Chapitre 4 : Le framework AngularJS

Objectifs :

- ✓ Développer des applications Web performantes avec AngularJS
- ✓ Maitriser les fonctionnalités clés du framework (filtres, contrôleurs, routes, templates...)

Prérequis : Maitriser la séquence 1

1. Introduction

AngularJS est né en 2009 dans les locaux de Google. Deux développeurs du nom de Brad Green et **Shyam Seshadri** commençaient sérieusement à déprimer devant leur projet appelé "Google Feedback". Une immense frustration les envahissait au fur et à mesure que leur code grandissait. Celui-ci comptait approximativement 17 000 lignes à ce moment-là. Autant de lignes de pur front-end qui sont instables et donc difficilement maintenables.

C'est à ce moment-là que **Shyam Seshadri** proposa de redévelopper entièrement la solution avec un framework fait maison. Au bout de trois semaines, l'application ne comptait plus que 1 500 lignes de codes, parfaitement testées.

À compter de ce jour, les autres développeurs de l'équipe ont décidé de prendre en main ce framework et de travailler avec au quotidien.

Il y a plusieurs frameworks JavaScript très populaires aujourd'hui : **Angular, React, Ember**, **…** les autres frameworks ont beaucoup de succès et sont utilisés sur des sites extrêmement bien fréquentés, React et Vue notamment. Angular présente également un niveau de difficulté légèrement supérieur, car on utilise le **TypeScript** plutôt que JavaScript pur ou le mélange JS/HTML de React. Ainsi, quels sont donc les avantages d'Angular ?

- ✓ **Angular est géré par Google :** il y a donc peu de chances qu'il disparaisse, et l'équipe de développement du framework est excellente.
- ✓ **Le TypeScript :** ce langage permet un développement beaucoup plus stable, rapide et facile.
- ✓ **Le framework Ionic :** le framework permettant le développement d'applications mobiles **multi-plateformes** à partir d'une seule base de code utilise Angular.

2. Concepts

Shyam Seshadri, en créant Angular à utiliser des concepts et des bonnes pratiques incontournables dans le monde du développement web actuel.

- ✓ **Architecture MVC (Modèle-Vue-Contrôleur) :** Consiste à avoir une stricte séparation entre les données (Modèle), la présentation des données (Vue), et les actions que l'on peut effectuer sur ces données (Contrôleur)
- ✓ **Data Binding :** Grâce à ce concept, les liens entre votre code HTML et JavaScript ne seront que plus forts. :p
- ✓ **Injection de dépendances :** tout comme l'architecture MVC, lorsque l'on parle d'injection de dépendances, on parle d'un concept prépondérant dans tout développement. Grâce à cela, les modules que vous développerez n'auront plus à se soucier d'instancier leurs dépendances.
- ✓ **La manipulation du DOM au moyen de directives :** la manipulation du DOM conduit souvent à la création de code difficilement maintenable et difficilement testable. Avec Angular, ce n'est plus le cas

3. Installation de l'environnement

3.1. Nodejs

Grâce à Node, vous aurez la possibilité de créer un petit serveur qui aura pour rôle de "servir" les fichiers (**JavaScript, HTML, CSS ...)**
Lorsque vous installez Node, vous installez également **NPM** qui est le gestionnaire de paquets (Package manager)). Grâce à cela, lorsque vous aurez besoin d'installer un module Node, une seule commande vous permettra d'installer le module ainsi que TOUTES ses dépendances.

3.2. Yeoman

Yeoman est un outil qui va permettre d'initialiser nos projets en nous créant une structure projet efficace mais également en nous installant l'ensemble des librairies qui nous seront nécessaires au cours de nos développements.

Yeoman comprend trois outils essentiels lorsque vous développez vos projets JavaScript:

- ✓ **Yo :** C'est un outil qui va permettre de mettre en place un projet.

- ✓ **Bower :** il s'agit d'un autre gestionnaire de paquets, au même titre que **npm**. Ces deux gestionnaires de paquets ne gèrent pas les dépendances de la même manière. À titre d'explication, **npm** gère automatiquement les dépendances des paquets, alors qu'il incombe au développeur de décrire les dépendances lorsque vous utilisez **bower**. De ce fait, il est assez courant d'utiliser **npm** lorsque vous programmez un serveur et **bower** pour les projets front-end.
- ✓ **Grunt :** C'est un outil qui permet de tester et lancer une application.

Installation de **bower**

```
npm install -g bower
```

Installation de **yo** et de quelques générateurs

```
npm install -g yo
```

L'option **-g** afin permet de rendre l'installation globale. Cela évitera d'avoir à le réinstaller à chaque nouveau projet

Afin de pouvoir créer des applications web, il est nécessaire d'installer au préalable des "générateurs". Dans notre cas, comme nous souhaitons créer des applications Angular, il est donc nécessaire d'installer le générateur d'applications Angular:

```
npm install -g generator-angular
```

4. Création de notre premier projet

Une fois que l'environnement de travail est bien installé la création d'un projet devient très facile. Ainsi pour mieux organiser nos projet il est conseillé de créer un répertoire de travail puis de mettre l'ensemble des projet Angular dans ce répertoire. Chaque projet Angular sera dans son propre répertoire. Dans ce cours Nous créons le repertoire « projetAngular » dans lequel nous mettrons tous no projet Angular.

```
mkdir  projetAngular
```

On va maintenant créé le répertoire pour notre première application

```
cd  projetAngular
```

```
mkdir  premierApp
```

Faut se placer dans le répertoire de l'application

```
cd   premierApp
```

Ensuite vous pouvez utilisé **yo** pour initialiser notre première application :

```
yo   angular
```

5. la structure d'un projet

5.1. Les fichiers de configurations

On dénombre 3 fichiers de configurations qui se trouvent TOUS à la racine de votre projet:

- ✓ **package.json**
- ✓ **bower.json**
- ✓ **Gruntfile.js**

5.1.1. Le fichier Package.json

Il s'agit du fichier de configuration utilisé par **npm.** Le but de ce fichier est de décrire notre application. On y retrouve notamment le nom et la version courante de notre application. On retrouve également les dépendances nécessaires au bon fonctionnement de notre application.

```
{
 "name": "firstapp",
 "version": "0.0.0",
 "dependencies": {},
 "devDependencies": {
  "grunt": "^0.4.1",
  "grunt-autoprefixer": "^0.7.3",
  "grunt-concurrent": "^0.5.0",
  ...
  "grunt-wiredep": "^1.7.0",
  "jshint-stylish": "^0.2.0",
  "load-grunt-tasks": "^0.4.0",
  "time-grunt": "^0.3.1"
 },
 "engines": {
  "node": ">=0.10.0"
 }
}
```

Dans le champ **"engine"**, on décrit la version de node nécessaire pour que notre application fonctionne.

5.1.2. Le fichier bower.json

C'est dans ce fichier de configuration que nous décrivons nos dépendances purement front-end. Par exemple, **Angular** constitue ici une dépendance.

```
{
 "name": "first-app",
 "version": "0.0.0",
 "dependencies": {
  "angular": "~1.2.0",
  "json3": "~3.3.1",
  "es5-shim": "~3.1.0"
 },
 "devDependencies": {
  "angular-mocks": "~1.2.0",
  "angular-scenario": "~1.2.0"
 },
 "appPath": "app"
}
```

5.1.3. Gruntfile.js

Grunt est un outil qui a pour but de fournir au développeur le moyen d'automatiser les tâches répétitives comme la compilation, l'exécution des tests unitaires...

Le fonctionnement de **Grunt** n'est pas magique. Il convient de décrire l'ensemble des tâches à effectuer dans le fichier Gruntfile.js. A la fin du fichier que l'ensemble des tâches sont décrites précisément :

```
grunt.registerTask("serve", "Compile then start a connect web server", function
(target) {
    if (target === "dist") {
      return grunt.task.run(["build", "connect:dist:keepalive"]);
    }
    grunt.task.run([
      "clean:server",
      "wiredep",
      "concurrent:server",
      "autoprefixer",
      "connect:livereload",
      "watch"
    ]);
  });
  grunt.registerTask("server", "DEPRECATED TASK. Use the "serve" task instead",
function (target) {
    grunt.log.warn("The `server` task has been deprecated. Use `grunt serve` to start a
server.");
    grunt.task.run(["serve:" + target]);
  });
  grunt.registerTask("test", [
    "clean:server",
    "concurrent:test",
    "autoprefixer",
    "connect:test",
    "karma"
  ]);
  grunt.registerTask("build", [
    "clean:dist",
    "wiredep",
    "useminPrepare",
    "concurrent:dist",
    "autoprefixer",
    "concat",
    "ngAnnotate",
    "copy:dist",
    "cdnify",
    "cssmin",
    "uglify",
    "filerev",
    "usemin",
    "htmlmin"
  ]);

  grunt.registerTask("default", [
    "newer:jshint",
    "test",
    "build"
  ]);
};
```

Par exemple, la dernière tâche constitue la tâche par **défault** qui est exécutée lorsque vous saisissez la commande suivante :

```
grunt
```

Cette tâche se décompose en 3 sous-tâches :

- ✓ une tâche de vérification de code via **jshint**
- ✓ une tâche de **test**
- ✓ une tâche de **build**

En remontant un peu dans le fichier, on remarque les lignes suivantes :

```
connect: {
    options: {
      port: 9000,
      hostname: "localhost",
      livereload: 35729
    },
    livereload: {
      options: {
        open: true,
        middleware: function (connect) {
          return [
            connect.static(".tmp"),
            connect().use(
              "/bower_components",
              connect.static("./bower_components")
            ),
            connect.static(appConfig.app)
          ];
        }
      }
    }
```

Ces lignes indiquent notamment qu'un serveur web sera fonctionnel lorsque vous saisirez la commande suivante :

```
grunt serve
```

5.2. L'application

Vous l'aurez remarqué, dans le projet, se trouve un répertoire app. Sans trop de surprise, c'est dans ce répertoire que se trouve le code de votre projet. Voyons les répertoires et fichiers importants que nous y trouvons:

- ✓ **Répertoire images :** c'est dans ce répertoire que nous mettrons les images.
- ✓ **Répertoire styles :** l'ensemble des feuilles de style CSS iront dans ce dossier.
- ✓ **Répertoire views :** c'est ici que nous mettrons les templates HTML.
- ✓ **Répertoire scripts :** c'est le cœur de l'application. C'est notamment ici qu'iront les contrôleurs, services et directives Angular.
- ✓ **Page index.html :** c'est le point d'entrée de notre application. Comme il s'agit d'une application web, il s'agit donc d'une page HTML.

5.3. Les tests

Les tests constituent une part extrêment importante, y compris lorsque vous faites du JavaScript. Lorsque vous faites du développement avec Angular, vous devez acquérir cette pensée du test car le framework est construit autour de ça.
Si vous naviguez dans le répertoire test vous constaterez deux choses très intéressantes :

- ✓ **Un répertoire spec :** vous constaterez un répertoire controller avec un fichier main.js. Il s'agit, vous l'aurez compris, des tests du contrôleur principal.
- ✓ **Un fichier de conf karma.conf.js** : karma est ce que l'on appelle un **"Test Runner".** Son rôle est tout simplement d'automatiser l'exécution des tests.

La commande suivante permet d'exécuter vos tests :

```
grunt test
```

NB: Il est possible qu'une erreur du type **"Warning: Task "karma" not found. Use --force to continue."** se produise. Cette erreur signifie que le **plugin karma** pour **Grunt** n'est pas installé. Dans ce cas, il faut l'installer avec la commande suivante:

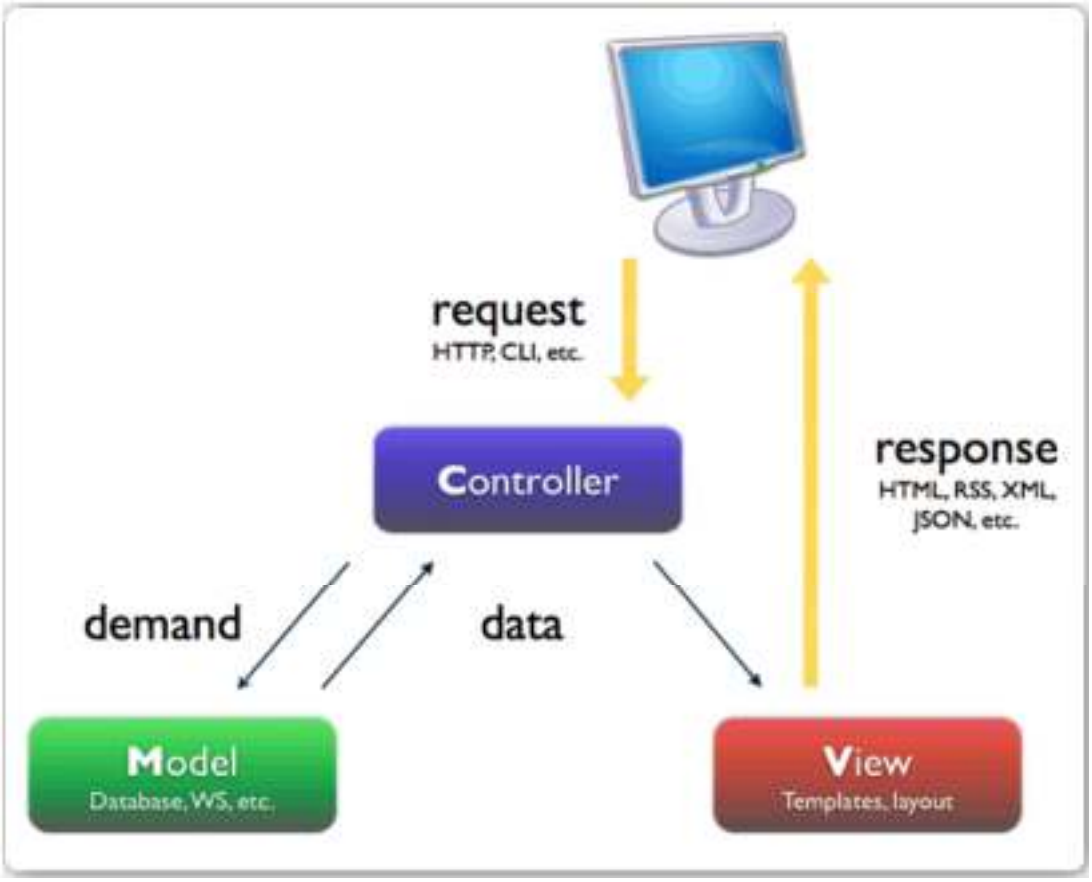
```
npm install grunt-karma --save-dev
```

6. Le modèle MVC

Le modèle MVC décrit une manière d'architecturer une application informatique en la décomposant en trois sous-parties :

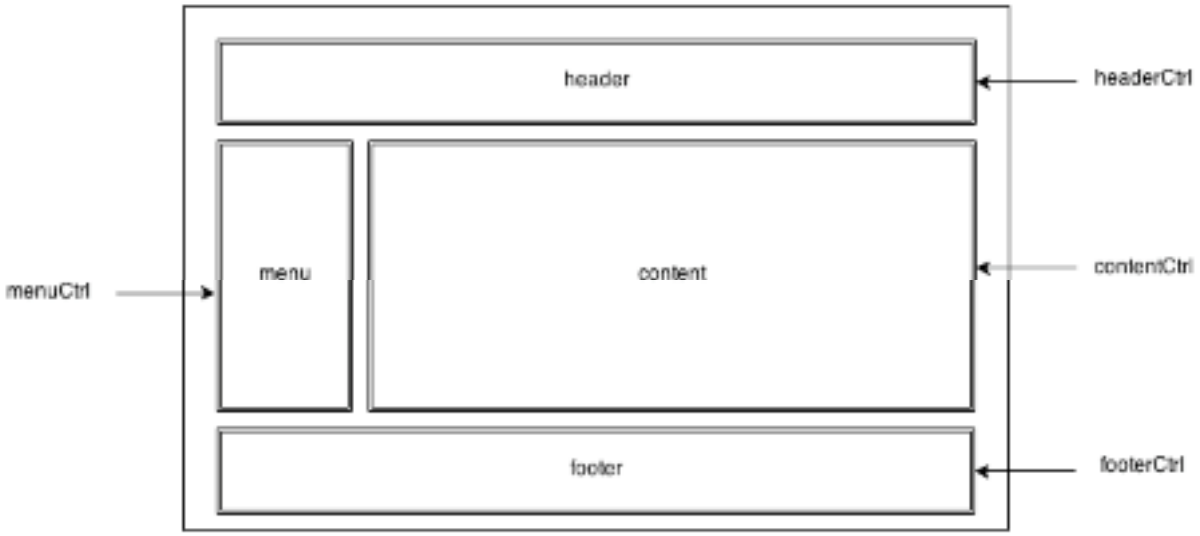
- ✓ **la partie Modèle :** La partie Modèle d'une architecture MVC encapsule la logique métier (business logic) ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (Modèle procédural) ou de classes (Modèle orienté objet).
- ✓ **la partie Vue :** La partie Vue s'occupe des interactions avec l'utilisateur : présentation, saisie et validation des données.
- ✓ **la partie Contrôleur :** La partie Contrôleur gère la dynamique de l'application. Elle fait le lien entre l'utilisateur et le reste de l'application.

Ce modèle de conception (design pattern) a été imaginé à la fin des années 1970 pour le langage Smalltalk afin de bien séparer le code de l'interface graphique de la logique applicative. Il est utilisé dans de très nombreux langages : bibliothèques Swing et Model 2 (JSP) de Java, frameworks PHP, ASP.NET MVC, etc.



7. Le MVC appliqué à AngularJS

Avec Angular JS utilisant le modèle MVC, il ne s'agit pas de mettre l'ensemble de votre code au sein du même contrôleur. Cela deviendrait très vite anarchique. Il s'agit donc de développer plusieurs "petits" contrôleurs qui vont agir sur différents éléments de la page.
Exemple :



Les différents contrôleurs d'une page

Chaque composant de la page possède son propre contrôleur. Cela permet de bien isoler votre code.
Le lien entre la vue HTML et le code JavaScript se fait au moyen de la directive **ng-controller**.
Ainsi, voici le squelette d'une page HTML :

```
<!doctype html>
<html>
   <head>
      <title></title>
   </head>
   <body>
      <section ng-controller="headerCtrl">
         <h1>HEADER</h1>
      </section>
      <section ng-controller="menuCtrl">
         <h1>MENU</h1>
      </section>
      <section ng-controller="contentCtrl">
         <h1>CONTENT</h1>
      </section>
      <section ng-controller="footerCtrl">
         <h1>FOOTER</h1>
      </section>
   </body>
</html>
```

voici les contrôleurs vides :

```
var app = angular.module("app", []);

app.controller("headerCtrl", function($scope){
   //...
});

app.controller("footerCtrl", function($scope){
   //...
});

app.controller("menuCtrl", function($scope){
   //...
});

app.controller("contentCtrl", function($scope){
   //...
});
```

8. La notion de data-binding

La plupart des frameworks qui permettent de créer des applications web utilisent cette notion de **data-binding**. C'est le cas de AngularJS. Il s'agit d'un moyen de lier la partie vue à la partie logique. En d'autres termes, grâce à cela, les éléments de votre code HTML seront liés à votre contrôleur JavaScript.

Exemple : Programme qui affiche le signe d'un entier saisi dans une zone de texte

Fichier javascript

```
var myApp = angular.module('myApp',[]);

myApp.controller("premierCtrl", function($scope){
   $scope.nombre = 0;
   $scope.verificateur = function(){
      if($scope.nombre<0)
            return  "negatif";
      else if($scope.nombre>0)
            return  "positif";
         else
            return  "null";

   };
});
```

Le fichier HTML

```
<div ng-app="myApp">
   <div ng-controller="premierCtrl">
      <input ng-model="nombre"/>
      <span>C'est un nombre <b ng-bind="verificateur()"></b></span>
</div>
```

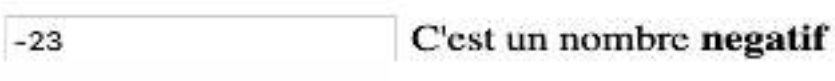
Dans ce fichier HTML, nous avons deux bindings :

- ✓ Le premier au niveau du champ input : nous avons la présence de l'attribut **ng-model** dans la balise HTML. En fait, le binding Angular se fait au moyen de ce que l'on appelle

des directives. Vous les reconnaissez car elles sont précédées de **ng.** Dans notre cas, nous disons tout simplement de lier la valeur du champ input à la valeur de la variable **nombre** présent **dans le code JavaScript**.

- Le deuxième au niveau de la balise **<b>.** Cette fois, nous utilisons la directive **ng-bind** a laquelle nous lions la fonction **verificateur()** qui retourne la chaîne de caractère **"positif, negatif ou null"** en fonction du nombre.

Résultat :



9. L'injection de dépendances

Au même titre que le data-binding, l'injection de dépendances est un concept clé lorsque vous développez avec **AngularJS.** L'injection de dépendances permet à des modules de ne pas se soucier de l'instanciation des modules dont ils dépendent. Il suffit d'appeler les dépendances et Angular se charge de les instancier et de les injecter pour nous.

Avec l'exemple précèdent sur le **data binding** , vous constaterez que nos contrôleurs possédaient tous une dépendance **: $scope** . C'est notamment grâce à ce module qu'il était possible pour nous de faire du **data-binding**. Seulement, à aucun nous avons procédé à l'instanciation du module. Cette instanciation est faite par le **système d'injection de dépendances** géré par Angular. Les contrôleurs ont juste à demander ce qu'ils veulent.

Angular possède nativement un certain nombre de modules **comme $scope , $location ...**

Ces modules sont tous précédés par **$** . C'est une convention qui nous permet de savoir qu'il s'agit effectivement de modules natifs à Angular.

l'injection de dépendances à plusieurs avantages :

- ✓ **La simplicité.** Vous n'avez plus à vous soucier du comment instancier les modules que vous utilisez. Lorsque vous développez quelque chose, vous n'avez pas besoin de vous soucier des autres composants.
- ✓ **La fiabilité :** Lorsque votre module est chargé, vous avez la certitude que toutes ses dépendances sont chargées et que vous avez la possibilité de les utiliser.

- ✓ **La réutilisabilité.** L'injection de dépendances permet donc d'inciter les développeurs à créer de petits modules unitaires et à les assembler par la suite pour créer des systèmes plus conséquents.

10. $scope et $watch

10.1. $scope

$scope constitue une dépendance de notre contrôleur. Il s'agit du mécanisme couramment utilisé par Angular afin d'exposer le modèle à la vue. En d'autres termes, le **data-binding** se fait grâce à cet objet **$scope**.

Toute donnée qui n'est pas attachée à **$scope** n'appartient pas au modèle et ne peut donc pas être exposée à la vue.

$scope est le contexte courant dans lequel vous pouvez agir. Chaque contrôleur possédera son propre objet **$scope** et donc son propre contexte.

Exemple :

```
var myApp = angular.module('myApp',[]);

myApp.controller("premierCtrl", function($scope){
    $scope.name = "World"
  };
});
```

Dans la vue :

```
<div ng-app="app">
  <div ng-controller="exempleCtrl">
    HELLO {{name}}!
  </div>
</div>
```

NB : Angular possède l'objet **$rootScope** qui permet d'accéder à l'ensemble des contextes présents dans votre page.

10.2. $watch

Il s'agit d'une fonction attachée à **$scope** qui va vous permettre d'observer certaines propriétés de votre modèle et de déclencher des opérations lorsque la valeur de ces propriétés changent.

La spécification de la fonction est la suivante :

```
$watch(watchFn, watchAction, deepWatch)
```

les paramètres :

- ✓ **watchFn :** la propriété de votre modèle que vous souhaitez observer. Ce paramètre peut-être soit une fonction, soit une expression.
- ✓ **watchAction :** fonction ou expression qui sera appelée lorsque **watchFn** change.
- ✓ **deepWatch :** Ce paramètre est optionnel. Il s'agit d'un **booléen** qui lorsqu'il est vrai indique à Angular qu'il doit déclencher **watchAction** lorsque les sous-propriétés d'un objet changent. Ce paramètre est pratique lorsque vous souhaitez examiner l'ensemble des propriétés d'un tableau ou d'un objet.

Exemple : Nous souhaitions développer un module d'achat. Lorsque ma facture est supérieure à 1000, on souhaite appliquer une promotion de 10%

```
app.controller("VenteCtrl", function($scope){
   $scope.articles = [{"nom": "Téléphone sans-fil", "quantite": 1, "prix": "300"},
{"nom": "Voiture", "quantite": 1, "prix": "1300"}];
   $scope.total = function(){
      var total = 0;
      for(var i = 0; i < $scope.articles.length; i++){
         total += $scope.articles[i].prix * $scope.articles[i].quantite;
      }
      return total;
   };
   function calculateDiscount(newValue, oldValue, scope){
      $scope.discount = (newValue > 1000) ? newValue * 0.10 : 0;
   };

   $scope.finalTotal = function(){
      return $scope.total() - $scope.discount;
   };

   $scope.$watch($scope.total, calculateDiscount);
});
```

11. Les directives

11.1. Qu'est-ce qu'une directive

Depuis le début de ce cours, vous avez remarqué que le code HTML que nous avons écrit était peuplé d'attributs comme **ng-controller, ng-repeat , ng-app , ng-model**... Cela constitue ce que l'on appelle des **directives**.

Les directives sont utilisées lorsque l'on souhaite modifier ou transformer le DOM (Document Object Model).

Angular fournit un certain nombre de directives :

- ✓ **ngController :** directive permettant d'attacher un contrôleur à la vue
- ✓ **ngRepeat :** directive permettant de répéter un template pour chaque élément d'une collection.
- ✓ **ngModel :** directive permettant de lier les input, textarea ou select à une propriété du contexte actuel.
- ✓ **ngApp :** Cette directive permet tout simplement d'initialiser votre application. Placez-la au niveau de votre balise body ou html, et passez-lui le nom de votre application.
- ✓ **…**

Il existe beaucoup de directives prédéfinies par **AngularJS**. Je vous invite à consulter la documentation officielle pour plus de détails.

11.2. Créer une directive

Il vous arrivera très fréquemment d'avoir besoin de créer vos propres directives. À titre d'exemple, une directive que vous serez amené à créer concerne la mise en attente de l'utilisateur lorsque celui-ci a réalisé une action qui demande un peu de temps. Par exemple, lors du chargement de votre page, il pourrait être pertinent d'afficher un curseur de chargement... Cela se fait au moyen d'une directive qu'il vous faudra créer.

11.2.1. Le nommage

Les directives natives à Angular sont TOUTES précédées de **ng**. Ce **ng** constitue l'espace de nom Angular ; par conséquent, évitez de l'utiliser lorsque vous créez vos propres directives.

La convention de nommage suit cependant une règle importante à laquelle il ne faudra pas déroger. Dans votre JavaScript, lorsque vous créerez votre directive, le nom que vous lui attribuerez devra être formé de la manière suivante :

```
namespaceDirectiveName
```

Pour appeler la directive dans une vue, on utilise la syntaxe suivante :

```
namespace-directiveName
```

Ainsi, si je crée la directive loading, mon module JavaScript s'appellera **maLoading** et je pourrai l'appeler dans mon code HTML par le nom **ma-loading**.

11.2.2. Les options

Le tableau suivant montre les différentes options pour la création d'une directve

Propriété	Description
restrict(Optionnel)	Permet de définir comment la déclaration de la directive doit se faire dans le code HTML. Les options sont : 'E' pour element. Exemple : <my-loading title="Paramètres">...</my-loading> 'A' pour attribut. Exemple : ... 'C' pour classe. Exemple : ... 'M' pour commentaires. Exemple : <!-- directive: my-loading Paramètres --> La valeur par défaut est 'A'. Il peut parfois être pertinent d'utiliser le 'M', surtout lorsque vous souhaitez ne pas avoir de conteneur dans votre HTML.
priority(Optionnel)	Dans le cas où, sur un même élément, vous auriez plusieurs directives, il est possible d'accorder une priorité à une directive. Les priorités hautes s'exécutent en premier. La valeur par défaut est 0.
templates(Optionnel)	Permet de remplacer le contenu d'un élément par un template. Cela est par exemple utile lorsque vous développez un système d'onglets. Au lieu d'écrire l'ensemble de votre code au sein d'une seule page HTML dans laquelle vous allez poser tout un ensemble de conditions, vous pouvez juste déclarer un template pour chaque onglet.
templateUrl(Optionnel)	Cette option permet de charger un template depuis un fichier.
replace(Optionnel)	Si vrai, remplace l'élément. Si faux ou non-spécifié, ajoute la directive à l'élément courant.
transclude(Optionnel)	Au lieu de remplacer ou ajouter du contenu, vous pouvez aussi déplacer le contenu original à l'intérieur du nouveau template grâce à cette propriété. Si définie à true, la directive supprimera le contenu original et le rendra disponible pour le réinsérer à l'intérieur du template en utilisant la directive ng-transclude.

scope(Optionnel)	Créer un nouveau contexte au lieu d'hériter du contexte du parent. Trois options s'offrent à vous lorsque vous souhaitez avoir un contexte. Soit : Vous utilisez le contexte existant qui est celui dans lequel évolue votre directive (valeur par défaut). Vous définissez un nouveau contexte qui va hériter du contexte du parent (en l'occurrence le contrôleur qui encapsule votre directive). Vous créez un contexte isolé qui n'hérite pas des propriétés du contexte du parent à moins que vous spécifiiez les attributs dont vous voulez hériter. Voila la syntaxe que vous devrez utiliser pour avoir le comportement voulu : contexte existant : scope : false nouveau contexte : scope : true contexte isolé : scope: { attributeName1: 'BINDING STRATEGIE',...} La stratégie de binding est définie selon trois symboles : @ : Passer l'attribut sous forme de chaîne de caractères. = : Utiliser le data-binding afin de lier la propriété à la propriété du parent. & : Passer une fonction depuis le parent qui sera exécutée plus tard.
controller(Optionnel)	Créer un contrôleur. Cela s'avère utile notamment lorsque vous avez plusieurs directives que vous voulez faire communiquer. On peut facilement imaginer qu'un menu ait besoin de connaître les items qu'il comporte pour pouvoir afficher ou cacher les bons éléments !
require(Optionnel)	Indique qu'une autre directive est requise pour que celle-ci fonctionne correctement.

Exemple : Création d'un module « myModule »

```
var myModule = angular.module(...);

myModule.directive('namespaceDirectiveName', function factory(injectables) {
   var directiveDefinitionObject = {
      restrict: string,
      priority: number,
      template: string,
      templateUrl: string,
      replace: bool,
      transclude: bool,
      scope: bool or object,
      controller: function controllerConstructor($scope,
                                                 $element,
                                                 $attrs,
                                                 $transclude),
      require: string,
      link: function postLink(scope, iElement, iAttrs) { ... },
      compile: function compile(tElement, tAttrs, transclude) {
         return {
            pre: function preLink(scope, iElement, iAttrs, controller) { ... },
            post: function postLink(scope, iElement, iAttrs, controller) { ... }
         }
      }
   };
   return directiveDefinitionObject;
});
```

12. Les Services

12.1. C’est quoi une service

Les services des singletons, c'est-à-dire des instances uniques d'objets. Le rôle d'un service est de fournir un ensemble de tâches nécessaires au fonctionnement de votre application.
Comme pour les directives, Angular fournit déjà des services très utiles comme :

- **$location :** interagir avec l'URL de votre navigateur.
- **$route :** changer de vue en fonction de l'URL.
- **$http :** communiquer avec les serveurs.

12.2. Création d’un service

Un service est considéré comme un simple module qui est chargé de retourner un objet. Cet objet possédera un ensemble de fonctions qui seront chargées d'exécuter un certain nombre de services.

Il suffit donc d'initialiser un nouveau module tout en spécifiant si ce module est de type **factory, service ou provider**.

12.3. Factory

```
app.factory("factoryExemple", function(){
      return{
      service1: function(){...},
      service2: function(){...},
      }
});
app.controller("factoryCtrl", function($scope, factoryExemple){
      factoryExemple.service1();
});
```

Un **factory** doit retourner un objet qui possède un certain nombre d'utilitaires qui assurent les services.

12.4. Service

```
app.service("serviceExemple", function(){
      this.service1 = function(){...};
      this.service2 = function(){...};
});
app.controller("serviceCtrl", function($scope, serviceExemple){
      serviceExemple.service1();
});
```

Lorsque vous utilisez le service, il s'agit d'une instance du service qui est passée au contrôleur. Vous n'avez donc pas la responsabilité de retourner une valeur dans votre service. Il vous suffit d'attacher vos fonctions et attributs **à this**.

13. Les filtres

13.1. Qu'est-ce que les filtres

Les filtres sont, au même titre que les directives et services, des modules Angular. Ils ont pour rôle de transformer la donnée afin de l'afficher de manière plus élégante. Ce sont donc des éléments qui vont enrichir votre **template** et dont la syntaxe reste assez simple :

```
{{expression | filtre : paramètre 1 : paramètre 2 : ...}}
```

Angular fournit, comme pour les directives et les services, des filtres prédéfinis. En voici quelques exemples :

- **date :** ce filtre est très utile puisqu'il vous permet de formater une date au bon format (anglais, français, avec le nom des mois, des jours...).
- **currency :** il s'agit d'un filtre qui vous permettra d'afficher une valeur monétaire correctement formatée (exemple : bon nombre de décimales et affichage de la monnaie).
- **lowercase et uppercase :** permet d'afficher une chaîne de caractères en minuscule ou majuscule

Exemple :

```
<div ng-app="app">
<div ng-controller="myController">
      <span>{{'2018-05-01' | date: 'dd MMM yyyy'}}</span>
</div>
</div>
```

Affichera : 01 May 2018

13.2. Création de filtres

Les filtres, tout comme les directives, services et contrôleurs, sont des modules. La syntaxe reste la même que pour les services par exemple.

Exemple : Création d'un filtre qui permet de renverser une chaîne de caractères et optionnellement de mettre les caractères en majuscules.

```
var app = angular.module("app", []);
app.filter('reverse', function() {
     return function(input, uppercase) {
               input = input || '';
               var out = "";
               for (var i = 0; i < input.length; i++) {
                    out = input.charAt(i) + out;
               }
               // condition basée sur un argument optionnel
               if (uppercase) {
                    out = out.toUpperCase();
               }
               return out;
          };
});
app.controller('myController', function($scope) {
$scope.greeting = 'hello';
});
```

Utilisation du filtre **reverce** :

```
<div ng-app="app">
<div ng-controller="myController">
<span>{{greeting | reverse: true}}</span>
</div>
</div>
```

14. Les routes

14.1. Instalation du module angular-route

La gestion des routes n'est pas prévue nativement par Angular. Il convient donc d'installer un module supplémentaire, une extension. Pour installer un module, on utilise **Bower**.

```
bower install angular-route --save
```

Cela permettra d'installer la dépendance et d'inscrire cette dépendance dans le fichier **bower_components.json.**

Pour rendre la dépendance utilisable, on utilise la commande suivante :

```
grunt build
```

14.2. Le routing avec Angular

Le routing consiste à associer une URL à un template et au contrôleur associé !

Voici donc la syntaxe utilisée pour réaliser le routing :

```
var module = angular.module('module', [])
module.config(function($routeProvider) {
	$routeProvider
		.when('url', {controller:aController, templateUrl:'/path/to/tempate'})
		.when(...other mappings for your app...)
		...
		.otherwise(...what to do if nothing else matches...);
)};
```

Commentaires:

- Il suffit d'énumérer vos règles de routing les unes à la suite des autres.
- Chaque règle est composée de **l'URL** pour laquelle la règle devra se déclencher puis d'un objet au format **JSON** indiquant le **template** à charger et le contrôleur concerné.
- La dernière règle permet de définir une règle par défaut, appelée si aucune autre règle n'a matché. Il est très courant dans ce cas-là, d'opérer une redirection.

Exemple

```
phonecatApp.config(['$routeProvider',
function($routeProvider) {
	$routeProvider.
		when('/personnes', {
			templateUrl: 'personness/personne-list.html',
			controller: 'PersonneListCtrl'
		}).
		when('/personnes/:personneId', {
			templateUrl: 'partials/phone-detail.html',
			controller: 'PhoneDetailCtrl'
		}).
		otherwise({
			redirectTo: '/phones'
		});
}]);
```

Voici la déclaration des contrôleurs :

```
var personneControllers = angular.module('personneControllers', []);
personneControllers.controller('PersonneListCtrl', ['$scope', '$http',
function ($scope, $http) {
      $http.get('personnes/personnes.json').success(function(data) {
      $scope.personnes = data;
});
$scope.orderProp = 'age';
}]);
personneControllers.controller('PersonneDetailCtrl', ['$scope',
'$routeParams',
function($scope, $routeParams) {
      $scope.phersonneId = $routeParams.personneId;
}]);
```

Références

https://openclassrooms.com/en/courses/2516051-developpez-vos-applications-web-avec-angularjs/2521751-les-directives

http://www.gchagnon.fr/cours/dhtml/angular.html

http://prof.bpesquet.fr/cours/modele-mvc/

http://expressjs.com/fr/guide/using-middleware.html

https://openclassrooms.com/fr/courses/1056721-des-applications-ultra-rapides-avec-node-js/1057503-le-framework-express-js

https://www.grafikart.fr/formations/nodejs/nodejs-intro

https://makina-corpus.com/blog/metier/2014/introduction-a-nodejs

https://blog.lesieur.name/installer-et-utiliser-nodejs-sous-windows/

http://wdi.supelec.fr/appliouaibe/Cours/JSserveur