# GSB avec Angular2, partie 1

## Premiers pas

## L'installation

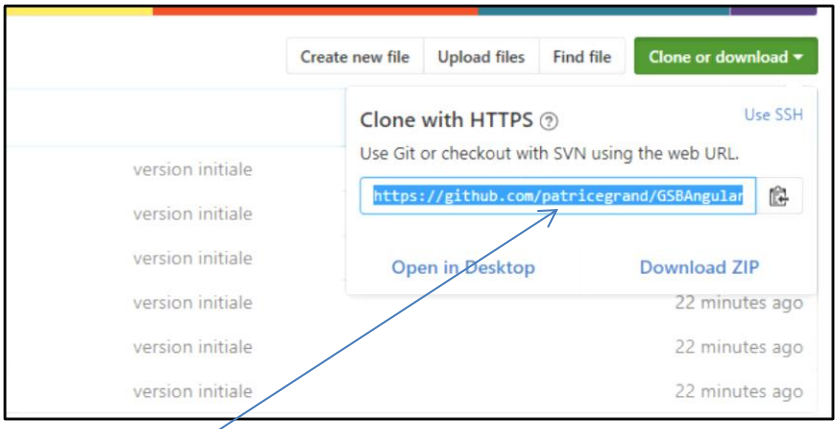
Angular 2 fait appel à des modules Node.js ; ainsi il faut au préalable installer Node.js sur son serveur ; si l'on travaille en local, c'est sur sa machine qu'il faudra bien sûr l'installer.
Pour installer Node.js, il suffit d'aller sur le site https://nodejs.org/en/download/ , et de sélectionner votre environnement ; si vous êtes sous Windows, privilégier le *.msi*.
Node.js est un Framework qui propose côté serveur de nombreuses bibliothèques de classes (des modules) ; plusieurs présentations générales du rôle de *Node.js* sont disponibles sur internet, elles sont suffisantes pour la suite.

L'installation de Node.js inclut celle du gestionnaire de packages NPM (Node Package Modules) ; la plupart des Frameworks proposent aujourd'hui ce type d'outil qui permet de gérer les installations, les dépendances et mises à jour de packages de manière très simple : NuGet chez Microsoft, Composer pour PHP, etc…

Nous allons ensuite installer une application minimum GSBAngular2 à partir de github dont les sources sont disponibles ici :

https://github.com/patricegrand/GSBAngular2_V1.0.git.

1) Cliquer sur *Clone or download*




2) Si vous avez un client Git, vous copiez l'url ci-dessus. Dans l'explorateur de fichiers, vous vous placez là où vous souhaitez installer l'application (ce peut être n'importe où sur le disque), vous ouvrez à cet endroit une fenêtre de commande (Ctrl + MAJ + click droit/*ouvrir une fenêtre de commande ic*i) et dans la console vous tapez :

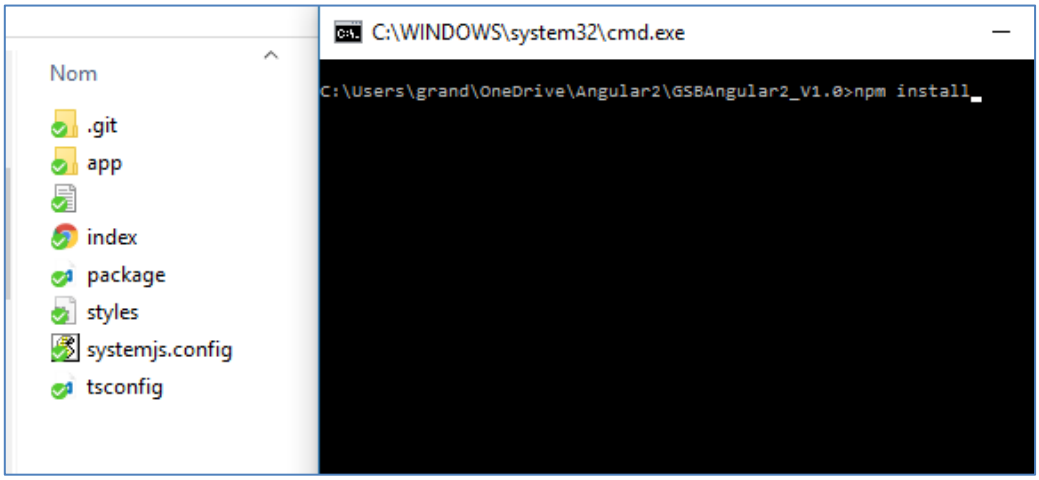
   **git clone** <*url collée*>

3bis) Sinon, vous téléchargez le fichier zip que vous décompressez n'importe où sur votre disque.

**Travail à faire**

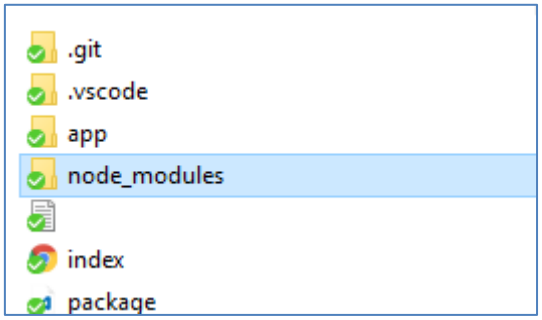
En suivant les étapes décrites plus haut installer Node.js ainsi que l'application initiale **GSBAngular2_V0**.

Avant de lancer l'application, il faut au préalable installer à la racine du répertoire de l'application les packages Angular nécessaires. Ceci se réalise grâce à la commande *npm install* :

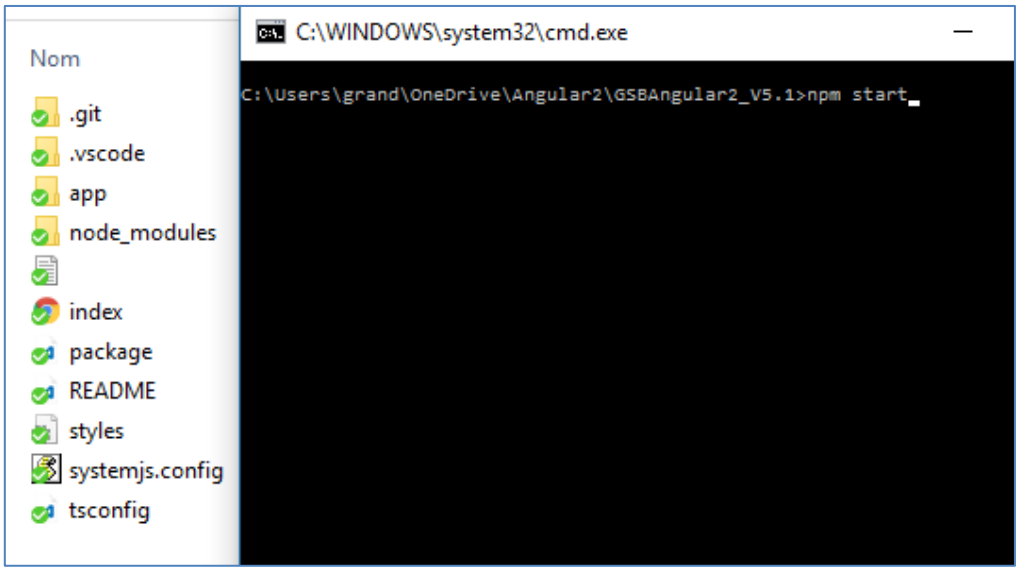



L'installation des packages prend un certain temps ; il ne doit pas y avoir de message d'erreur, les *warning* peuvent être ignorés.

Un répertoire *node_modules* est ajouté, il contient les bibliothèques nécessaires à l'application :




Dernière étape, le lancement de l'application ; ceci se réalise grâce à la commande *npm start* :




Cette commande npm start *transpile* le code TypeScript en JavaScript, lance un ***serveur web local*** ainsi que le navigateur par défaut avec le fichier index.html.
Si tout se passe normalement, vous devez voir apparaitre le titre « Gestion des rapports de visite » !!

En observant le code, on constate que la *transpilation* a généré, pour chaque fichier TypeScript, un fichier JavaScript ainsi qu'un fichier map (utile pour le débogage).

**Travail à faire**

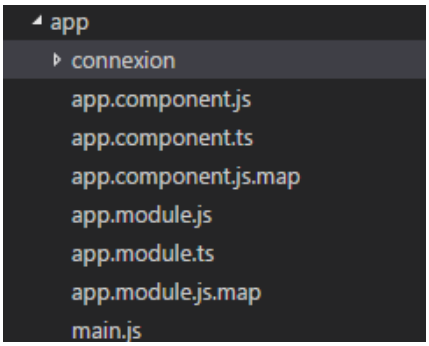
Exécuter les deux commandes *npm install* et *npm start* ; vérifiez que tout se passe bien.

## Création d'un premier composant

Une application Angular 2 est une hiérarchie de composants à développer ; chaque *composant* est un nouvel élément html qui offre une représentation (code html), un comportement (code TypeScript), un style au sens CSS. On peut être amené à regrouper des composants dans un *module* par souci d'organisation. L'application doit contenir au moins un module ; dans notre application initiale, un seul module est utilisé, le module *app*.
L'application de gestion des rapports de visite commence par la connexion du visiteur médical ; nous allons créer un composant gérant la connexion, pas à pas.

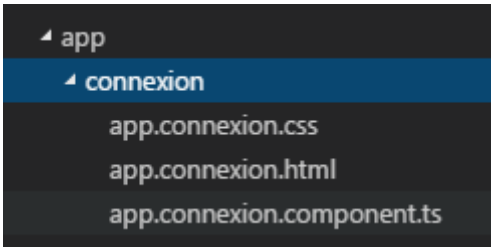
### 1) Création du composant de connexion, version simplifiée

Chaque composant sera créé dans un répertoire spécifique (règle d'usage) et ceci à l'intérieur du module *app*. Par exemple ici le répertoire connexion sera utilisé pour le composant de connexion.




#### 1.a) Etape 1, création des fichiers nécessaires

Nous allons regrouper les fichiers constituant le composant dans ce répertoire *connexion* :




- Le fichier de définition du composant : ***app.connexion.component.ts***
- Le fichier décrivant la « vue » du composant : ***app.connexion.html***
- Le fichier décrivant le style du composant : ***app.connexion.css***

Remarque : notez les ***règles de nommage d'usage des fichiers*** *<nom module>.<nom du composant>*…
Dans cette première étape, notre composant de fera pas grand-chose, si ce n'est afficher une simple légende ; nous nous concentrons ici sur « l'architecture » du composant.

**Travail à faire**

Créer le répertoire connexion ainsi que les 3 fichiers (sans code pour le moment) nécessaires à l'implémentation du composant de connexion.

Regardons ce que devront contenir ces fichiers.

### 1.a.1 le fichier *app.connexion.component.ts*

app.connexion.component.ts

```
1   import { Component } from '@angular/core';
2
3   @Component({
4     moduleId: module.id ,
5     selector: 'my-connexion',
6     templateUrl: 'app.connexion.html',
7     styleUrls: ['app.connexion.css']
8
9   })
10  export class ConnexionComponent {
11  }
```

- La ligne 10 définit le composant qui est une simple classe *TypeScript*, sans code pour l'instant. Le qualificateur *export* indique que cette classe pourra être utilisée en dehors de son fichier.
- Les lignes 3 à 8 *enrichissent –décorent-* cette classe à l'aide du décorateur **Component** (ligne 3) – les décorateurs sont suffixés par un @-. Techniquement, le décorateur est une fonction (noter les parenthèses) qui prend comme argument un objet TypeScript (noter le bloc d'accolages) ; cet objet contient des champs dont le seul imposé est « *selector* » (c'est la manière dont le composant sera appelé dans le code html, ici *<my-connexion></my-connexion>*.
- Ainsi, le décorateur expose ici 4 propriétés associées à la classe ConnexionComponent :
  - ligne 4, *moduleId* dont la valeur module.id permet de lever tous les problèmes d'accès à ce composant ; bref, il faut toujours ajouter cette propriété et cette valeur
  - ligne 5, *selector*, décrit plus haut
  - ligne 6, *templateUrl* qui permet d'associer un fichier html plutôt qu'une description *inline* du code html, comme c'était les cas pour *AppComponent*.
  - ligne 7, styleUrls qui permet d'associer un tableau de fichiers (noter le bloc de crochets distinctif des tableaux) de style au composant.

- La ligne 1, obligatoire pour pouvoir utiliser le décorateur *Component (ligne 3)* ; on importe Component (une fonction) depuis *@angular/core*.

### 1.a.2 le fichier app.connexion.html

Le code est minimum pour cette étape :

app.connexion.html

```
1   <h1>
2       Connexion
3   </h1>
4
```

### 1.a.3 le fichier app.connexion.css

Là encore, le code proposé ne sert qu'à comprendre l'architecture d'un composant :

app.connexion.css

```
1  h1 {
2    color: #100;
3    font-family: Arial;
4    font-size: 200%;
5  }
```

Il nous reste à rendre opérationnel l'ensemble c'est-à-dire en intégrant et appelant le composant *ConnexionComponent* :

### 1.b) Etape 2, intégration et appel du composant

Il faut commencer par ajouter le composant au module :

app.module.ts

```
1  import { NgModule }      from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { FormsModule } from '@angular/forms';
4
5  import { AppComponent }   from './app.component';
6  import { ConnexionComponent }   from './connexion/app.connexion.component';
7  @NgModule({
8    imports:      [ BrowserModule,FormsModule ],
9    declarations: [ AppComponent,ConnexionComponent ],
10   bootstrap:    [ AppComponent ]
11 })
12 export class AppModule { }
```

Dans la partie *declarations* du module, on ajoute au tableau le nouveau composant (ligne 9) ; pour cela il faut également importer la classe (ligne 6).
Remarques :

- le nom du composant est celui de sa classe,
- l'import du module *FormsModule* est nécessaire pour créer le futur formulaire, sans oublier de faire un import du fichier correspondant (ligne 3).

Il ne reste plus qu'à appeler le composant ; une solution est d'utiliser le composant *racine*, *AppComponent*, et d'insérer l'appel du composant créé :

app.component.ts

```
1  import { Component } from '@angular/core';
2  @Component({
3    selector: 'my-app',
4    template: `<h1>Gestion des rapports de visite </h1>
5               <my-connexion></my-connexion>`
6  })
7  export class AppComponent {
8   }
9
```

Ligne 5, on insère simplement la balise *my-connexion*.
Remarque : les « quote » déjà présentes doivent être modifiées en « anti-quote » (seulement dans le cas où le code HTML est écrit sur plusieurs lignes, ici 2).

**Travail à faire**

Ecrire le code du composant *ConnexionComponent*. Testez en lançant la commande *npm start*, à partir d'une console ouverte dans le répertoire de l'application ou à partir d'une console intégrée à votre IDE.

## 2) Amélioration du composant de connexion

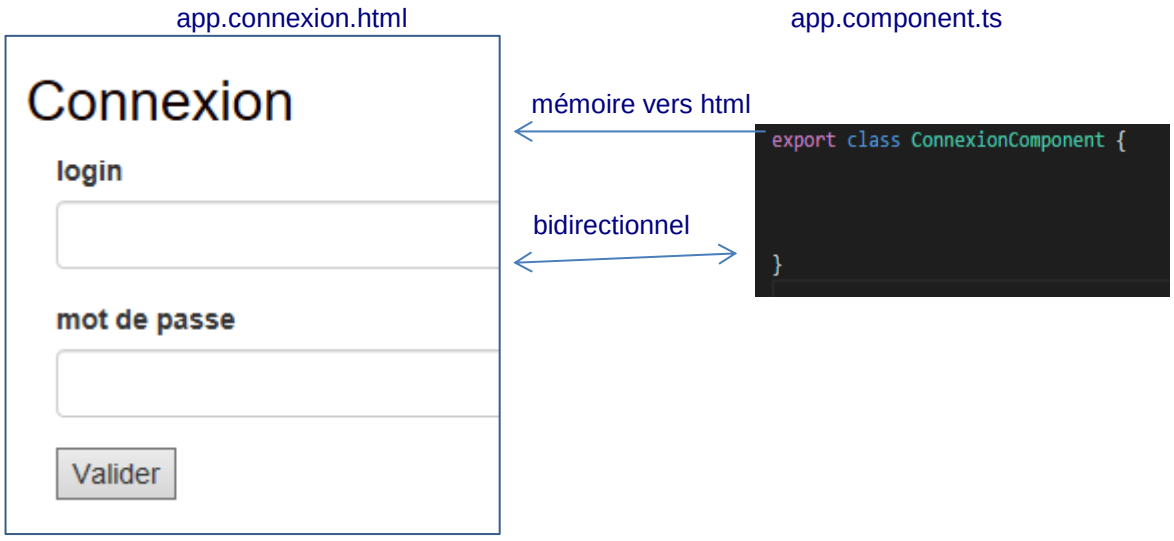
Nous avons vu comment créer et utiliser un composant. Nous allons maintenant nous concentrer sur ses fonctionnalités.

La page de connexion doit ressembler à ceci :


C'est un formulaire très classique avec des zones affichées lors de la première soumission, des zones de saisies à remplir par l'utilisateur et qui seront évaluées après la validation.
Pour mettre en œuvre tout ceci, Angular 1 et 2 proposent un mécanisme très personnel de *binding*, c'est-à-dire de liaison entre les données affichées (ou saisies) et leur représentation en mémoire. C'est pourquoi il nous faut faire un petit détour par la gestion du binding dans Angular 2.

### 2.1 Le binding

Ainsi, le binding lie des données affichées dans les pages html et ces données en mémoire ; la classe du composant est le lieu candidat à cette gestion en mémoire :



### 2.1.1 binding unidirectionnel vers le code html

Les 3 zones de libellé, « Connexion », « login », « mot de passe » peuvent être affichées à partir de données internes à la classe ConnexionComponent ; par exemple pour le titre :

Dans le code html

app.connexion.html

```
<h1>
    {{titre}}
</h1>
```

Dans la classe

```
export class ConnexionComponent {

  titre : string ="Connexion";

```

**Remarque** : pour déclarer un champ (attribut) de la classe, TypeScript utilise la syntaxe :
*<nom du champ> : <type>( = <valeur>)* ; la valeur est optionnelle mais nécessaire ici.
Toute variable se déclare de la même manière.
Pour atteindre un champ dans une *méthode* de la classe, on le préfixe par *this*, *this.titre = « »* ;
Ceci est la première forme de binding, celle qui valorise des éléments html par du code dans la classe associée : le code html « connait » le champ titre et se chargera à l'exécution de remplacer {{titre}} par sa valeur.

### 2.1.2 Binding bidirectionnel

Ce binding permet, par exemple, d'initialiser un champ input à partir de la mémoire et de récupérer dans la classe les données saisies par l'utilisateur.

Dans le code html

```
<form  name="frmLogin" >
    <input type="text"
           name="lg"
           [(ngModel)]="login"
    />
```

Dans la classe

```
export class ConnexionComponent {
  login : string = "Dupond";
```

**Remarques** :

- L'élément input doit avoir un attribut *name* valorisé,
- le binding vers la classe se réalise grâce à la directive *ngModel* (propre à Angular2),
- notez la syntaxe un peu « particulière » qui accompagne cette directive ; certains auteurs évoquent à ce propos « une banane dans une boite » pour s'en souvenir…,
- ainsi, dans la classe, la propriété *login* sera disponible pour tout traitement,
- l'initialisation de la propriété login n'a bien sûr pas de grand intérêt ici, elle pourrait être avantageusement remplacée par un attribut *placeholder*.

### 2.1.3 Le binding bidirectionnel un peu particulier

Il s'agit du mixte des deux précédents ; si l'on désire afficher dans le formulaire ce qui est saisi dans un élément *input*, on peut combiner les deux syntaxes :

Dans le code html

```
<form  name="frmLogin" >
    <input type="text"
           name="lg"
           [(ngModel)]="login"
    />
<p>{{login}}</p>
```

Dans la classe

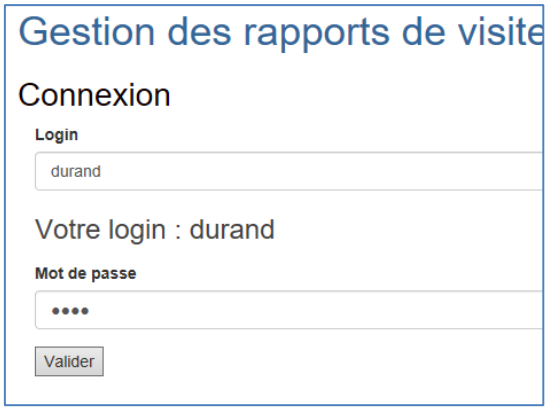
```
export class ConnexionComponent {
  login : string ;
```

A chaque modification de la donnée saisie, la propriété *login* se mettra à jour dans le formulaire.

L'annexe 1.a entre un peu plus dans le détail du mécanisme de binding.

**Travail à faire**

En vous aidant de l'annexe 1.b, compléter le code du composant ConnexionComponent afin de produire la sortie suivante :




Pour les zones affichées, vous utiliserez le binding présenté au paragraphe 2.1.1
Vous ne vous souciez pas du bouton de validation, c'est l'objet du point suivant.

### 2.2 La soumission du formulaire

Lorsque l'on clique sur le bouton valider, on imagine qu'un traitement va s'effectuer côté serveur ; comme nous n'avons pas encore mis en place la partie serveur, nous allons nous concentrer sur le traitement côté JavaScript (pour nous TypeScript).

Le mécanisme est proche du JavaScript traditionnel ; l'application va « réagir » à un événement de type *click*. C'est dans les balises du formulaire que l'on indique la méthode associée à l'événement :

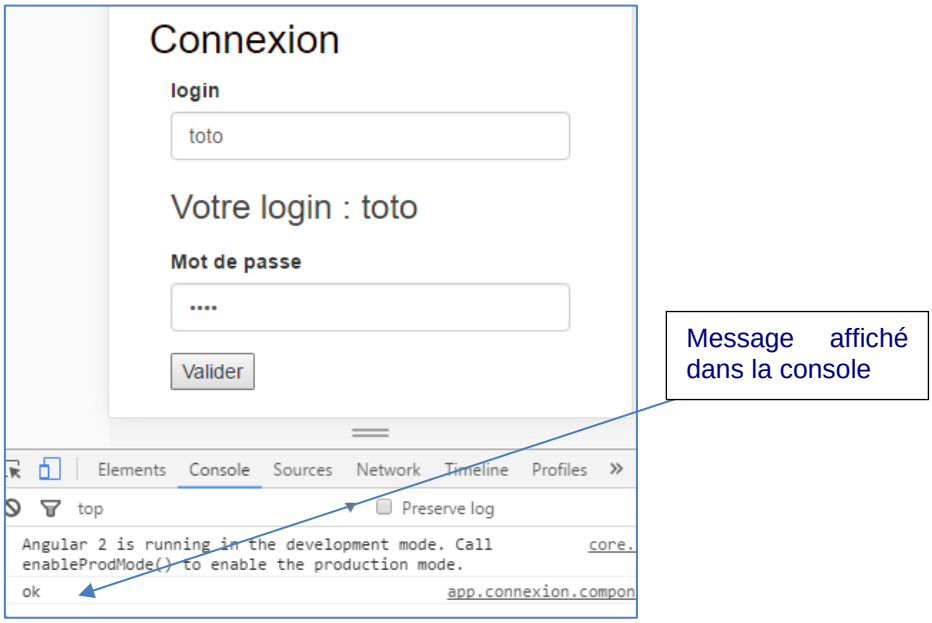
```
<form class="col-lg-6" name="frmLogin" (ngSubmit)="valider()">
```

ngSubmit est une *directive* du Framework, la présence des parenthèses indique le binding entre l'événement *click* sur le bouton associé et la méthode *valider*.
Cette méthode *valider* se trouve bien sûr dans la classe ConnexionComponent :

```
valider() : void{
          if(this.login != "toto" || this.mdp !="titi")
            console.log("ok");
          else
            console.log("erreur");
```

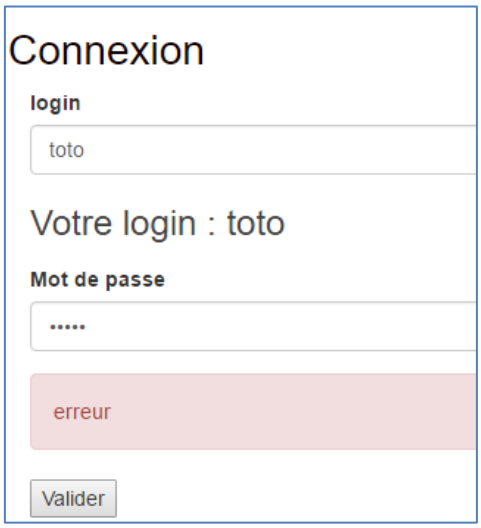
**Remarques** :

- Vous pouvez *copier/coller* ce code.
- La syntaxe de la déclaration de la méthode (ligne 14) peut paraître troublante en TypeScript, elle respecte néanmoins le format :
  nom de la propriété (ou méthode), deux points (:), type (ici void), code de la méthode ou valeur éventuelle de la propriété. Comme pour les propriétés ou variables, le type n'est pas obligatoire ; ainsi on peut écrire : valider : {...}.
- Bien sûr, les valeurs attendues des *login* et *mdp* ne seront pas en *dur*, mais issues d'une requête envoyée au serveur.
- L'opérateur *this* est obligatoire.




Dernier point à aborder ; présentons une sortie plus digeste que la console pour l'utilisateur. Ceci va nous permettre de découvrir une nouvelle dimension d'Angular2.

Nous désirons avoir un message en direction de l'utilisateur dans l'hypothèse où la saisie est erronée :




Nous vous fournissons le code html produisant le message :

```
<div class="alert alert-danger" [hidden]="estCache">{{lblMessage}} </div>
```

- Vous pouvez copier/coller ce code,
- les classes CSS sont des classes Bootstrap,
- *hidden* est une *propriété* de la *div*, de type booléen.

**Travail à faire**

Compléter vos codes sources afin de gérer la soumission du formulaire via notamment la méthode *valider*.
La gestion du message d'erreur en cas d'échec de connexion devra être intégrée.

Vous trouverez la correction sur le site :

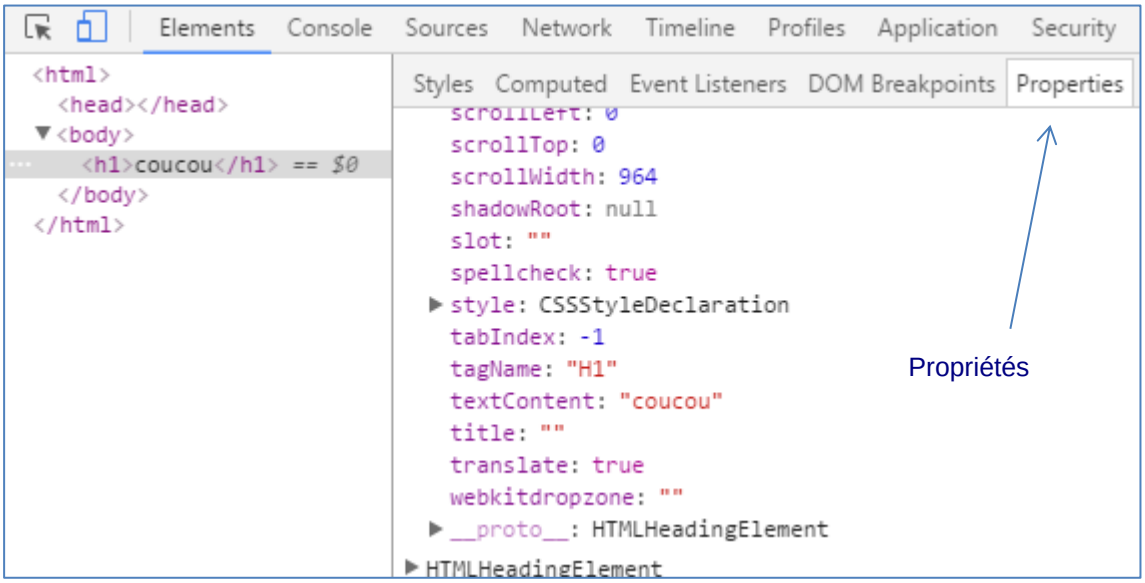
https://github.com/patricegrand/GSBAngular2_V1.1.git

## Annexe 1.a pour aller plus loin avec le binding

Revenons un peu sur html, ce langage bien connu ; il est composé de balises, nommées dans le DOM éléments html. Chaque élément html possède des attributs universels (name, id, type,..) qui pour certains éléments html, n'ont aucun effet.
Le navigateur, pour chaque balise (h, br, input,…) rencontrée, va générer (à l'aide de son API propre) un *élément html* sous la forme d'un nœud. Ainsi, chaque élément html (un objet) va être pourvu de *propriétés* correspondant aux attributs de la balise.
Un petit exemple vaut mieux qu'un long discours :
Soit le code html ultra simple :

```
<h1>coucou</h1>
```

Lançons son interprétation dans Chrome, en mode développement (F12) :




On constate que *textContent* est une propriété de l'élément *h1* et que sa valeur est « coucou ».
Ainsi, puisque c'est une propriété (lecture/écriture) on pourrait réaliser la même chose en utilisant la propriété *textContent* plutôt que l'affichage brut entre les deux h1.

Bref, une balise possède des attributs, l'élément html généré lui des propriétés ; une grosse différence entre les deux est que la propriété est *sensible à la casse.*

Angular 2 recommande d'utiliser les propriétés.
Ainsi, le code «recommandé» pour afficher la propriété titre est :

```
<h1 [textContent]="titre">

</h1>
```

Les crochets permettent d'atteindre la propriété, cette propriété est bindée à la propriété *titre* de la classe associée.

La syntaxe utilisée dans le support est ce que certains auteurs appellent, de manière très imagée, du « sucre syntaxique » -un raccourci syntaxique- :

```
<h1>{{titre}}

</h1>
```

Ce raccourci est très souvent utilisé pour faire référence à la propriété *textContent* ; de plus il avait fait les beaux jours d'Angular 1…

Nous aurons l'occasion de rencontrer les propriétés (autre que textContent) qu'il faudra gérer en utilisant les crochets.

Concernant le binding bidirectionnel, on comprend mieux cette présence des crochets dans la syntaxe :

```
<form  name="frmLogin" >
    <input type="text"
           name="lg"
           [(ngModel)]="login"
    />
<p>{{login}}</p>
```

*ngModel* est traitée comme une propriété –présence des crochets- (Angular 2 parle de directive) ; de plus, un mécanisme événementiel est inséré de manière transparente (mais visible avec Chrome) afin de gérer chaque modification.

## Annexe 1.a : trame du code HTML pour le formulaire de connexion

Pour utiliser un style un peu plus professionnel, nous avons utilisé la bibliothèque BootStrap ; il faut ajouter son appel dans le fichier index.html :

index.html

```
<!DOCTYPE html>
<html>

<head>
    <title>GSB gestion des rapports de visite</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" >
    <link rel="stylesheet" href="styles.css">
```

On vous fournit la trame du code du fichier app.connexion.html demandé :

```
<form class="col-lg-6" name="frmLogin" >
    <div class="form-group">
        <label for="login"></label>
        <input type="text" name="lg" class="form-control"/>
    </div>
    <div class="form-group">
        <label for="mdp"></label>
        <input type="password" name="motdepasse" class="form-control"/>
    </div>
    <button type="submit">Valider</button>
</form>
```

Ce code peut être *copier/coller*.