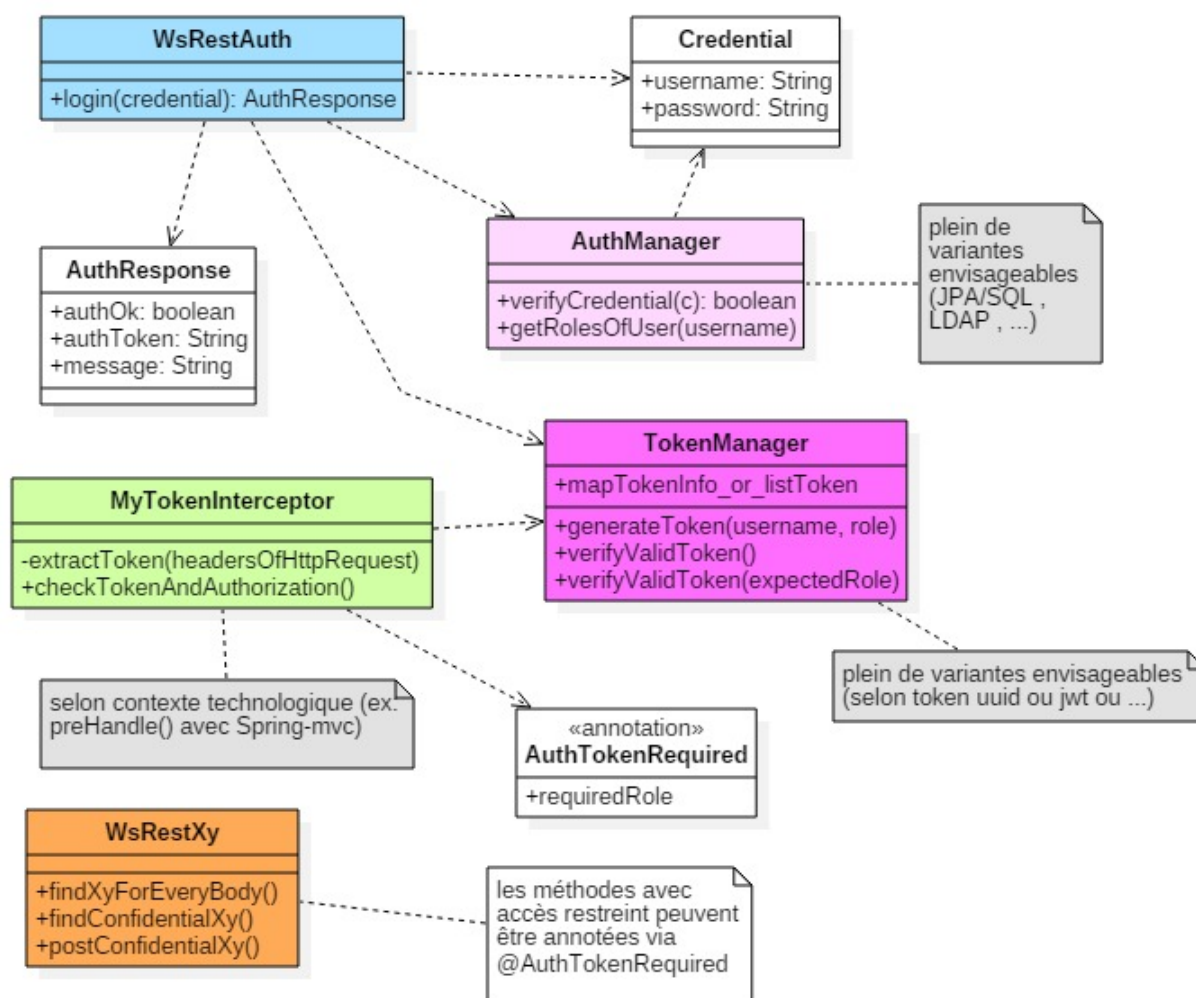
Authorization ●	Headers (1)	Body	Pre-request Script	Tests						
	<table><tr><th>Key</th><th>Value</th></tr><tr><td>Authorization</td><td>Bearer e51cd176-a522-454c-9c0a-36ca74cdb2d0</td></tr><tr><td>New key</td><td>Value</td></tr></table>	Key	Value	Authorization	Bearer e51cd176-a522-454c-9c0a-36ca74cdb2d0	New key	Value			
Key	Value									
Authorization	Bearer e51cd176-a522-454c-9c0a-36ca74cdb2d0									
New key	Value									

exemple (javascript / jquery) :

```
<script src="jquery-3.2.1.js"></script>
<script>
    var authToken; //token d'authentification (en mode bearer) à retransmettre
    ...

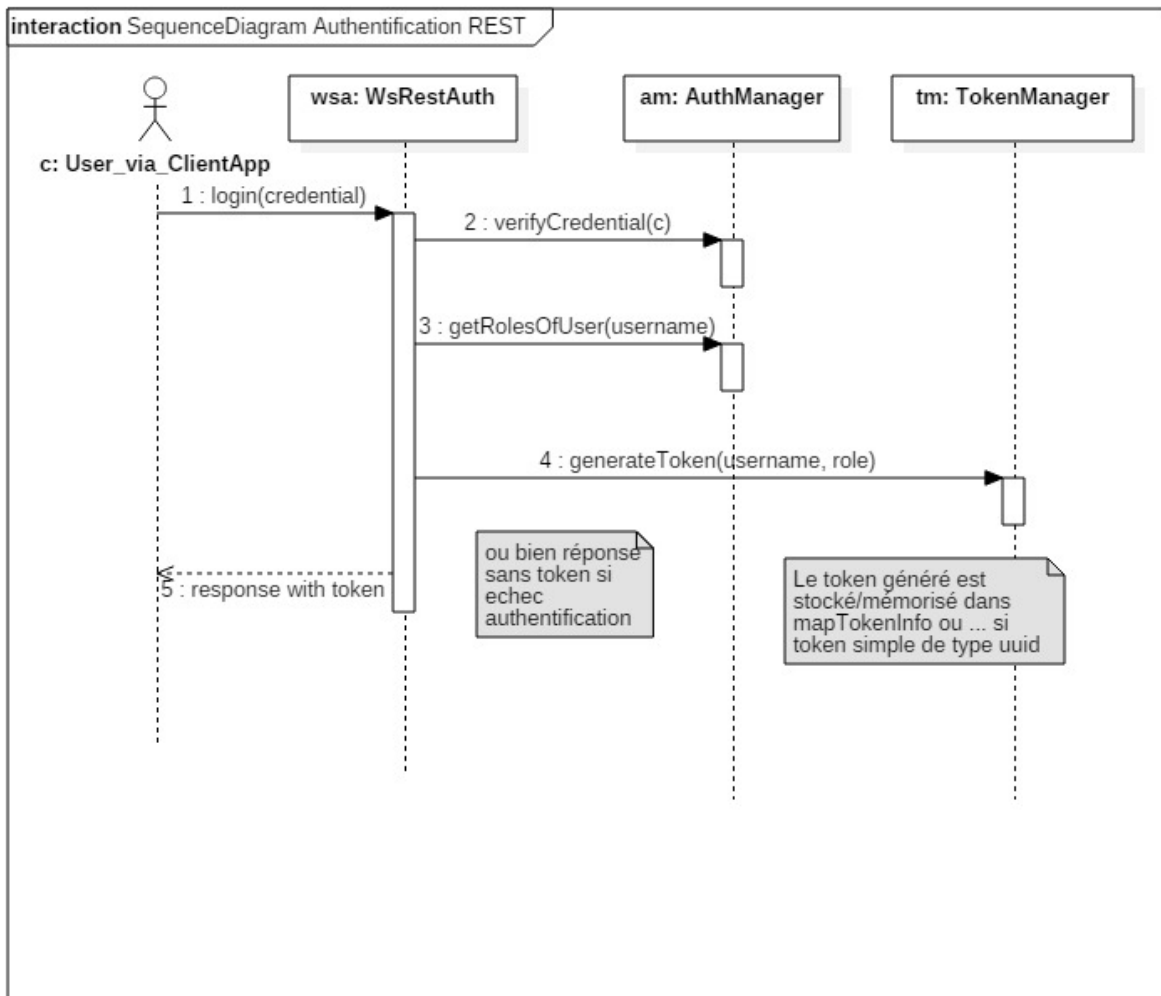
    $(document).ajaxSend(function(e, xhr, options) {
        //retransmission du jeton d'authentification
        //dans l'entête http de la requete ajax
        //xhr = XmlHttpRequest = objet technique du navigateur
        //qui déclenche les requêtes ajax
        xhr.setRequestHeader('Authorization','Bearer '+ authToken);
    });
    $.ajax({
        type: "GET",
        url: "ws/rest/confidential/news",
        contentType : "application/json" ,
        success: function (response) {
            if (response) {
                $("#spanMsg").html(JSON.stringify(response));
            }
        }
    });
    ...
```

2.2. Responsabilités techniques coté serveur :



<i>Composants</i>	<i>Responsabilités techniques</i>
AuthManager (gestionnaire d'authentification)	vérifier login/credential via dataBase ou autre
TokenManager (gestionnaire de "token")	Gérer (générer, vérifier, ...) une sorte de jeton (uuid, jwt, ...)
WsRestAuth (ws de login/authentification)	WS REST vérifiant login/credential et retournant token dans message de réponse global (ex : AuthResponse retourné au format JSON)
MyTokenInterceptor	Intercepteur technique (selon techno : Spring-mvc ou jax-rs ou ...) permettant de vérifier la validité du jeton véhiculée par une requête.
WsRestXy	WS REST fonctionnel avec partie en accès restreint annotée via @AuthTokenRequired

2.3. Service d'authentification / génération token



exemple (postman) :

POST	http://localhost:8080/serverSpringMvc/ws/rest/auth/verifAuth	
Authorization	Headers (1)	Body
	Key	Value
<input checked="" type="checkbox"/>	Content-Type	application/json

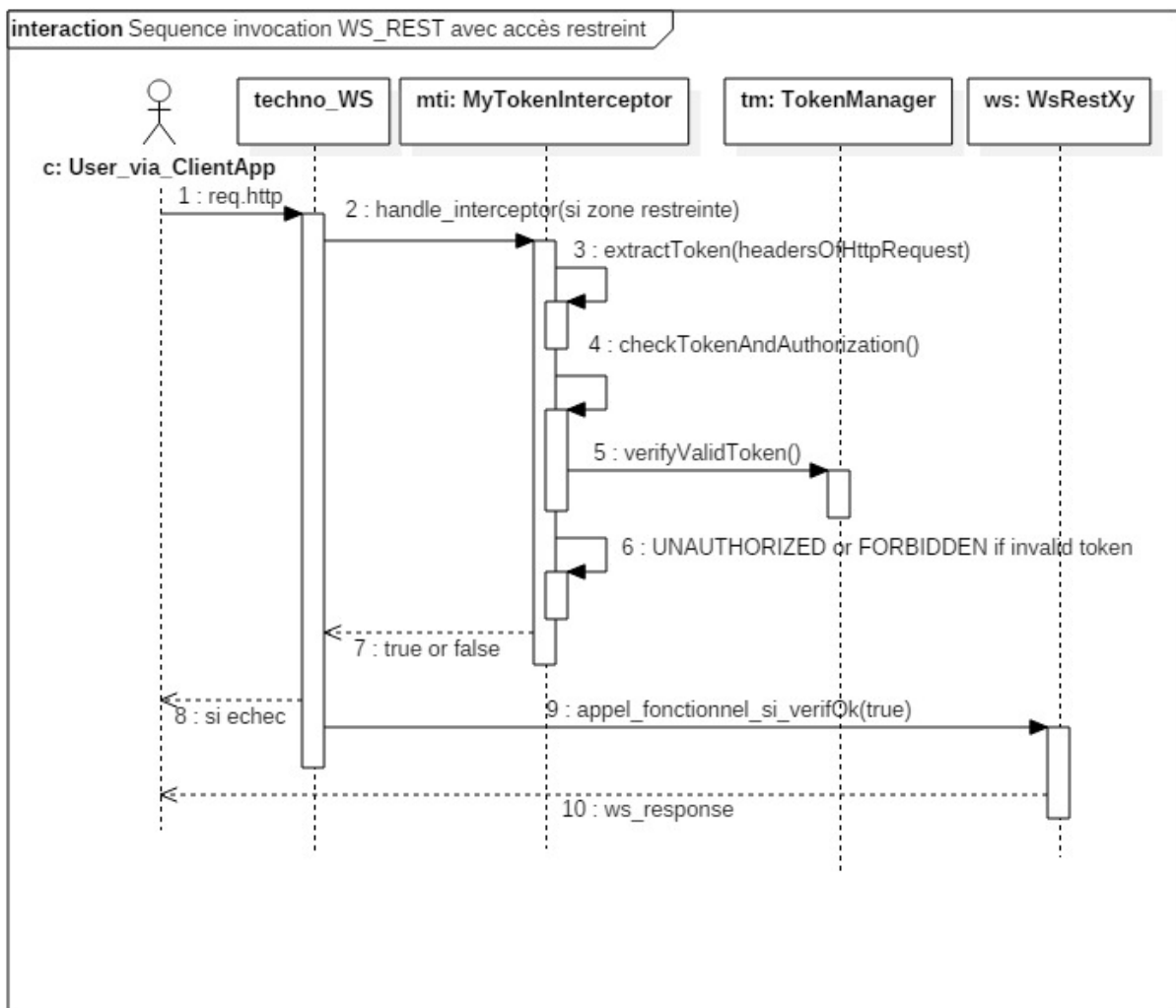
Authorization Headers (1) Body ● Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json) ▼

```
1 { "username" : "toto" , "password": "pwd_toto" }
```

```
{
  "authToken": "e51cd176-a522-454c-9c0a-36ca74cdb2d0",
  "authOk": true,
  "message": "authentification reussie"
}
```

2.4. Intercepteur et vérification d'un jeton



Status: 200 OK

si ok

Status: 403 Forbidden

ou bien

si faux jeton (invalid token)

Status: 401 Unauthorized

ou bien

si pas de jeton

ANNEXES

3. Migration "AngularJs/v1.x" et Angular 2+

3.1. Evolutions de AngularJs (V1) à Angular (V2)

Basé sur un modèle à base de composants (à la syntaxe plus rigoureuse), le framework "Angular 2" reprend les fonctionnalités fondamentales de la version 1 ("sorte de MVC" coté client/navigateur, appels de Web-services "REST") tout en supportant mieux les points suivants :

- liaison (binding) entre modèle et vue mieux contrôlé (plus performant, plus prédictible)
- meilleure adaptation à l'environnement cible (grands écrans, mobiles, ...)
- moins de conflits entre les librairies, ...

Le langage de développement préconisé a changé (javascript pour angular1, typescript pour Angular2). Plus fortement typé et plus expressif, le langage typescript permet d'écrire du code plus lisible, plus compact et plus facile à maintenir.

Après une longue période de gestation, la première version "finale" d'angular 2 se nomme en fait "Angular" tout court (sans JS). AngularJs étant le nom des anciennes versions 1.x.

<i>AngularJs (V1)</i>	<i>Angular (V2)</i>
Code javascript (ES5)	Code TypeScript compilé/transpilé en JS/ES5 ou bien DART ou bien ...
Logique structurelle : " parties de page " (contrôleur, \$scope, ...)	Logique structurelle : " composant et sous-composant " (orienté objet)
Binding bidirectionnel quelquefois pas performant (trop coûteux en opérations CPU)	Binding plus fin (dans un sens ou les deux)
Environnement de développement complètement libre (notepad++ peut suffire).	NodeJs et npm est nécessaire comme environnement de développement et il faut gérer une compilation/transpilation "ts → js".
Plutôt orienté "single page" / application simple	Utilisable sur des applications évoluées

Au final, le développement "Angular" (V2) est très différent de celui d'une application AngularJs.

NB : La notion de @Component de Angular >=2 correspond à une combinaison simplifiée de 2 composants complémentaires d'angular 1.x "Contrôleur + Directive".

D'une manière générale, les syntaxes angular2 sont beaucoup plus compactes et lisibles/maintenables que celles d'angular 1.x

4. Utilisation du service Http avec Promise

4.1. Promise via http en lecture/recherche (get)

Exemple élémentaire en lecture (inspiré du tutoriel officiel et un peu adapté) :

hero.service.ts

```
import {Injectable} from '@angular/core';
import {Hero} from './hero';
//import {HEROES} from './mock-heroes'; //old version without http , without in-memory-web-api
import { Headers, Http } from '@angular/http';

@Injectable()
export class HeroService {

  private headers = new Headers({'Content-Type': 'application/json'});

  private heroesUrl = 'app/heroes'; //URL to web api ou bien http://localhost:8080/xyzApp/heroes

  constructor(private http: Http) { }

  public getHeroPromise(id: number): Promise<Hero> {
    return this.getHeroesPromise()
      .then(heroes => heroes.find(hero => hero.id === id));
  }

  public getHeroesPromise(): Promise<Hero[]> {
    //return this.getHeroesPromiseQuickly();
    //return this.getHeroesPromiseSlowly();
    return this.getHeroesPromiseViaHttp();
  }

  private getHeroesPromiseViaHttp(): Promise<Hero[]> {
    return this.http.get(this.heroesUrl)
      .toPromise()
      .then(response => response.json().data as Hero[])
      .catch(this.handleError);
  }

  private handleError(error: any): Promise<any> {
    console.error('An error occurred', error); // for demo purposes only
    return Promise.reject(error.message || error);
  }
}
```

Configuration requise au niveau du module:

app.module.ts

```

import './rxjs-extensions';
import { NgModule }    from '@angular/core';
import { HttpClientModule } from '@angular/http';
import { HeroService }    from './hero.service';
...
@NgModule({
  imports:    [ ... , HttpClientModule ],
  declarations: [ AppComponent , ... ],
  providers:  [ HeroService ],
  bootstrap:  [ AppComponent ]
})
export class AppModule { }

```

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/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import 'rxjs/add/operator/do';             import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';            import 'rxjs/add/operator/switchMap';
// convert observable to promise:
import 'rxjs/add/operator/toPromise';

```

Utilisation depuis un composant de l'application :

```

...
export class HeroesComponent implements OnInit {
  public heroes : Hero[] ;
  ...
  constructor( private heroService: HeroService) { }
  ngOnInit(): void { this.getHeroes(); }
  getHeroes(): void {
    this.heroService.getHeroesPromise().then(heroes => this.heroes = heroes );
  }
  ...
}

```

4.2. Simulation de Promise (sans appel HTTP)

mock-heroes.ts

```
import {Hero} from './hero';

export var HEROES: Hero[] = [
  { "id": 11, "name": "Mr. Nice" },
  { "id": 12, "name": "Narco" },
  { "id": 13, "name": "Bombasto" },
  { "id": 14, "name": "Celeritas" },
  { "id": 15, "name": "Magenta" },
  { "id": 16, "name": "RubberMan" },
  { "id": 17, "name": "Dynamia" },
  { "id": 18, "name": "Dr IQ" },
  { "id": 19, "name": "Magma" },
  { "id": 20, "name": "Tornado" }
];
```

hero-service.ts

```
import {HEROES} from './mock-heroes';
...
public getHeroesPromise() : Promise< Hero[] > {
  return this.getHeroesPromiseQuickly();
  //return this.getHeroesPromiseSlowly(); // return this.getHeroesPromiseViaHttp();
}

private getHeroesPromiseQuickly() : Promise< Hero[] > {
  return Promise.resolve(HEROES);
}

private getHeroesPromiseSlowly() : Promise< Hero[] > {
  return new Promise<Hero[]>(resolve =>
    setTimeout(()=>resolve(HEROES), 2000) // 2 seconds
  );
}
```

4.3. Http/Promise en mode put, post, delete

Exemple élémentaire (inspiré du tutoriel officiel) :

hero.service.ts

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

@Injectable()
export class HeroService {

  private headers = new Headers({'Content-Type': 'application/json'});

  private heroesUrl = 'app/heroes'; //URL to web api ou bien http://localhost:8080/xyzApp/heroes

  constructor(private http: Http) {}

  ...
  public updatePromise(hero: Hero): Promise<Hero> {
    const url = `${this.heroesUrl}/${hero.id}`;
    return this.http
      .put(url, JSON.stringify(hero), {headers: this.headers})
      .toPromise()
      .then(() => hero)
      .catch(this.handleError);
  }

  public createPromise(name: string): Promise<Hero> {
    return this.http
      .post(this.heroesUrl, JSON.stringify({name: name}), {headers: this.headers})
      .toPromise()
      .then(res => res.json().data) // return new (created) Hero with (auto_incr) id
      .catch(this.handleError);
  }

  public deletePromise(id: number): Promise<void> {
    const url = `${this.heroesUrl}/${id}`;
    return this.http.delete(url, {headers: this.headers})
      .toPromise()
      .then(() => null)
      .catch(this.handleError);
  }
}
```

Exemples d'utilisation depuis un composant :

```
this.heroService.createPromise(name)
  .then(hero => { this.heroes.push(hero); this.selectedHero = null; });
```

```
this.heroService.updatePromise(this.hero)
  .then() => this.msg="updated");
```

```
this.heroService
  .deletePromise(hero.id)
  .then() => {
    this.heroes = this.heroes.filter(h => h !== hero);
    if (this.selectedHero === hero) { this.selectedHero = null; }
  });
```

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

Angular (v2) 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.

Configuration nécessaire au niveau de **app.module.ts**

```
import './rxjs-extensions';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/http';

// Imports for loading & configuring the in-memory web api
import { InMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService } from './in-memory-data.service';
...
import { HeroService } from './hero.service';
import { AppComponent } from './app.component';

@NgModule({
  imports: [ ... , HttpClientModule , InMemoryWebApiModule.forRoot(InMemoryDataService) ],
  ...
})
export class AppModule { }
```

Définition de la **structure** du **jeu de données** servant à définir la **simulation** du **service web** (en mode **CRUD**) :

in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    let heroes = [
      {id: 11, name: 'Mr. Nice'},
      {id: 12, name: 'Narco'},
      {id: 13, name: 'Bombasto'},
      {id: 14, name: 'Celeritas'},
      {id: 15, name: 'Magnetia'},
      {id: 16, name: 'RubberMan'},
      {id: 17, name: 'Dynamia'},
      {id: 18, name: 'Dr IQ'},
      {id: 19, name: 'Magma'},
      {id: 20, name: 'Tornado'},
      {id: 21, name: 'Didier'}
    ];
    return {heroes};
  }
}
```

Ce service "**InMemoryDataService**" étant associé à la racine de la configuration de **InMemoryWebApiModule** (au sein de **app.module.ts**) , l'URL relative de ce "pseudo service" sera

"app" (le nom du module courant).

De façon cohérente vis à vis de l'exemple précédent , l'accès en mode CRUD à la partie "heroes" se fera via l'URL relative "**app/heroes**" .

→ d'où la valeur

```
private heroesUrl = 'app/heroes'; //URL to web api
```

que l'on retrouvait dans les exemples précédents de ce chapitre .

XIV - Annexe – Ancien Http (avant HttpClient)

1. Utilisation du service Http avec Observable (rxjs)

1.1. Ancien service Http de Angular 2 à 4.2

NB : de la version 2.0 à 4.2 le framework angular ne proposait que le service Http pour appeler des Web Services REST .

Depuis la version 4.3 le nouveau service HttpClient rend obsolète l'ancien service Http .
D'autre part , les nouveautés syntaxiques apportées par RxJs 6 et Angular 6 augmente encore l'écart entre ancien et nouveau style concernant les appels http .

1.2. Observable (premières versions de rxjs , avant v6)

Angular (v2,v4,v5) 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';

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

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

NB: Le fichier *rxjs-extensions.ts* (rassemblant plein de import) est facultatif mais conseillé pour la ré-utilisabilité simple et rapide "des nombreux petits import" . ..

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';
```

1.3. Appel Http/get et réponse Observable (v2,v4,v5)

compte.service.ts

```
import {Injectable,Inject} from '@angular/core';
import './rxjs-extensions';
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 {
  private _headers = new Headers({'Content-Type': 'application/json'});
  // NB: my-api est à configurer dans proxy.conf.json (ng serve --proxy-config proxy.conf.json)
  private _compteUrlBase :string = "./my-api/comptes" ; // + ?numClient=...; // REST call
  constructor (private _http: Http ) {
  }

  public getComptesOfClientObservable(numCli: number) : Observable< Compte[] > {
    let comptesUrl : string = null;
    comptesUrl = this._compteUrlBase + "?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 { ...
  ngOnInit() { this.fetchComptes(); //refresh  }
  fetchComptes() {   this._compteService.getComptesOfClientObservable(this.clientId)
                    .subscribe(comptes => this.comptes = comptes ,
                               error => console.log(error));
  }
  ... }
```

1.4. Modes post, ... avec Observable (v2,v4,v5)

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

```
import './rxjs-extensions';
...
export class CompteService {
  private _headers = new Headers({'Content-Type': 'application/json'});
  private _virementUrl :string = "/my-api/virement" // POST REST call
  // Rappel: my-api est à configurer dans proxy.conf.json (ng serve --proxy-config proxy.conf.json)

  public postVirementObservable(virement: Virement): Observable<Virement> {
    console.log( "virementUrl = " + this._virementUrlBase);
    return this._http
      .post(this._virementUrlBase, 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));
  }
}
```

2. Retransmission des éléments de sécurité (v2,v4,v5)

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
- éventuels 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); //ici stockage du jeton dans localStorage HTML5
    //alternative : stockage du jeton en mémoire dans un (sous-)service commun.
  }
}
```

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

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

loadTokenFromLocalStorage(){
  var token = localStorage.getItem('token');
  if(token){
    this._headers.set('Authorization','Bearer ' + token);
  }
}
...
this.loadTokenFromLocalStorage();
return this._http.get(urlWsRest,{headers: this._headers})
  .map(response => response.json()).catch(e => Observable.throw(e));
...
```

XV - Annexe – RxJs

1. introduction à RxJs

1.1. Principes de la programmation réactive

La programmation réactive consiste essentiellement à programmer un enchaînement de traitements asynchrones pour réagir à un flux de données entrantes .

Un des intérêts de la programmation réactive réside dans la souplesse et la flexibilité des traitements fonctionnels mis en place :

- En entrée, on pourra faire librement varier la source des données (jeux de données statiques , réponses http , input "web-socket" , événement DOM/js ,)
- En sortie, on pourra éventuellement enregistrer plusieurs observateurs/consommateurs (si besoin de traitement en parallèles . par exemple : affichages multiples synchronisés) .

1.2. RxJs (présentation / évolution)

RxJs est une bibliothèque **javascript** (assimilable à un mini framework) spécialisée dans la programmation réactive .

Il existe des variantes dans d'autres langages de programmation (ex : RxJava , ...) .

RxJs s'est largement inspiré de certains éléments de programmation fonctionnelle issus du langage "**scala**" (map , flatMap , filter , reduce , ...) et a été à son tour une source d'inspiration pour "**Reactor**" utilisable au sein de Spring 5 .

RxJs a été dès 2015/2016 mis en avant par le framework Angular qui a fait le choix d'utiliser "Observable" de RxJs plutôt que "Promise" dès la version 2.0 (Angular 2) .

Depuis, le framework "Angular" a continué d'exploiter à fond la bibliothèque RxJs .
Cependant , les 2 frameworks ont beaucoup évolué depuis 2015/2016 .

La version **4.3** de **Angular** a apporté de grandes simplifications dans les appels de WS-REST via le service **HttpClient** (rendant *obsolète* l'ancien service *Http*) .

La version 6 de Angular a de son côté été **restructurée** pour intégrer les gros changements de **RxJs 6** . Heureusement, pas de bouleversement en V7 (tranquille continuité).

La version 6 de RxJs s'est restructurée en profondeur sur les points suivants :

- changement des éléments à importer (nouvelles syntaxes pour les import { } from "")
- changements au niveau des opérateurs à enchaîner (plus de préfixe , pipe() , ...) .

1.3. Principales fonctionnalités d'un "Observable"

Observable est la structure de données principale de l'api "RxJs" .

- Par certains cotés , un "Observable" ressemble beaucoup à un objet "Promise" et permet d'enregistrer élégamment une suite de traitements à effectuer de manière asynchrone .
- En plus de cela , un "Observable" peut facilement être manipulé / transformé via tout un tas d'opérateurs fonctionnels prédéfinis (ex : map , filter , sort , ...)
- En outre , comme son nom l'indique , un "Observable" correspond à une mise en oeuvre possible du design pattern "observateur" (différents observateurs synchronisés autour d'un même sujet observable).

2. Fonctionnement (sources et consommations)

Source configurée et initialisée

```
.pipe(
  callback_fonctionnelle_1 ,
  callback_fonctionnelle_2 ,
  ....
) .subscribe(callback_success , callback_error , callback_terminate)
```

NB : les paramètres callback_error et callback_terminate de .subscribe() sont facultatifs et peuvent donc être omis s'ils ne sont pas utiles en fonction du contexte.

NB : il faut (en règle général , sauf cas particulier/indication contraire) appeler .subscribe() pour que la chaîne de traitement puisse commencer à s'exécuter .

3. Réorganisation de RxJs (avant et après v5,v6)

La version 6 de RxJs a été beaucoup restructurée :

- plus de préfixe "Observable." devant of() et autres fonctions
- pipe() nécessaire pour enchaîner une série d'opérateurs (ex : map , filter , ...)
- réorganisation des éléments à importer

Quelques correspondances "avant/après" pour une éventuelle migration :

anciennes versions de RxJs (ex : v4)	versions récentes de RxJs (ex : v6)
Observable.of(data) ;	of(data) ;
import { } from "" ;	import { Observable, of } from "rxjs"; import { map , flatMap ,toArray ,filter} from 'rxjs/operators';

3.1. Imports dans projet Angular / typescrit

```
import { Observable, of } from 'rxjs';
import { map, flatMap, toArray, filter } from 'rxjs/operators';
```

exemple d'utilisation (dans classe de Service) :

```
public rechercherProduitSimu$(prixMaxi : number) : Observable<Produit[]> {
  let tabProduit = [
    { numero : 1, label : "produit 1", prix : 50 },
    { numero : 2, label : "produit 2", prix : 30 },
    { numero : 3, label : "produit 3", prix : 80 },
    { numero : 4, label : "produit 4", prix : 500 }
  ]
  return of(tabProduit)
    .pipe(
      flatMap(pInTab=>pInTab),
      map((p : Produit)>=>{p.label = p.label.toUpperCase(); return p;}),
      filter((p) => p.prix <= prixMaxi),
      toArray()
    );
}
```

3.2. Imports "umd" dans fichier js (navigateur récent)

EssaiRxjs.html

```
...
<body>
  Essai Rxjs (Observable, pipe, subscribe)
  <hr/>
  <p>ouvrir la console web du navigateur</p>
  <script src="lib/rxjs.umd.min.js"></script>
  <script src="js/essaiRxjs.js"></script>
</body>
...
```

avec `rxjs.umd.min.js` récupéré via <https://unpkg.com/rxjs/bundles/rxjs.umd.min.js> ou bien via une URL externe directe (CDN) `<script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>`

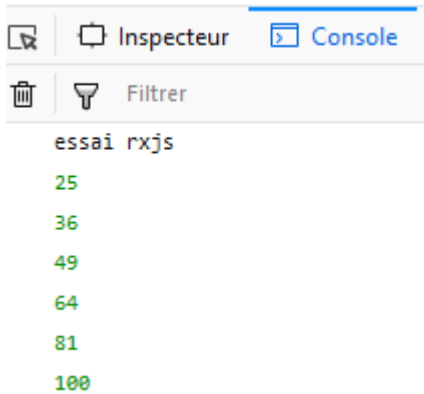
NB : "The global namespace for rxjs is rxjs"

essaiRxJs.js

```
console.log('essai rxjs');

const { range } = rxjs;
const { map, filter } = rxjs.operators;

range(1, 10).pipe(
  filter(x => x ≥ 5),
  map(x => x * x)
).subscribe(x => console.log(x));
```



4. Sources classiques générant des "Observables"

4.1. Données statiques (tests temporaires , cas très simples)

```
let jsObject = { p1 : "val1" , p2 : "val2" } ;
of(jsObject)...subscribe(...) ;

let tabObj = [ { ...} , { ... } ] ;
of(tabObj)...subscribe(...) ;

of(value1 , value2 , ... , valueN)...subscribe(...) ;
```

4.2. Données numériques : range(startValue,endValue)

```
range(1, 10).subscribe(x => console.log(x)) ;
1
2
...
10
```

4.3. source périodique en tant que compteur d'occurrence

```
const obsvI1 = interval(1000 /*ms*/);
//subscriptionObjsvI1 is the result of .subscribe() call
const subscriptionObjsvI1 = obsvI1.subscribe(n =>
{ console.log(` the number of interval  occurrence (starting at 0) is ${n} `);
  if(n>=5) {
    subscriptionObjsvI1.unsubscribe(); //stop if n>=5
  }
});
```

4.4. source en tant qu'événement js/DOM

```
import { fromEvent } from 'rxjs';

const el = document.getElementById('my-element');

// Create an Observable that will publish mouse movements
const mouseMoves = fromEvent(el, 'mousemove');

// Subscribe to start listening for mouse-move events
const subscription = mouseMoves.subscribe((evt /*: MouseEvent */) => {
  // Log coords of mouse movements
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);

  // When the mouse is over the upper-left of the screen,
  // unsubscribe to stop listening for mouse movements
  if (evt.clientX < 40 && evt.clientY < 40) {
    subscription.unsubscribe();
  }
});
```

4.5. source en tant que réponse http (sans angular HttpClient)

```
import { ajax } from 'rxjs/ajax';

// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

4.6. source en tant que données reçues sur canal web-socket

...

5. Principaux opérateurs (à enchaîner via pipe)

Rappel (syntaxe générale des enchaînements) :

```
Source_configurée_et_initialisée
.pipe(
  callback_fonctionnelle_1 ,
  callback_fonctionnelle_2 ,
  ....
).subscribe(callback_success , callback_error , callback_terminate)
```

avec plein de variantes possibles

Exemple :

Principaux opérateurs :

map	Transformations quelconques (calculs , majuscules , tri ,)
flatMap	
toArray	
filter	Filtrages (selon comparaison,)

5.1. map() : transformations

En sortie , résultat (retourné via return) d'une modification effectuée sur l'entrée .

Exemple 1:

```
const obsNums = of(1, 2, 3 ,4 ,5);
const squareValuesFunctionOnObs = map((val) => val * val);
const obsSquaredNums = squareValuesFunctionOnObs(obsNums);
obsSquaredNums.subscribe(x => console.log(x));
```

// affiche 1 4 9 16 25

Exemple 2:

```
const obsStrs = of("un" , "deux" , "trois");
obsStrs.pipe(
```

```

        map( s => s.toUpperCase() )
    )
    .subscribe(s => console.log(s));
// affiche UN  DEUX  TROIS

```

5.2. flatMap() et toArray()

De façon à itérer une séquence d'opérateurs sur chaque élément d'un tableau tout en évitant une imbrication complexe et peu lisible de ce type :

```

observableSurUnTableau
.pipe(
    map((tableau)=>{
        return tableau.map(
            (itemInTab)=>{itemInTab.label = itemInTab.label.toUpperCase();
            return itemInTab;}
        );
    });
);

```

on pourra placer une séquence d'opérateurs qui agiront sur chacun des éléments du tableau entre **flatMap()** et **toArray()** :

```

observableSurUnTableau
.pipe(
    flatMap(itemInTab=>itemInTab) ,
    map(( itemInTab )=>{ ... } ) ,
    filter((itemInTab) => itemInTab.prix <= 300 ) ,
    toArray()
).subscribe( (tableau) => { ... } );

```

5.3. filter()

```

const obsVals = of(12 , -15 , 30 , -8 , 40);
obsVals.pipe(
    filter( (v) => v >= 0 )
)
.subscribe(v => console.log(v));

```

//affiche 12 30 et 40

```

range(1,10).pipe(
    filter( (v) => v % 2 === 0 )
)
.subscribe(v => console.log(v + " est une valeur paire"));

```

5.4. map with sort on array

```

const obsTab = of([ {numero:1,label:'produit1',prix:40.0},
                    {numero:2,label:'produit2',prix:30.0},
                    {numero:3,label:'produit3',prix:35.0},

```

```

        {numero:4,label:'produit4',prix:15.0},
        {numero:5,label:'produit5',prix:35.0}
    ]);
obsTab.pipe(
    map( (tab) => tab.sort( (p1,p2) => (p1.prix > p2.prix) ) )
)
.subscribe(t => console.log( JSON.stringify(t) ));

```

6. Passerelles entre "Observable" et "Promise"

6.1. Source "Observable" initiée depuis une "Promise" :

```

import { from } from 'rxjs';

// Create an Observable out of a promise :
const observableResponse = from(fetch('/api/endpoint'));

observableResponse.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});

```

6.2. Convertir un "Observable" en "Promise"

```

observableXy.toPromise()
    .then(...)
    .catch(...)

```

XVI - Annexe – Web Services REST (coté serveur)

1. Généralités sur Web-Services REST

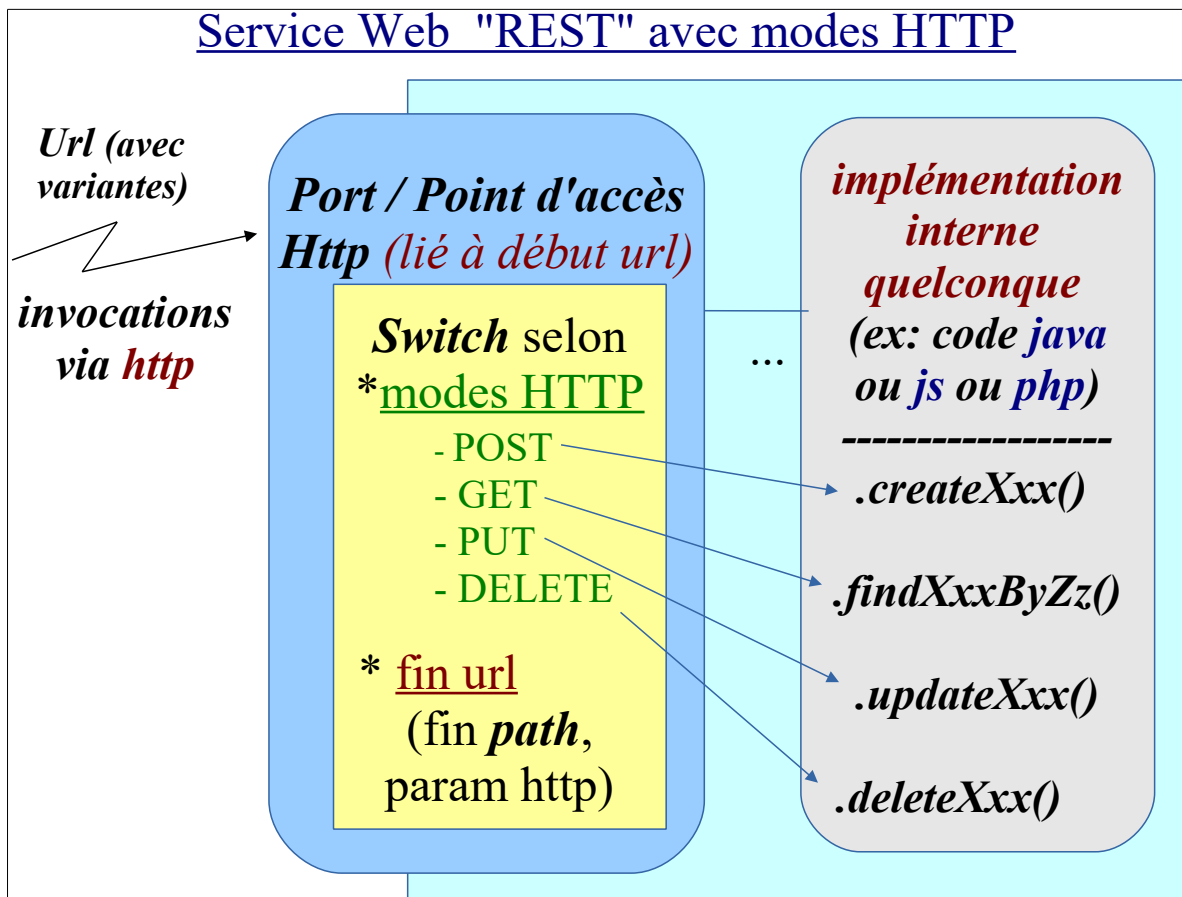
2 grands types de services WEB: SOAP/XML et REST/HTTP

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.

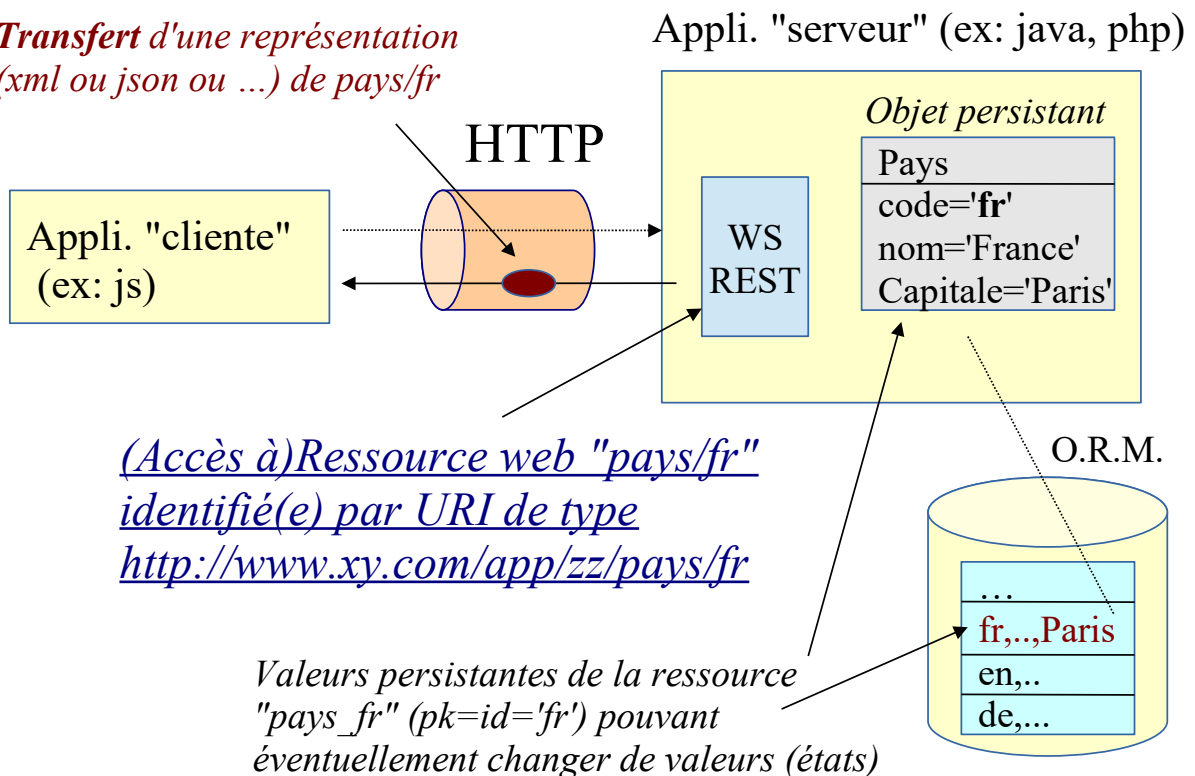
Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

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>

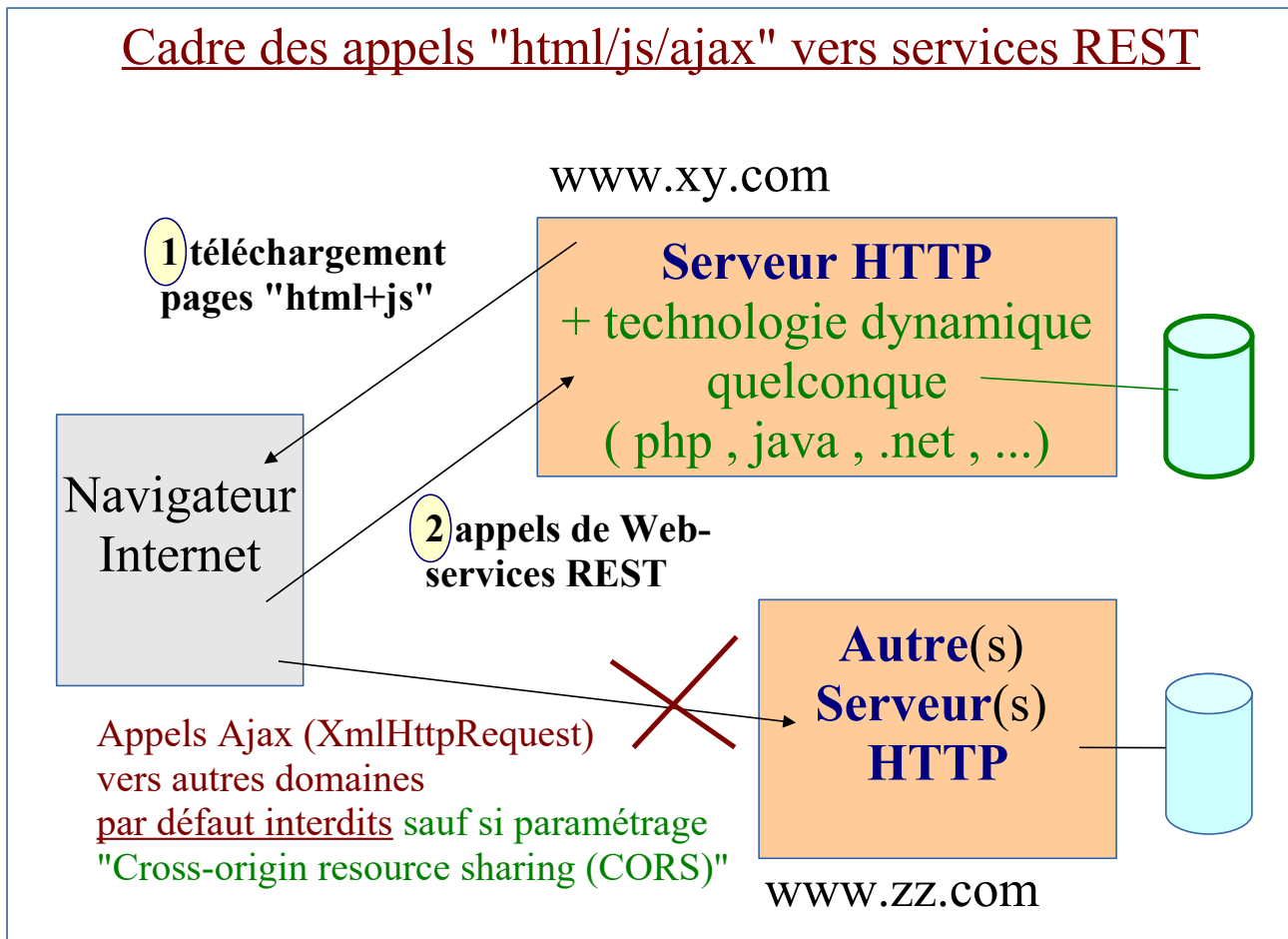
donne les résultats suivants "json" ou "xml":

```
{ "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")
@CrossOrigin(allowAllOrigins = true)
// ou bien autorisations plus fines
public class ClientRestJsonService {
    ...}
```

code java

Paramétrage "CORS" avec Spring-mvc

```
import org.springframework.web.bind.annotation.CrossOrigin;
...
@RestController
@CrossOrigin(origins = "*")
//@CrossOrigin(origins = { "http://localhost:4200" ,
//                        "http://www.partenaire-particulier.com" })
@RequestMapping(value="/rest/products" , headers="Accept=application/json")
public class ProductCtrl {
    ...
}
```

et/ou implémentation de *WebMvcConfigurerAdapter*
comportant une redéfinition de `public void addCorsMappings(CorsRegistry registry) .`