GSB avec Angular2, partie 2

Routage, gestion d'un menu, l'accès aux données, notion de service

Introduction

Dans cette partie nous allons aborder deux notions nouvelles, le routage et l'accès aux données en utilisant un service dédié : un service REST.
Vous pouvez utiliser le dépôt, disponible ici : https://github.com/patricegrand/GSBAngular2_V2.0 correspondant à la correction de la première partie.

1) Le routage

Le routage consiste à lier une route (un chemin) à l'exécution de code.

Commençons tout d'abord à ajouter plusieurs nouveaux composants (*component*) afin de pouvoir profiter du routage. Nous désirons obtenir l'enchainement suivant :

Ainsi, trois composants doivent être créés :

- une barre de navigation (navbar),
- un composant qui n'affiche que « Les médecins »,
- un composant qui n'affiche que « Les visites ».

Concernant la navbar, on vous fournit le code du fichier *app.navbar.html*, qui utilise des classes Bootstrap :

```
<nav class="navbar">
    <ul class="nav navbar-nav" style="display:flex; flex-direction: row;
justify-content: center">
        <li><a>Gestion des rapports</a></li>
        <li><a>Gestion des médecins</a></li>
    </ul>
</nav>
```

Remarque : ce code peut être *copier/coller.*

Rappel : la création d'un composant nécessite, **dans notre choix d'architecture**, les étapes suivantes :

1. Création des fichiers « ts » et « html » (éventuellement « css ») dans un répertoire dédié au composant.
2. **Déclaration du composant** dans *app.modules.ts*.

Pour les trois nouveaux composants, il a été décidé de s'appuyer sur l'arborescence suivante :

navbar
    app.navbar.component.ts
    app.navbar.html
medecins
    app.medecins.component.ts
    app.medecins.html
visites
    app.visites.component.ts
    app.visites.html

Pour chaque composant, la propriété *selector* doit être renseignée (voir partie 1) ; on envisage les trois valeurs suivantes :

- NavbarCompent; selector : my-navbar.
- MedecinsComponent; selector : my-medecins.
- VisitesComponent; selector : my-visites.

Travail à faire

Créez ces trois composants en vous inspirant du composant *ConnexionComponent* décrit dans la partie 1 ; ajoutez-les au module.
Vérifiez qu'il n'y ait pas d'erreur de syntaxe en lançant la compilation de votre projet.
(le bon fonctionnement de ces composants ne pourra être testé qu'une fois l'authentification opérationnelle)

Passons maintenant à la partie routage qui va nous permettre d'enchaîner des composants.

Avec Angular 2 le routage peut être déporté dans un module spécifique ou simplement mis en œuvre dans un module existant ; c'est cette deuxième option que nous allons mettre en œuvre.

1.1) La déclaration des routes dans le module

Au préalable, pour utiliser le mécanisme de routage qui ne doit pas faire un appel au serveur (pas de rechargement de page), il faut ajouter dans index.html le paramétrage :

```
4   <head>
5       <base href="/">
6       <title>GSB gestion des rapports de visite</title>
```

Dans le module App, les routes sont simplement déclarées dans un tableau, la suite des *import* :

```
import { MedecinsComponent }  from './medecins/app.medecins.component';
import { VisitesComponent }  from './visites/app.visites.component';
import { NavbarComponent }  from './navbar/app.navbar.component';


const appRoutes: Routes = [
  { path: '', component: ConnexionComponent },
  { path: 'medecins', component: MedecinsComponent },
  { path: 'visites', component: VisitesComponent },
  { path: 'accueil',  component: NavbarComponent  }
];
```

Le code peut être *copier/coller*.

Ce tableau de type *Routes* est composé d'objets dont deux propriétés sont ici valorisées :

- **path** : c'est le chemin qui sera utilisé
- **component** : c'est le composant associé à ce chemin

On peut interpréter une route ainsi :

« *Si l'url est localhost/medecins, le composant à charger est MedecinsComponent* »

Bien sûr, pour pouvoir utiliser la classe *Routes*, il faut l'importer (fichier app.module.ts) :

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

Le code peut être *copier/coller*.

On importe aussi *RouterModule* pour la suite.

En effet, le tableau est fourni à la classe *RouterModule* grâce à la méthode *forRoot* qui doit donc être importée dans le décorateur du module:

```
26  @NgModule({
27    imports:      [ BrowserModule,FormsModule,RouterModule.forRoot(appRoutes) ],
```

Une dernière question concerne le chargement du composant : à quel endroit est-il chargé dans la page ? Et bien une balise spécifique est utilisée, *<router-outlet>* :

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                <router-outlet></router-outlet>`
6   })
7   export class AppComponent {
8
```

C'est à cet endroit que sera inséré, à la demande, le composant pointé par la route.

Travail à faire

Mettez en œuvre le routage des 3 composants. Testez que vous obtenez bien les formulaires présentés plus haut, en saisissant directement dans le navigateur chacune des routes (par exemple *localhost/accueil* doit faire apparaître uniquement le menu).
Attention, afin de faire apparaître la navbar dans chaque composant, il faudra l'ajouter dans chaque fichier HTML !

1.2 Appel des routes dans les pages

Bien sûr, ces routes doivent être appelées dans les pages, souvent à partir d'un menu (notre navbar) ; insérons donc les routes dans la navbar :

app.navbar.html

```
<nav class="navbar">
    <ul class="nav navbar-nav" style="display:flex; flex-direction: row; justify-content: center">
        <li><a routerLink = "/visites" >Gestion des rapports</a></li>
        <li><a routerLink = "/medecins" >Gestion des médecins</a></li>
    </ul>
</nav>
```

Dans la balise <a, nous ajoutons la propriété *routerLink* qui pointe sur les routes définies dans le tableau de routes.

Mais une route peut être aussi appelée directement dans le code ; ainsi pour faire apparaître le menu uniquement il faudra le faire dans le code du *ConnexionComponent* dans le cas où les données saisies sont correctes. Pour cela la classe Router propose une méthode :

```
this.router.navigate(['accueil']);
```

Nous abordons ainsi un aspect très important dans Angular 2 : la classe ConnexionComponent a besoin d'un objet de type Router mais elle n'en possède pas encore ; on va *injecter* un objet de la classe Router :

```
export class ConnexionComponent {


        lblLogin: string="Login";
        lblMdp: string="Mot de passe";
        titre: string = "connexion";
        login: string;
        mdp: string;
        estCache: boolean=true;
        lblMessage: string ="";
        constructor(private router : Router){

        }
```

L'injection se réalise par l'intermédiaire des arguments du constructeur ! La présence de private (ou public) demande à Angular 2 de créer un champ nommé *router*, de type *Router* ; le *new* se fera automatiquement. La présence de private (ou public) est obligatoire et ceci afin de distinguer l'injection d'un simple argument.
Dans le code, on accèdera au champ par la syntaxe classique *this.route*.

L'injection est un mécanisme très largement partagé par de nombreux frameworks ; il nous affranchit d'instancier des objets dans la très grande majorité des situations.
Bien sûr, le fichier doit connaître la classe *Router* ; il faut donc faire un import au préalable :

```
3    import { Router } from '@angular/router';
```

Vous êtes maintenant prêt à gérer l'enchainement des components.

Travail à faire

Complétez la classe ComponentConnexion (et notamment la méthode « valider ») afin de rediriger l'utilisateur sur la page d'accueil dans le cas où les données d'authentification saisies sont correctes.
Rappel : pour le moment seul le compte « toto » avec le mot de passe « titi » est valide.

2) L'utilisation des données, le service REST

Commencer par lire l'annexe 1 qui propose un rappel sur les services REST ainsi que la ***présentation du service GSB et son installation***.

2.1) Utilisation d'un service pour consommer le service REST

Pour « consommer » le service REST fourni dans notre cas par le serveur Apache avec le port 80 par défaut, nous allons créer une classe dédiée, un **service** dans un fichier ***app.service.data.ts*** :

Un service est une simple classe, décorée vous l'imaginez bien par un décorateur qui précisera sa fonction :

```
1  import { Injectable } from '@angular/core';
2  import { Http, Response } from '@angular/http';
3  import{ Observable } from 'rxjs/Rx';
4  import 'rxjs/add/operator/map';
5
6  @Injectable()
7  export class DataService {
8      private  urlDomaine :  string = "http://localhost/restGSB";
9      constructor(private http: Http) {}
10     public connexion( loginIn : string, mdpIn : string ) :
11 Observable<string[]> {
12         let url :string = this.urlDomaine + "/connexion?login=" +
13 loginIn + "&mdp=" + mdpIn;
14         let req = this.http
15                     .get(url)
16                     .map((r: Response)=>r.json());
17         return req;
       }
   }
```

Ce code peut être *copier/coller*.

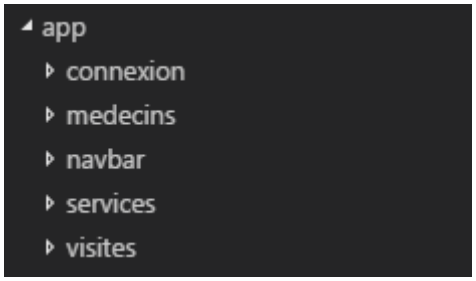
Remarques :

- Ligne 6, la classe (service) DataService n'est décorée que par le décorateur (fonction sans argument) **Injectable.** Ceci va permettre *d'injecter* un objet de cette classe dans n'importe quelle classe ; voir plus haut les commentaires sur l'injection
- Ligne 9, nous sommes en présence justement d'une injection d'un objet http de type Hhttp ;
- Ligne 8, l'adresse du service REST est indiquée
- Ligne 10, la méthode connexion met en œuvre un mécanisme proche de la programmation événementielle : création d'une requête, *req* ligne 12 qui retourne un *Observable*, mais cette requête n'est pas exécutée encore ; pour cela il faudra s'y *abonner*. Nous verrons plus loin comment on peut le faire. Notons que ces requêtes sont *asynchrones* et donc non bloquantes (l'application peut se poursuivre en attente des données envoyées pas le service REST). C'est la bibliothèque *rxJs* qui réalise ce travail ; elle ne fait pas partie d'Angular2 et est utilisée dans d'autres Framework JavaScript.
- Ligne 12, l'url pour atteindre le service REST est construite
- Ligne 15, la méthode *get* est définie
- Ligne 16, la méthode *map* (importée en ligne 4), précise le type de format de réception de la réponse. **L'argument de map est une fonction**, Nous sommes en présence d'une syntaxe un peu particulière pour définir cette fonction : une **expression lambda**. Ceci ressemble à la méthode anonyme (sans nom) : *map(function(r : Response){ return r.json() ;})*.

Les expressions lambda (empruntées à la programmation fonctionnelle) ou fonctions lambda (lorsque le code est plus conséquent) deviennent très courantes aujourd'hui ; elles ont l'avantage de la concision.

- Lignes 14, 15 et 16, chaque instruction retourne un Observable sur lequel est appliquée une nouvelle méthode ; tout ceci pouvait être écrit sur une même ligne, la disposition proposée est guidée par un souci de clarté.

Il est préférable de regrouper les services dans un répertoire dédié, nommé ici *services* :



Travail à faire

- Créer le répertoire *services*.
- Ajouter un fichier *app.service.data.ts* dans lequel vous collerez le code du service présenté plus haut.
- Indiquer au module qu'il dispose d'un **service** dans la partie *providers* de son décorateur (sans oublier d'importer son fichier) et qu'il importe HttpModule, sans oublier non plus d'ajouter un import :

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

```
23  @NgModule({
24    imports:      [ BrowserModule,FormsModule,HttpModule,RouterModule.forRoot(appRoutes) ],
25    declarations: [ AppComponent,ConnexionComponent,NavbarComponent,MedecinsComponent,VisitesComponent ],
26    providers: [DataService],
27    bootstrap:    [ AppComponent ]
28  })
```

Il ne reste plus qu'à gérer le component de connexion en appelant le service :

```
22          lblMessage: string ="";
23          visiteur: any;
24
25      constructor(private dataService : DataService, private router : Router ){}
26      valider():void{
27              this.dataService.connexion(this.login,this.mdp)
28                                      .subscribe(
29                                          (data)=>{/*Code à écrire*/}
30                                          ,(error)=>{/*Code à écrire*/}
31                                                  );
32       }
33  }
34
```

- Ligne 23, un champ *visiteur* a été ajouté afin de le valoriser grâce aux données transmises par le service REST (certaines données du visiteur) ; le type *any* permet de déclarer tout type.
- Ligne 25, le constructeur injecte le service
- Ligne 27, l'objet Observable retourné par la méthode *connexion* du DataService est récupéré afin de s'abonner à l'aide de la méthode *subscribe* qui prend (ici) deux arguments : des fonctions lambda. La première présente le code (à compléter) qui s'exécute lorsque les données sont récupérées du service REST ; la deuxième contient le code (à compléter) qui s'exécute lorsqu'une erreur est retournée par le service REST.

Travail à faire

Modifier la méthode *valider* et compléter le code proposé ci-dessus qui devra :
- pour la première fonction, ligne 29, d'une part récupérer les données de l'argument *data* et router vers l'accueil d'autre part
- pour la deuxième fonction, ligne 30, gérer l'affichage du message d'erreur.

Vous testerez en vous connectant avec un compte visiteur ; un compte de test est présent dans la base : aribiA/aaaa

La correction est disponible ici :
https://github.com/patricegrand/GSBAngular2_V2.1

Annexe 1 : les services REST

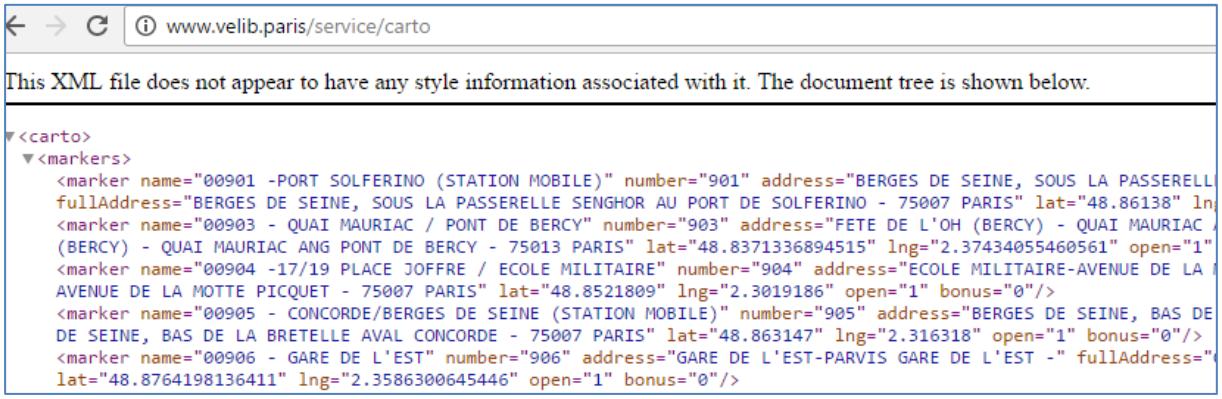
Nous ne reviendrons pas sur la définition d'un service REST, Wikipedia ou tout autre site propose des présentations suffisantes si vous découvrez cette notion.
Nous allons simplement illustrer cette architecture à l'aide d'exemples.

2.a.1 Le service Velib de la ville de Paris

La ville de Paris propose deux URL :

- L'une permettant de visualiser, au format XML, l'ensembles des stations :

http://www.velib.paris/service/carto



Chaque station est identifiée par une propriété *number* (la première vaut 901, par exemple) et sa localisation (adresse et géolocalisation) est fournie ; d'autre propriétés sont présentes.

- L'autre fournit des informations sur l'état actuel de la station, par exemple pour la station dont la propriété *number* est 901 :

http://www.velib.paris/service/stationdetails/901

On peut connaître ainsi, le nombre de places totales (20), le nombre de vélos disponibles (2) entre autres.

Toutes les applications Velib (SmartPhone souvent) utilise ce service REST. Il en est de même pour de nombreuses autres applications (météorologique, géographique, Amazon,…).

2.a.2 Notre service REST GSB

La base de données est disponible dans le fichier *gsbrapports.sq*l.

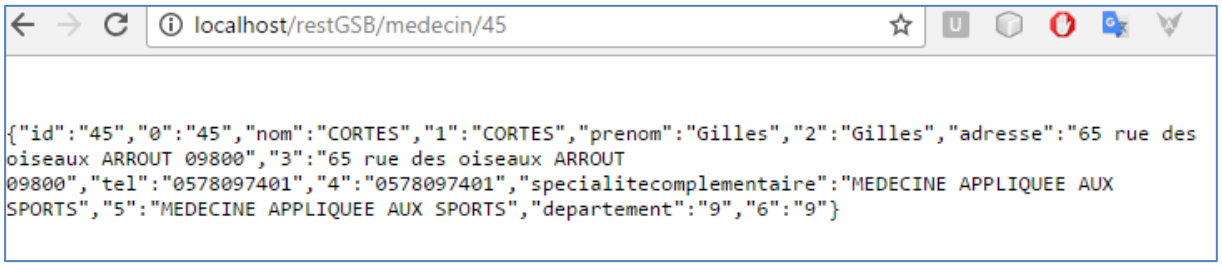
Le code du service REST est disponible ici : https://github.com/patricegrand/restGSB.git

Travail à faire

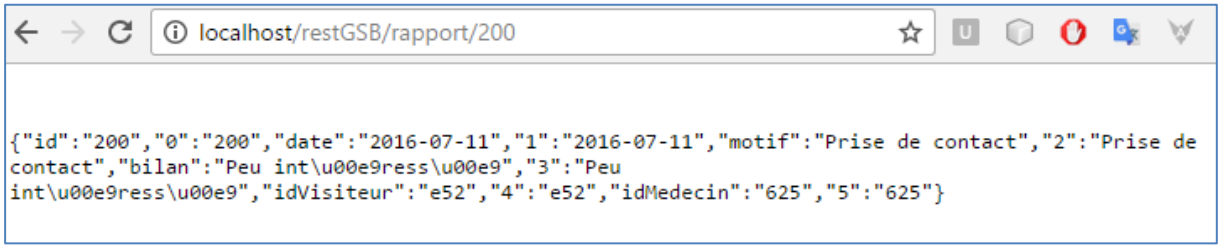
Créer une base de données ***gsbrapports*** ; importer le script sql qui se trouve dans le fichier *gsbrapports.sql*.
Copier le code du service REST à la racine de répertoire de publication (www) local ou distant.

Nous pouvons maintenant utiliser ce service à partir d'URL ; voici quelques exemples :

- Les informations sur le médecin d'id 45.

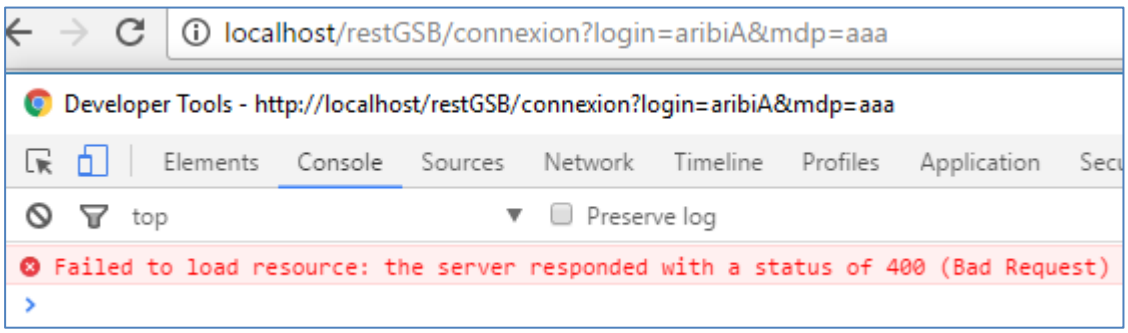


- Les informations sur le rapport d'id 200.



- Les informations d'un visiteur dont on fournit le login et le mot de passe.

- Un retour de la classe 400 si les informations de connexion ne sont pas valides.



Remarque : le format retourné par ce service est JSON (le format pour le service Velib était XML).

De nombreuses autres requêtes sont utilisables dans le service REST fourni mais il n'est pas nécessaire de les connaître. Il suffit lorsque l'on utilise un service REST de connaître :

- le format de l'URL à utiliser,
- le format des données de la réponse retournée.