Découverte du Framework Angular JS avec le contexte GSB

Partie 5 : pour aller un peu plus loin…

Description du thème

Propriétés	Description
Intitulé long	Découverte du Framework Angular JS avec le contexte GSB dans sa partie gestion des rapports de visite
Formation concernée	BTS SIO option SLAM
Matière	SLAM 4
Présentation	Accompagnement dans la découverte d'Angular. Développement pas à pas d'une application à partir du contexte GSB
Notions	• D4.1 - Conception et réalisation d'une solution applicative • D4.2 - Maintenance d'une solution applicative Savoir-faire • Programmer un composant logiciel • Exploiter une bibliothèque de composants • Adapter un composant logiciel • Programmer au sein d'un Framework
Prérequis	Les principes du développement web, PHP, SQL, JavaScript
Outils	SGBD MySQL, un environnement de développement
Mots-clés	GSB, Angular JS, Ajax, MVVM
Durée	6 heures
Auteur(es)	Patrice Grand. Relectures de Cécile Nivaggioni, Yann Barrot, Luc Frebourg et Gaëlle Castel.
Version	v 1.0
Date de publication	novembre 2016

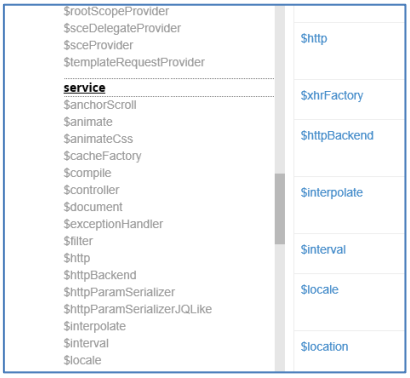
Présentation

Nous avons vu, au cours des 4 premières parties, les notions principales d'Angular JS : les directives, le modèle MVV-M, le binding, l'injection de dépendance. Elles nous ont permis de développer notre application.
Nous allons explorer deux notions, celle de *service* et celle de *directive* mais côté *fournisseur* et pas seulement côté *consommateur*, comme cela a été le cas jusqu'ici.

L'application de départ est dans le répertoire gsbAJSV5.0 ; elle est identique à la version 4.1.

I Notion de service

Vous avez eu l'occasion d'utiliser des services. Le site ***Angular.org*** présente tous les services disponibles dans son API :



Vous avez utilisé les services *$location*, *$http* et *$rootScope*. Techniquement un service est **un objet** avec des propriétés et des méthodes. Mais c'est un objet un peu particulier car il est unique ; ainsi si vous l'utilisez à plusieurs endroits dans votre code, vous manipulez le même objet (on dit à son propos que c'est un *singleton*).
Pour utiliser cet objet, vous l'injectez au moment où vous en avez besoin ; le Framework s'occupe des dépendances et de la construction, pratique, non ?

Bref, construire son propre service, c'est définir un objet, avec ses propriétés et ses méthodes et lui faire bénéficier du mécanisme de l'injection. Angular vous facilite la tâche et vous propose des types de services par défaut. Trois sont proposés, la *factory*, le *provider* et …le *service*. Les trois sont très proches et ne se distinguent que par la manière de les utiliser et donc de définir le code de l'objet.

C'est à la déclaration que le choix se fait :

- *app.factory('maFactory', function(){ …}) ;*
- *app.provider('monProvider', function(){ …}) ;*
- *app.service('monService', function(){ …}) ;*

Pour utiliser ce service, il suffira de l'injecter dans le code comme pour tout service :

app.controller("monController", function(monProvider){…}) ;

Nous allons nous contenter de la *factory*.

1 Un service simple : gestion des dates

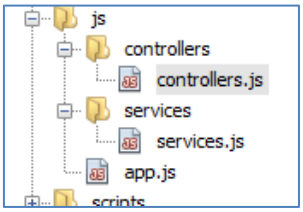
A deux reprises, nous avons dû convertir une date afin de la rendre compatible pour un appel Ajax :

```
var dateRapport = $scope.r.date;
var jour = dateRapport.getDate();
var mois = dateRapport.getMonth() + 1 ;
var annee = dateRapport.getFullYear();
var date = annee+'-'+mois+'-'+jour;
```

Nous allons en faire un service, à l'aide d'une *factory*.

1.1 Organisation du code

Ajoutons un répertoire *services* à notre projet, ainsi qu'un nouveau fichier *services.js* :



N'oublions pas d'ajouter ce fichier à notre projet :

```
<script src="js/app.js"></script>
<script src="js/controllers/controllers.js"></script>
<script src="js/services/services.js"></script>
```

Nous voici prêt à écrire notre *factory*.

1.2 Le code de notre factory

Dans le fichier *services.js*, le code doit avoir l'architecture suivante :

```
app.factory('maDateFactory',function(){
        var date = {

          // code de l'objet

                }

        return date
});
```

- ligne 25, on donne un nom à cette *factory*, qui sera le nom à utiliser dans le code,
- ligne 26, on déclare un objet date,
- c'est dans les lignes suivantes qu'il faudra écrire le code de l'objet,
- une *factory* retourne obligatoirement un objet, ligne 33.

Le code terminé :

```
24
25  app.factory('maDateFactory',function(){
26          var date = {
27                  get : function(uneDate){
28                          var jour = uneDate.getDate();
29                          var mois = uneDate.getMonth() + 1 ;
30                          var annee = uneDate.getFullYear();
31                          var newDate = annee + '-' + mois + '-' + jour;
32                          return newDate;
33                  }
34          }
35          return date
36  });
```

- ligne 27, l'objet *date* ne possède qu'une seule méthode, *get*, qui a un argument (ici nommé *uneDate*).
- le code est le même que celui rappelé plus haut, en utilisant l'argument *uneDate.*
- la ligne 32 précise que la méthode **get retourne la chaîne** contenue dans *newDate*.

1.3 Appel de la factory

Dans le code appelant, nous aurons bien sûr l'injection de la *factory* dans le contrôleur (si c'est un contrôleur qui est concerné) :

```
app.controller('choisirRapportController',function($scope, $http, maDateFactory){
        $scope.btnVisible = true;
```

Au niveau de l'appel :

```
193     $scope.changementDate = function(){
194         $scope.message  ="";
195         var dateRapport = $scope.dateRapport;
196         var date = maDateFactory.get(dateRapport);
197
```

- ligne 196, l'appel de la *factory* se fait en utilisant son nom et sa méthode *get*.

Travail à faire

- Créer la *factory* « maDateFactory », puis modifier le contrôleur *choisirRapportController* afin qu'il fasse appel à cette dernière.
- Modifier le contrôleur *nouveauRapportController* afin qu'il utilise la *factory* « maDateFactory ».

2 Un autre service simple : remplacer la propriété du rootScope

Lors de la gestion des médecins nous devions, dans la partie 3, mémoriser le médecin sélectionné afin de l'attacher aux deux sous-menus :

Cette solution avait le défaut d'utiliser le *rootScope* et donc de faire bénéficier tous les scopes d'une propriété (un médecin) qui ne les concernait peut-être pas tous ; cette utilisation du *rootScope* (sorte de poubelle à variables *globales*), même si elle fonctionne bien, n'est pas à privilégier.

Nous pouvons ainsi *refactorer* le code en question en utilisant cette fois un nouveau service à l'aide d'une factory.

Le service sera fonctionnellement très simple : un objet (le médecin) avec un *get/set* sur le champ privé.

2.1 Création de la factory

Commençons par ajouter une *factory* dans le fichier *services.js*.

```
39
40  app.factory('monMedecinFactory', function(){
41      var medecin = {
42                      _medecin : null,
43                      set : function(leMedecin){
44                              this._medecin = leMedecin;
45                          },
46                      get : function(){
47                          return this._medecin;
48                      }
49                  };
50      return medecin;
51  });
```

- Ligne 40, nous créons la *factory*, dont le nom est *monMedecinFactory,*
- Ligne 41, nous déclarons un objet local *medecin* que nous retournerons en ligne 50,
- L'objet local contient 3 propriétés :
  - un champ *_medecin,*
  - une méthode *set* pour valoriser ce champ,
  - une méthode *get* pour le retourner.

  Notez que les propriétés sont séparées par une virgule.

2.2 Utilisation de la factory

Nous injectons la *factory* dans chaque contrôleur qui va l'utiliser :

```
45  /*----------------------Medecins-----------------------------*/
46
47  app.controller('medecinsController',function($scope,$http, monMedecinFactory){
48
```

- Nous avons supprimé l'injection du *rootScope*, qui ne servira plus.

Dans le code, nous remplaçons le code existant par notre appel de la *factory* :

```
3       $scope.choisirMedecin = function(medecin){
4               $scope.rechercheMedecin.nom = medecin.nom + " "+medecin.prenom;
5               $scope.medecins ={};// vide la liste;
6           //   $rootScope.medecin = medecin; //enregistrement pour communiquer a
7               monMedecinFactory.set(medecin);
8               $scope.appelTel = "tel:" + medecin.tel;
9
0           } ;
1   });
```

Nous pouvons maintenant, dans les contrôleurs qui en ont besoin, récupérer le médecin :

```
83  /*----------------------------------Controleur MajMedecin-----------------------------*/
84  app.controller('majMedecinController',function($scope, $http,$location, monMedecinFactory){
85      //   var medecin = $scope.medecin; // héritage du rootScope
86      var medecin = monMedecinFactory.get();
87      if(medecin == undefined)
88          $location.url("medecins");
```

- Ligne 84, injection de la *factory,*
- ligne 86, appel pour récupérer le médecin grâce à la méthode get.

Remarque :

Si le mécanisme peut fonctionner c'est parce que, comme nous l'avions indiqué plus haut, il n'y a **qu'un seul objet *monMedecinFactory* de manipulé** !! Nous sommes bien en présence d'un *singleton*.

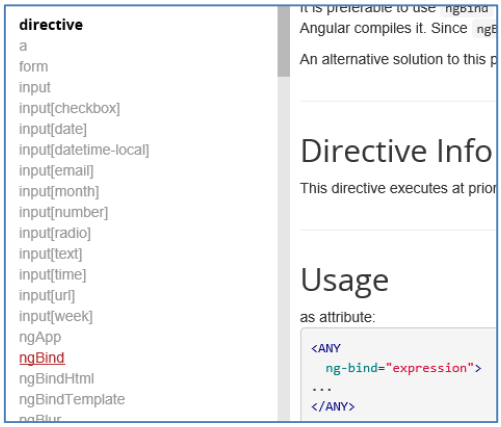
Travail à faire
Procédez à la création de la *factory* comme indiqué ci-dessus ; mettez-la en œuvre dans le code afin de remplacer l'utilisation de la propriété du *rootScope* par la *factory* dans la gestion des médecins.

II Les directives

1 Généralités

Nous avons très rapidement pris connaissance des directives sous Angular JS ; en effet, leur emploi est obligatoire pour définir le contour de l'application (*ng-app*), de la zone d'insertion des vues *ng-view*, etc…
Une directive permet d'enrichir le comportement d'un élément HTML ; par exemple la directive *ng-repeat* itère sur l'élément HTML qui la contient ou *ng-show* permet de masquer/montrer l'élément HTML.
Une directive doit donc être invoquée dans un élément HTML ou même être un nouvel élément HTML.

De très nombreuses directives sont disponibles dans le noyau d'Angular ; sur le site angular.org, on peut les consulter :



Remarquons que le nom JavaScript de la directive ne correspond pas à son nom HTML ; par exemple *ngBind* est le nom JavaScript et *ng-bind* son nom HTML (ceci est lié à HTML 5). Ceci n'a pas une importance particulière mais il faut le savoir. L'usage veut qu'une directive ait un préfixe (*ng* pour celles proposées par Angular) et un suffixe qui précise son rôle ; le tout respectant la *camelCase* (le premier mot est en minuscule, le second (et troisième éventuellement) commence par une majuscule.

Ainsi les directives suivantes sont en règle avec l'usage et avec HTML5 :

JavaScript HTML

myNavbar ----------→ my-navbar
myMenuCommun----→ my-menu-commun
formulaire ----------→ formulaire

L'écriture des directives est à mon avis la partie la plus délicate sous Angular, principalement parce qu'il y a de nombreux paramètres à prendre en compte. Tous ne sont pas obligatoires mais il est bon d'en connaitre l'existence. Angular.org présente dans son onglet *DeveloperGuide* la documentation officielle permettant de créer ses propres *directives*.

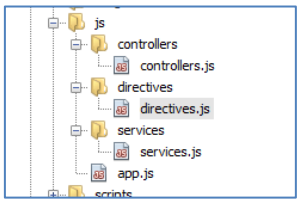
Mais nous ne sommes pas obligés d'écrire de nouvelles directives ; ceci arrive surtout lorsque nous constatons que la couche de présentation (associée à HTML) propose du code très répétitif.

Justement ☺ c'est notre cas concernant la gestion des menus. Nous avons certes déporté les menus et sous-menu dans un fichier distinct *menuCommun.html* qui charge *menuMedecins.html* ou *menuRapports.html* mais la partie JavaScript reste lourde et répétitive ; ainsi tous les contrôleurs commencent par à peu près les mêmes lignes d'initialisation :

```
$scope.titre = "Gestion des visites";
$scope.btnVisible = true;
$scope.isCollapsed = true;
$scope.srcMenu = "vues/menuRapports.html";
$scope.menu = function(){
                $scope.isCollapsed = !$scope.isCollapsed;
              };
```

Bref, écrire une directive nous rendra bien service.

Il faudra commencer par ajouter un répertoire et un fichier dédié aux directives écrites :



N'oublions pas d'ajouter ce fichier :

```
<script src="js/app.js"></script>
<script src="js/controllers/controllers.js"></script>
<script src="js/services/services.js"></script>
  <script src="js/directives/directives.js"></script>
```

2 Ecriture de la directive

L'objectif est donc d'encapsuler le code présenté plus haut dans la directive.
Pour commencer, nous déclarons cette directive dans le fichier directives.js :

```
app.directive("monMenu",function(){
```

Ainsi, son appel html sera *mon-menu ;* on a le choix entre :

- *<mon-menu> </mon-menu>* : version élément HTML
- ou <div*[p ou autre] mon-menu* > : version attribut

Il est possible de limiter le choix à un seul type d'appel grâce à la propriété *restrict*. Notre directive bénéficiera des deux types d'appel.

Le code de la directive doit retourner un objet qui contient différentes propriétés, presque toutes optionnelles.

2.1 Une directive très simple :

```
app.directive("monMenu",function(){

    return{
        templateUrl : "vues/menuCommun.html" ,

    };
});
```

Elle retourne un objet qui ne contient qu'une seule propriété *templateUrl*, fichier associé à la directive.

Si on remplace l'inclusion du fichier

```
<div ng-include src = "'vues/menuCommun.html'" ></div>
```

par la directive *mon-menu* ; l'ensemble continue à fonctionner mais on n'y a pas gagné beaucoup. En fait la directive reste prisonnière de son contexte ; il faut gérer toutes les propriétés associées à la vue menuCommun.htm à l'intérieur de la directive. Et c'est possible.

2.2 Une directive fonctionnelle

Nous allons faire communiquer la directive avec les propriétés de la vue :

```
app.directive("monMenu",function(){

    return{
        templateUrl : "vues/menuCommun.html" ,
        scope : {
            titre : "@",
            btnVisible :"=",
            isCollapsed :"=",
            srcMenu : "@"
            },
```

Une seconde propriété est ajoutée à l'objet retourné, « *scope* ». C'est un objet pour lequel on définit 4 propriétés correspondant aux 4 propriétés de la vue :

```
<table>
    <tr>
        <td><button type="button" class="btn btn-default btn-lg btn-circle"
                    ng-show="btnVisible" ng-click="menu()">
                <span class="glyphicon glyphicon-align-justify"></span>
            </button></td>
        <td><h3>{{titre}}</h3></td>
    </tr>
</table>
<!-- menu déroulant-->
<div  uib-collapse="isCollapsed" >
    <div class="well well-lg">
        <div ng-include src="srcMenu"></div>
    </div>
</div>
```

Pour chaque propriété dans la directive, on indique le type de liaison :

- ''@'' : pour indiquer que la propriété est une chaîne de caractère
- ''='' : dans les autres cas

Ces propriétés définies dans le scope de la directive jouent le rôle d'argument qu'il faudra valoriser dans le code HTML ; par exemple :

```
<mon-menu titre="Gestion des médecins"
        btn-visible="true"
        is-collapsed="true"
        src-menu = "vues/menuMedecins.html">
</mon-menu>
```

Ainsi, on encapsule bien les propriétés du scope à l'intérieur de la directive ; sympathique, non ?

Il nous reste à gérer *ng-click = menu()*

Il est possible d'ajouter du code fonctionnel à notre directive :

```
app.directive("monMenu",function(){

    return{
        templateUrl : "vues/menuCommun.html" ,
        scope : {
            titre : "@",
            btnVisible :"=",
            isCollapsed :"=",
            srcMenu : "@"
            },
        link : function($scope){
            $scope.menu = function(){
                                $scope.isCollapsed = !$scope.isCollapsed;
                            };
        }
    };
});
```

La propriété *link* de la directive permet d'appeler une fonction anonyme qui gère la propriété *menu*.

Ainsi donc, le code suivant devient inutile :

```
app.controller('medecinsController',function($scope,$http, monMedecinFa

//    $scope.titre = "Gestion des médecins";
 //   $scope.btnVisible = true;
//    $scope.isCollapsed = true;
//    $scope.srcMenu = "vues/menuMedecins.html";
//    $scope.menu = function(){
//                        $scope.isCollapsed = !$scope.isCollapsed;
//                    };
```

Il est remplacé par l'appel de la directive :

```
<!--<div ng-include src = "'vues/menuCommun.html'" ></div>-->
<mon-menu
    titre="Gestion des médecins"
    btn-visible="true"
    is-collapsed="true"
    src-menu = "vues/menuMedecins.html">
</mon-menu>
```

Remarque : vous pourrez, sous NetBeans, obtenir un message d'erreur sur cette directive ; nous avions expliqué dans la partie 3 les problèmes de compatibilité entre cet EDI et AngularJS. Vous pouvez lever cette erreur en forçant l'EDI à accepter cette nouvelle balise (clic droit sur l'erreur).

Travail à faire

- Créer la directive « monMenu ».
- Revisiter (*refactoring*) le code de l'application en utilisant la directive créée.

Ceci termine notre présentation d'Angular ; le corrigé se trouve dans le répertoire **gsbAJSV5.1.**