Devenez un ninja avec AngularJS

Ninja Squad

Table of Contents

Devenez un ninja avec

AngularJS

ninja squad

Chapter 1. Introduction

Angular a été créé par une équipe de Googlers (et toujours principalement maintenu par eux) en 2009. En quelques années, il est devenu l'un des frameworks JavaScript les plus appréciés de la communauté pour créer des applications **Single Page** avec des centaines de contributeurs, de forks et de petites étoiles sur Github. La licence utilisée est la **licence MIT**.

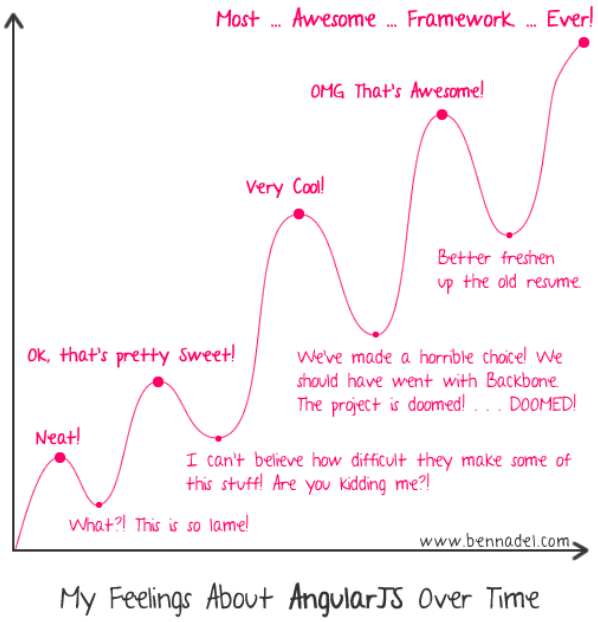
Le framework est actuellement en version 1.3.x, mais une partie de l'équipe travaille également sur ce que deviendra la version 2.0 dans quelques mois.

Savez-vous pourquoi le projet s'appelle Angular ? Tout simplement parce que HTML utilise des 'angular brackets' : les symboles '<' et '>'.

Angular n'abstrait pas le HTML, JS ou CSS. Il n'impose pas d'objet parent, ne nécessite pas d'extension navigateur. Nous allons faire du développement Web, du vrai, 100% JavaScript !

Apprendre Angular est un vrai plaisir, même si le chemin à parcourir n'est pas sans accroche. Un bloggeur[1], débutant en AngularJS, a produit cette image qui résume assez bien l'impression que l'on peut avoir :

[1] http://www.bennadel.com/blog/2439-My-Experience-With-AngularJS-The-Super-heroic-JavaScript-MVW-Framework.htm



Les navigateurs supportés sont :

- Chrome
- Safari
- Firefox
- Opera
- IE >= 9 (IE8 a été abandonné depuis la version 1.3).
- Android browser
- iOS

Angular appartient à la famille des frameworks MVC-like, c'est-à-dire qu'il propose de structurer son code autour :

- de Models : de purs objets JavaScript qui contiennent les données
- de Views : des templates HTML accédant aux Models exposés
- de Controllers : des fonctions JS qui exposent le model, à travers l'objet $scope et des méthodes utilitaires.

Angular possède un concept qui lui est propre : les directives. Ce sont elles qui nous permettront d'ajouter du comportement à notre application. Toutes nos applications

seront écrites grâce à ces directives. Parfois sans même s'en rendre compte, tellement leur utilisation est transparente !

D'autres frameworks se partagent le coeur des développeurs avec AngularJS. Parmi les plus notables on peut citer :

BackboneJS[2], le plus ancien, qui laisse plus de liberté mais demande plus de code.

EmberJS[3], qui a notamment conquis la communauté Ruby.

ReactJS[4], le plus récent, créé par les développeurs de Facebook.

Cette liste n'est bien sûr pas exhaustive, et nous ne nous attarderons pas sur les spécificités de chacun. Angular n'est pas la réponse à toutes les applications Web mais il fait très bien ce qu'on lui demande et est clairement l'un des préférés des développeurs ! C'est en tout cas le préféré de notre équipe de ninjas ;).

[2] http://backbonejs.org/
[3] http://emberjs.com/
[4] http://facebook.github.io/react/

Chapter 2. Getting started

Angular nous invite à utiliser un style déclaratif pour construire l'interface et un style impératif pour construire la logique de notre application. Le framework repose sur un mécanisme de **directives**, utilisées sous forme de mot clés à insérer dans le HTML, pour simplifier le développement. C'est surprenant au début, mais on s'y fait rapidement, et on en vient vite à l'apprécier.

Les développeurs d'Angular ont une vision claire de ce que doit être le Web, et encouragent le développeur à utiliser certaines pratiques :

- peu de 'boilerplate'
- découplage du DOM et de la logique
- utilisation de composants
- automatisation des tests (oui, oui, en JS !)
- découplage client/serveur

La simplification du développement est également assurée par différents principes que nous verrons en détail :

- binding bi-directionnel
- pattern MV*
- modularité
- injection de dépendances
- routage
- validation

2.1. From zero to something

Angular permet de limiter au maximum le code 'boilerplate' à écrire. Pour utiliser Angular dans une application, rien de plus simple : il suffit d'inclure la librairie 'angular.js' dans votre HTML. Vous pouvez bien sûr récupérer manuellement Angular, dans sa version minifiée ou non, ou l'inclure grâce à un provider CDN ou encore utiliser un outil de gestion des dépendances comme Bower[1].

[1] http://bower.io/

```
<html>
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body></body>
</html>
```

Maintenant, nous pouvons indiquer à Angular la partie de l'application qu'il doit surveiller avec la directive ng-app. Nous reviendrons dessus en détail plus tard.

```
<html>
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app></body>
</html>
```

Tout le template HTML dans le <body> est maintenant surveillé (et interprété) par Angular !

Il est possible de donner un nom à notre application, par exemple 'app'.

```
<html>
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app="app"></body>
</html>
```

Et dans ce cas, nous devrons définir un module du même nom dans notre code JavaScript, qui contiendra le code et les composants de notre application.

```
<html>
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app="app">
    <script>
      var app = angular.module('app', []);
    </script>
  </body>
</html>
```

L'application pourra éventuellement dépendre de modules extérieurs, comme 'angular-ui-bootstrap' qui permet d'utiliser les composants basés sur le framework CSS Boostrap[2].

```
<html>
  <head>
    <script src="lib/angular.js"></script>
    <script src="lib/ui-bootstrap.js"></script>
  </head>
  <body ng-app="app">
    <script>
      var app = angular.module('app', ['ui.bootstrap']);
    </script>
  </body>
</html>
```

Nous reviendrons sur ce principe de modules plus en détail.

2.2. From zero to something better

Si vous démarrez un nouveau projet, vous allez vouloir trouver les bonnes pratiques de structuration de votre application, de build (minification, concaténation, vérification du code, lancement des tests…), etc… Pas de panique, nous aborderons les principaux points tout au long de notre progression.

Vous croiserez probablement le nom de ces outils dans vos recherches :

Bower[3], le gestionnaire de dépendances;

Grunt[4], un outil de build pour définir et enchaîner les différentes tâches;

[2] http://getbootstrap.com
[3] http://bower.io/
[4] http://gruntjs.com/

Gulp[5], une alternative à Grunt, plus récente et qui gagne en popularité;

Yeoman[6], un générateur de code très populaire.

Il existe déjà de nombreux exemples ou générateurs de projets pour vous aider. On retiendra notamment :

- Le générateur AngularJS officiel de Yeoman[7]. Il vous préparera un projet tout prêt, avec son build Grunt, son fichier Bower, sa configuration pour recharger à chaud vos modifications. Un très bon point de départ, très complet.
- Un autre générateur Yeoman[8], celui-ci basé sur Gulp plutôt que Grunt.
- Le projet d'exemple Angular Seed[9] de l'équipe Angular.
- d'autres générateurs 'full-stack', c'est-à-dire incluant également une partie serveur, comme le très populaire JHipster[10] qui vous intéressera si vous faites du Java.

Ces différents projets sont de très bons points de départ, pour se familiariser avec l'écosystème effervescent du monde JavaScript.

[5] http://gulpjs.com/
[6] http://yeoman.io/
[7] https://github.com/yeoman/generator-angular
[8] https://github.com/Swiip/generator-gulp-angular
[9] https://github.com/angular/angular-seed
[10] http://jhipster.github.io/

2.3. Expression

Si l'on vous dit qu'Angular interprète nos templates, c'est que nous allons pouvoir écrire du HTML contenant des expressions dynamiques, que le framework se chargera de compiler. On utilise la syntaxe {{ }} pour indiquer une expression à évaluer. Cette expression doit être une expression JavaScript valable, par exemple :

```
{{ 2 + 2 }} // 4
{{ "ninja " + "squad" }} // ninja squad
```

En réalité, elle ne peut pas être n'importe quelle expression JavaScript, car son évaluation par Angular est un peu différente d'un 'eval()' classique. En effet, les variables ne sont pas recherchées dans l'objet 'window' mais dans l'objet 'scope' que nous allons voir dans un instant. Ces expressions pardonnent également les null ou undefined qui peuvent survenir. Cela peut parfois être surprenant car l'accès à une variable indéfinie échouera donc silencieusement (en affichant une chaîne de caractères vide à la place de la référence inconnue). Il n'est en revanche pas possible d'utiliser des conditions, boucles ou throw dans les expressions. Enfin, nous verrons plus tard que ces expressions peuvent utiliser des filtres.

Chapter 3. Controllers et binding

3.1. Binding

L'une des parties récurrentes du travail de développeur Web consiste à maintenir l'affichage des champs en accord avec le modèle, lorsque celui-ci est modifié. Et lorsque l'utilisateur interagit avec la vue, par exemple via des champs de formulaire, il faut mettre à jour le modèle en fonction des données saisies. C'est souvent du code verbeux, sujet aux erreurs d'inattention et dont on se passerait volontiers. Angular gère tout ce boilerplate pour nous : c'est ce que l'on appelle le binding, en l'occurrence un binding bi-directionnel puisqu'il fonctionne à la fois dans le sens 'modèle → vue' et dans le sens 'vue → modèle'.

Pour binder un champ à un model, Angular a introduit un mot clé, une directive dans le jargon du framework, nommée ng-model. Lorsque celle-ci est ajoutée à un champ, Angular se charge de surveiller les changements et met à jour l'affichage de ce champ pour vous dans tous les endroits de votre application !

Par exemple, avec ce HTML, la seconde 'div' se mettra automatiquement à jour à chaque frappe de l'utilisateur dans l''input' :

```
<input type="text" ng-model="name" placeholder="Name..." />
<div>Hello {{name}}!</div>
```

Le modèle peut également être l'attribut d'un objet :

```
<input type="text" ng-model="person.name" placeholder="Name..." />
<div>Hello {{person.name}}!</div>
```

Le modèle peut également servir dans le HTML, comme 'class' par exemple !

```
<div><input type="text" ng-model="name" placeholder="Name..." /></div>
<div class="{{name}}">Hello {{name}}!</div>
```

3.2. Scope

Vous avez sans doute remarqué que les expressions précédentes font appel à des variables (name, person…). Où le framework les trouvent-ils ? Angular fournit un objet très particulier que vous allez manipuler très régulièrement : le scope (nommé $scope

dans le code). Il représente le modèle de l'application et le contexte d'exécution pour les expressions.

Par exemple, lors de l'évaluation de l'expression {{ name }}, Angular va chercher dans le scope courant un attribut nommé 'name'. Si un tel attribut n'est pas trouvé, l'expression renverra 'undefined', et affichera donc une chaîne vide. On peut ainsi utiliser le scope pour initialiser des valeurs de son application, ou récupérer les champs modifiés par un utilisateur (puisque je vous rappelle que le binding bi-directionnel d'Angular fait que chaque modification d'un champ par l'utilisateur entraîne la mise à jour de l'attribut du scope correspondant).

Si l'on reprend l'évaluation d'une expression plus complète comme {{ "hello" + name }}, alors Angular va procéder comme suit :

```
{{ "hello " + name }}
-> $scope.$eval('"hello " + name')
-> $scope.$eval('"hello "') + $scope.$eval('name')
-> "hello " + $scope.name
```

Ici, soit l'attribut $scope.name existe, et il est remplacé par sa valeur, soit il n'existe pas, et Angular le remplacera par une chaîne vide.

Lorsqu'une application Angular est démarrée, un objet $rootScope est créé. Ce scope racine sera le parent de tous les scopes créés. En effet, les scopes sont organisés de façon hiérarchique, à l'instar de la structure DOM de l'application. Le scope sert également de glue entre la vue et le controller. Chaque controller créé aura son propre scope. Les directives peuvent également avoir leur propre scope.

Il est ainsi possible d'initialiser et de manipuler les valeurs de son application dans son controller comme nous allons le voir.

3.3. Controller

Il est plus simple de gérer chaque vue de son application avec un controller dédié. Ce controller sera chargé d'initialiser les valeurs, contiendra les différentes fonctions à appeler. Comme nous venons de le voir, l'objet scope nous permettra de communiquer avec la vue. Ce scope aura une portée limitée au DOM du controller.

Pour déclarer un controller, on utiliser la directive ng-controller :

```
<html>
```

```
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app>
    <div ng-controller="InputCtrl">
      <input type="text" ng-model="name"/>
      <div>Hello {{name}}!</div>
    </div>
    <script>
      function InputCtrl($scope) {
        $scope.name = "Cedric";
      }
    </script>
  </body>
</html>
```

La valeur passée à la directive ng-controller référence le nom de la fonction du controller. Ici, Angular va chercher une fonction nommée 'InputCtrl' et l'appeler. Vous remarquerez que la fonction prend comme paramètre le scope, nous reviendrons plus tard sur ce sujet. Pour l'instant, pensez simplement à bien le passer à votre fonction. La fonction initialise ensuite l'attribut name du scope avec la valeur "Cedric".

Le scope du controller n'aura comme portée que la balise sur laquelle il est déclaré (et les balises englobées par celle-ci bien sûr). Ainsi, si l'on prend l'exemple suivant, quasiment identique, mais avec une expression utilisant name en dehors de la div gérée par le controller :

```
<html>
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app>
    <div ng-controller="InputCtrl">
      <input type="text" ng-model="name"/>
      <div>Hello {{name}}!</div>
    </div>
    <div>Hello {{name}}!</div>
    <script>
      function InputCtrl($scope) {
        $scope.name = "Cedric";
      }
    </script>
  </body>
</html>
```

Alors la deuxième div ne contiendra que 'Hello' à l'affichage.

On peut, bien sûr, utiliser plusieurs controllers dans une même page :

```
<html>
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app>
    <div ng-controller="InputCtrl">
      <input type="text" ng-model="name"/>
      <div>Hello {{name}}!</div>
    </div>
    <div ng-controller="SecondCtrl">
      <input type="text" ng-model="name"/>
      <div>Hello {{name}}!</div>
    </div>
    <script>
      function InputCtrl($scope) {
        $scope.name = "Cedric";
      }
      function SecondCtrl($scope) {
        $scope.name = "Ninja";
      }
    </script>
  </body>
</html>
```

La première div affiche alors 'Hello Cédric' et la seconde 'Hello Ninja'. Vous vous doutez qu'il viendra un moment où, comme on le ferait avec un framework MVC côté serveur, nous allons vouloir indiquer à Angular d'afficher telle vue, avec tel controller, pour telle URL. Vous avez raison, et nous y viendrons plus tard !

Vous avez remarqué ? La déclaration d'un controller est très simple : une fonction dans un script, et hop!, Angular le retrouve et l'enregistre. Il est également possible, et c'est la façon que l'on privilégiera à partir de maintenant, d'utiliser la fonction `controller` sur l'application si nous l'avons nommée (ou un module dédié aux controllers).

> Depuis la version 1.3.0, cette manière plus propre de déclarer un controller est la seule supportée par défaut!

```
<html>
  <head>
```

```
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app="app">
    <div ng-controller="InputCtrl">
      <input type="text" ng-model="name"/>
      <div>Hello {{name}}!</div>
    </div>
    <script>
      var app = angular.module('app', []);
      app.controller('InputCtrl', function($scope) {
        $scope.name = "Cedric";
      });
    </script>
  </body>
</html>
```

Un controller peut également posséder des fonctions à appeler, il suffit de déclarer ces fonctions comme un attribut quelconque du scope :

```
<html>
  <head>
    <script src="lib/angular.js"></script>
  </head>
  <body ng-app="app">
    <div ng-controller="ClickCtrl">
      <div ng-click="sayHello()">Click here</div>
    </div>
    <script>
      var app = angular.module('app', []);
      app.controller('ClickCtrl', function($scope) {
        $scope.sayHello = function() {
          alert("Hello !");
        };
      });
    </script>
  </body>
</html>
```

Notez au passage l'utilisation de la directive ng-click pour déclencher une action, la méthode sayHello, sur l'événement 'click' ! Ainsi lorsque l'on clique sur la div contenant "Click here", la popup d'alerte s'affiche.

Les fonctions des controllers sont également disponibles dans les expressions, ainsi pour un controller comme suit :

```
function HelloCtrl($scope) {
  $scope.hello = function(name) {
    return "Hello " + name + "!";
  };
}
```

Il devient possible d'utiliser la fonction 'hello' dans une expression :

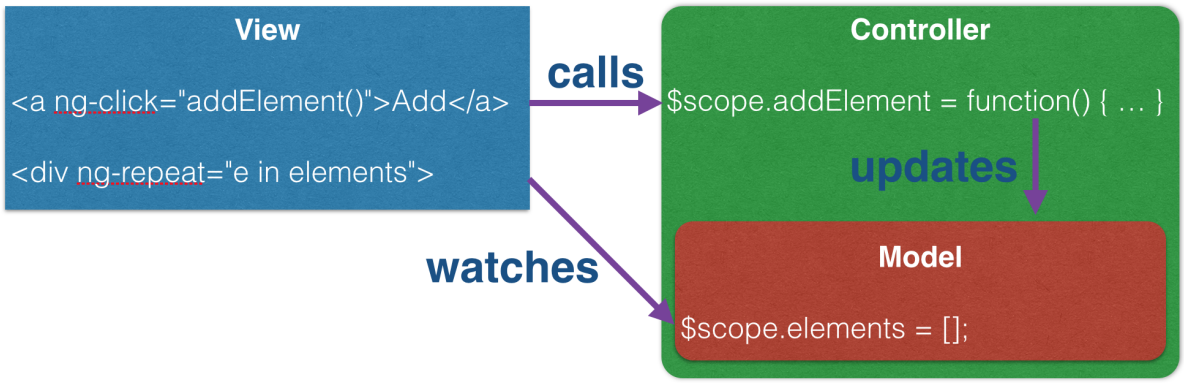
```
{{ hello('Cedric') }} // Hello Cedric!
```

Il est préférable de ne pas utiliser le scope directement dans les fonctions lorsque c'est possible, mais plutôt de passer les variables nécessaires en argument (on limite ainsi le couplage avec le scope, ce qui simplifie la testabilité).

Si l'on résume ce que l'on vient d'apprendre, nous avons :

- des templates, qui peuvent contenir des expressions et des directives, qu'Angular surveille, interprète et réaffiche pour nous à chaque changement.
- des controllers, qui peuvent manipuler notre modèle (le scope) et définir des fonctions utilisables au sein du controller ou depuis les templates.
- le modèle, représenté par les attributs du scope, et qui peuvent être de n'importe quel type JS (Boolean, String, Number, Date, Array, Object, Function, Regex…).

Le schéma ci-dessous montre un cas où le scope contient un tableau d'éléments, affichés dans l'interface grâce à la directive ng-repeat (qui affiche la balise sur laquelle elle est positionnée pour chaque élément du tableau).



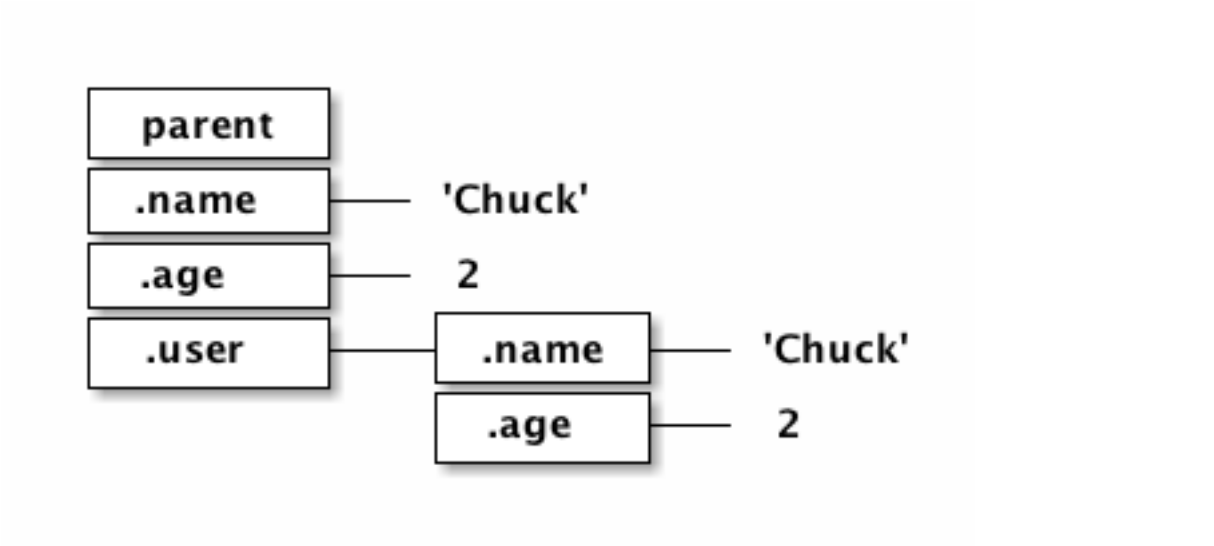
3.4. Héritage des scopes

En Angular, tous les $scope héritent du scope de base nommé $rootScope. On parle ici d'héritage prototypal, ce qui est un bien grand mot, j'en conviens. Si vous n'êtes pas familier avec ce type d'héritage, cette partie devrait vous intéresser, car c'est un concept du langage JavaScript depuis ses débuts.

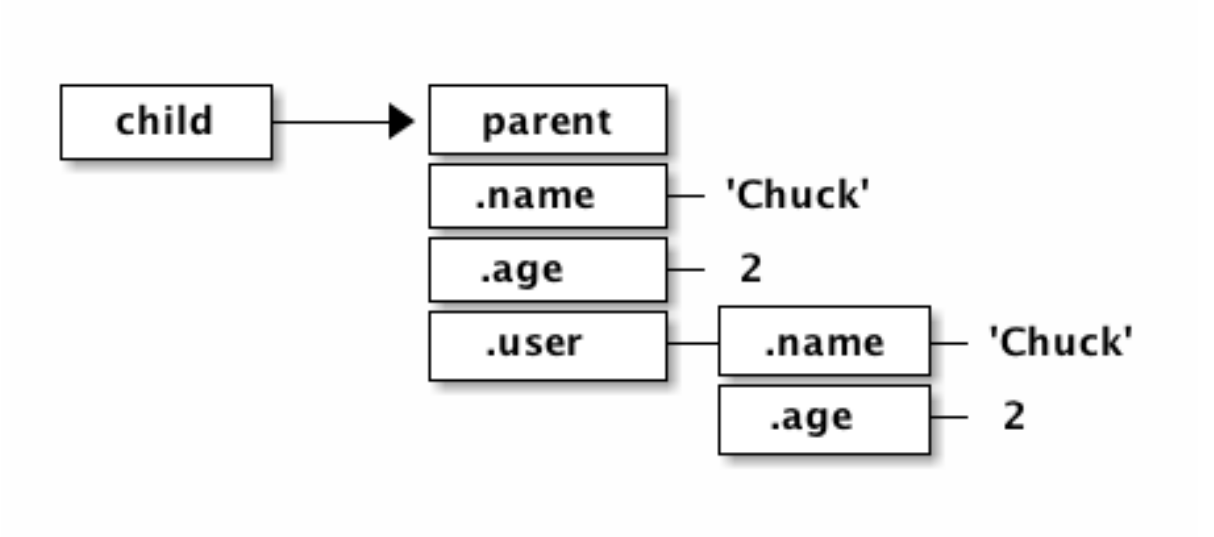
Prenons un scope parent, contenant par exemple un nom, un âge, un objet plus complet représentant l'utilisateur et une fonction, par exemple une méthode de login. Admettons qu'un scope hérite de ce scope parent, nommons le 'scope enfant' pour simplifier. A noter qu'en JavaScript une méthode ou un tableau sont stockés comme des objets.

```
parentScope.name = 'Chuck'; // Chuck is our beloved mascot :)
parentScope.age = '2';
parentScope.user = { name: 'Chuck', age: 2};
```

On peut représenter cela schématiquement :



L'objet 'child', le scope enfant, hérite de l'objet 'parent'. Nous allons manipuler les 3 propriétés : 'age', 'name' et 'user'. Comme indiqué plus haut, un tableau et une fonction se comporteraient comme l'objet 'user'.

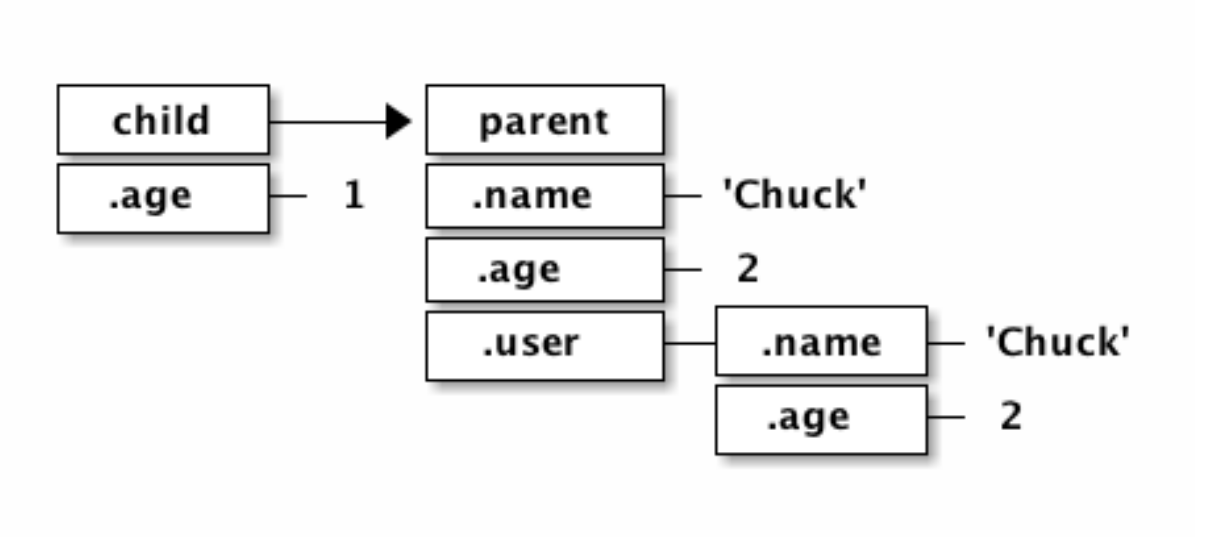


Si nous accédons à une propriété du 'scope enfant', comme la chaîne de caractères 'name', l'entier 'age' ou l'objet 'user', JavaScript va regarder si la propriété existe sur l'objet scope enfant, et si ce n'est pas le cas, remonter la chaîne des prototypes, à savoir chercher la même propriété sur l'objet scope parent (dont hérite le scope enfant). La propriété est trouvée et sa valeur est renvoyée.

```
console.log(childScope.name);  // 'Chuck';
console.log(childScope.age);   // '2';
console.log(childScope.user);  // { name: 'Chuck', age: 2}
console.log(childScope.login); // function(){ $scope.login = true }
```

En revanche, si l'on donne une valeur à une propriété du scope enfant, cette valeur va **masquer** la valeur du scope parent. Par exemple :

```
childScope.age = 1;
```

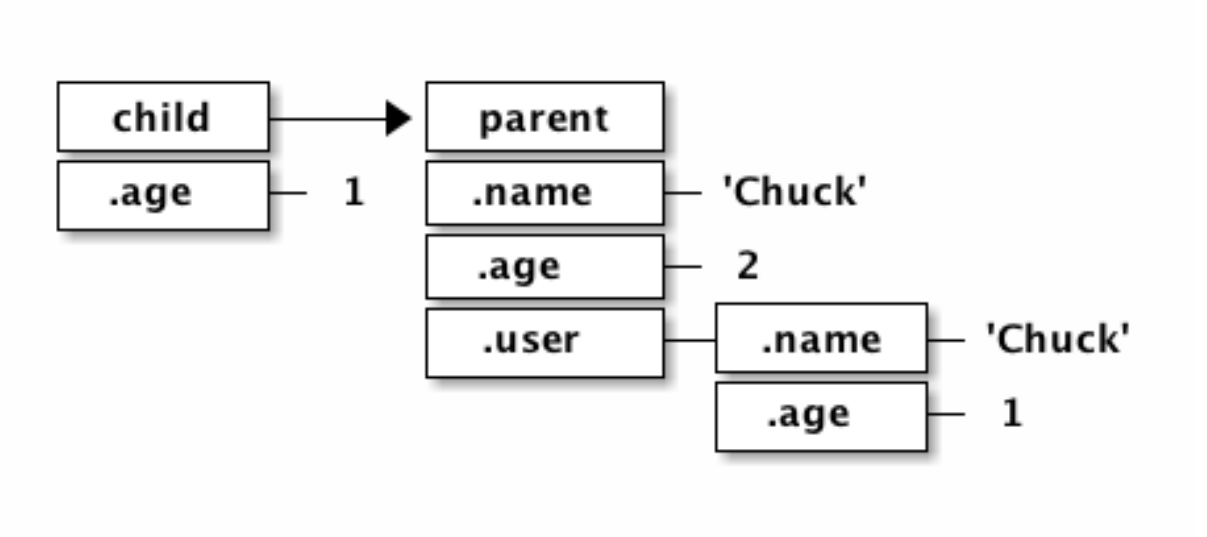


La propriété 'age' existe maintenant bien sur l'objet 'childScope' avec la valeur 1.

```
console.log(childScope.age);  // '1';
console.log(parentScope.age); // '2';
```

En revanche, lorsque l'on tente d'affecter une nouvelle valeur à une propriété d'un objet ou un élément d'un tableau, la chaîne prototypale est remontée. Ainsi, changer l'âge de l'objet 'user' sur le scope enfant entraînera l'écriture de la nouvelle valeur pour l'objet 'user' du scope parent :

```
childScope.user.age = 1;
```

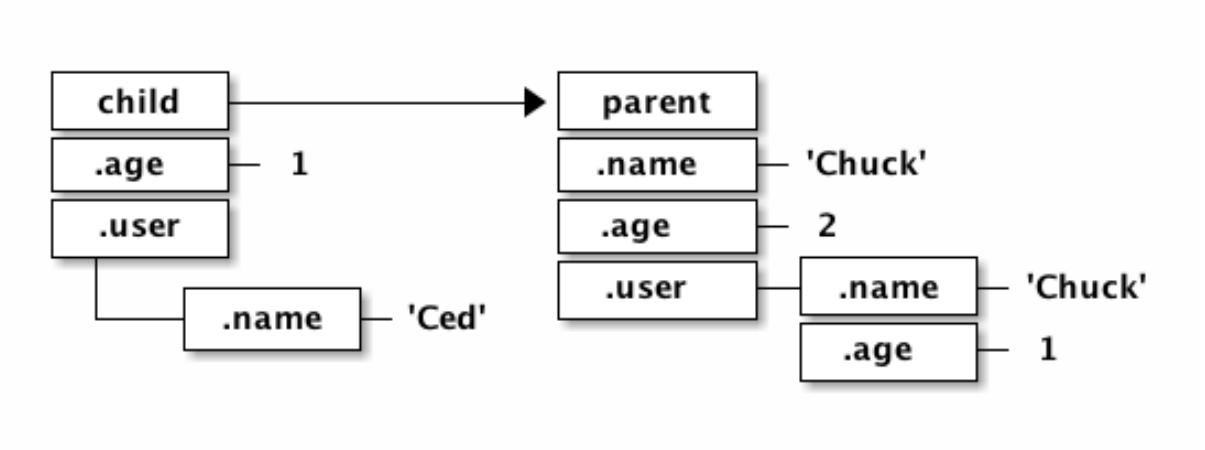


La propriété a bien été modifiée au niveau du parent :

```
console.log(childScope.user.age);  // '1';
console.log(parentScope.user.age); // '1';
```

Si par contre nous affectons un nouvel objet à la propriété 'user' du scope enfant, alors la propriété sera créée au niveau de l'objet enfant :

```
childScope.user = { name: 'Ced' };
```



Angular fonctionne de la même façon pour ses scopes. Créer un controller à l'intérieur d'un controller entraîne que le scope du controller fils héritera du scope du controller parent. C'est très pratique : le controller enfant, et donc sa vue associée, peuvent ainsi voir les valeurs stockées dans le controller parent. Dans ce cas, le controller à l'intérieur du premier est appelé 'nested controller'.

Dans l'exemple ci-dessous, on a ainsi un controller ParentCtrl qui définit user.name. Le controller ChildCtrl hérite de celui-ci, et la vue associée a donc également accès à la valeur de user.name.

```
<div ng-controller="ParentCtrl">
  {{ user.name }}
  <div ng-controller="ChildCtrl">
    {{ user.name }}
  </div>
</div>


<script>
  var app = angular.module('app', []);
  app.controller('ParentCtrl', function($scope) {
```

```
    $scope.user = { name: 'Cedric' };
  });
  app.controller('ChildCtrl', function($scope) {});
</script>
```

Le scope enfant possède une propriété spéciale $parent qui permet de référencer directement le scope parent. Son utilisation n'est pas nécessairement une bonne pratique, il vaut mieux l'éviter.

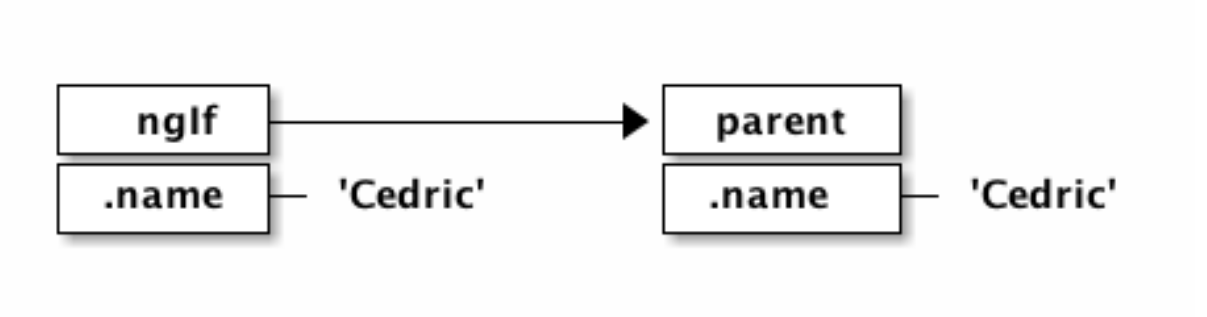
Les directives créent elles aussi parfois leurs propres scopes. On peut donc avoir des surprises si l'on utilise seulement des primitives pour stocker ses données, puisque mettre à jour un attribut 'age' dans un scope enfant, masquera, mais ne modifiera pas, l'attribut 'age' du scope parent. On se retrouve alors avec des modèles qui ne sont plus en phase. Alors que si l'on utilise des objets, comme 'user.age', alors affecter une nouvelle valeur à 'user.age' modifiera bien la valeur de l'attribut 'user.age' du scope parent.

Par exemple, la directive ngIf, qui permet d'afficher ou non la balise sur laquelle est appliquée selon l'expression qui lui est passée en paramètre, crée son propre scope. Si on modifie le modèle à l'intérieur du ngIf, nous allons avoir un problème si nous avons utilisé une primitive pour le stocker !

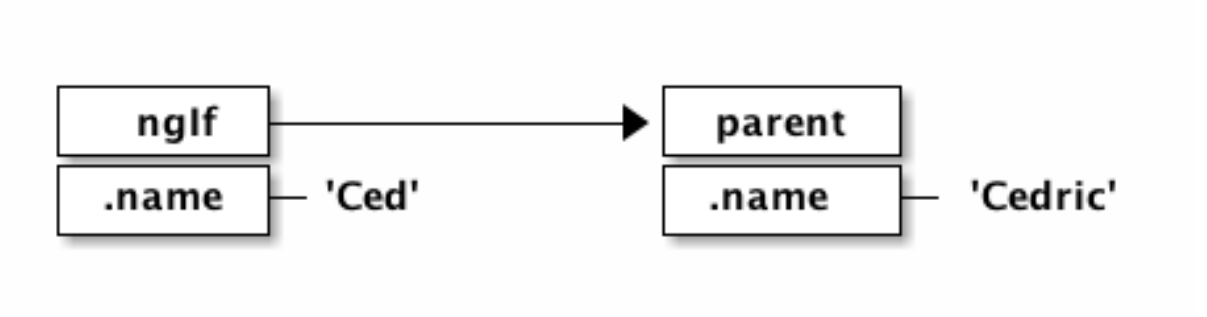
```
<div ng-controller="ParentCtrl">
  {{ name }}
  <div ng-if="true">
    {{ name }}
    <input ng-model="name">
  </div>
</div>


<script>
  var app = angular.module('app', []);
  app.controller('ParentCtrl', function($scope) {
    $scope.name = 'Cedric';
  });
</script>
```

Ici, lorsque nous modifions le modèle grâce à l'input, seule la valeur à l'intérieur du ngIf va être modifiée !

Au début, nous avons :



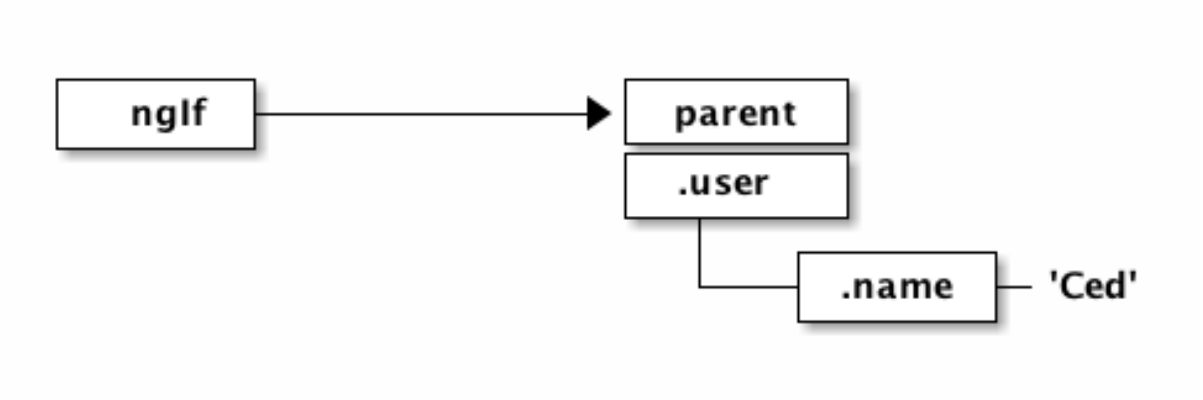
Mais dès que nous modifions la valeur de l'input, seul l'attribut du scope du ngIf est modifié :



En revanche, si nous utilisons un objet et ses attributs pour stocker notre modèle :

```
<div ng-controller="ParentCtrl">
  {{ user.name }}
  <div ng-if="true">
    {{ user.name }}
    <input ng-model="user.name">
  </div>
</div>


<script>
  var app = angular.module('app', []);
  app.controller('ParentCtrl', function($scope) {
    $scope.user = { name: 'Cedric' };
  });
</script>
```

Cette fois, les modifications de l'input impactent bien le modèle du controller ParentCtrl !



Vous pouvez tester cet exemple avec ce Plunker[1].

Il est donc vivement conseillé de toujours stocker ses données dans des objets, pour éviter le problème évoqué ci-dessus.

Si vous vous retrouvez dans un cas compliqué, vous pouvez vous référer à cet article du wiki[2] Angular, extrêmement complet.

Cette subtilité n'est pas tant liée à AngularJS qu'à JavaScript en général, mais, d'après notre expérience, mieux vaut le savoir dès que l'on commence, cela évite de longues heures de debug sans comprendre pourquoi les modèles ne sont plus liés !

Si vous ne devez retenir qu'une chose de cette partie : **Ayez toujours un '.' dans vos ngModel**.

[1] http://plnkr.co/edit/jaWI9ivm9a7MWCKU1qYt?p=preview
[2] https://github.com/angular/angular.js/wiki/Understanding-Scopes

Chapter 4. Filtres

4.1. Principe

L'une des fonctionnalités les plus appréciables et méconnues d'Angular réside dans les filtres disponibles. Dans toute application web, il est courant de devoir transformer des valeurs à l'affichage, ou de filtrer ou réordonner une collection. Les filtres Angular répondent à ce problème et sont applicables à la fois en HTML ou en JavaScript, à un seul élément ou un tableau. En HTML, la syntaxe se rapproche du style Unix, où l'on peut chaîner les commandes à l'aide du pipe. Par exemple, on peut appliquer un filtre nommé 'uppercase' sur une expression de la façon suivante :

```
{{ expression | uppercase }}
```

On peut également chaîner plusieurs appels de filtre :

```
{{ expression | uppercase | trim }}
```

Certains filtres prennent des paramètres : c'est le cas du filtre 'number', qui nécessite de préciser le nombre de chiffres suivant la virgule. Pour passer les paramètres au filtre, il suffit de procéder comme suit :

```
{{ expression | number:2 | currency:'$'}}
```

Si un filtre prend plusieurs paramètres, alors on les sépare également avec des :. C'est le cas du filtre 'orderBy' qui permet de trier un tableau d'objets selon un de leur champ, et dont le deuxième paramètre permet d'inverser le tri.

```
{{ array | orderBy:'name':true }}
```

Il est également possible d'invoquer les filtres depuis le code JavaScript :

```
var toUppercase = $filter('uppercase');
toUppercase('hello'); // HELLO
```

Angular va alors retrouver la fonction de filtre correspondant à la chaîne de caractères passée en paramètre de la méthode $filter(), que l'on peut récupérer et appliquer comme bon nous semble.

4.2. Filtres disponibles

Angular propose par défaut certains filtres communs :

- number : permet de préciser le nombre de chiffres après la virgule à afficher (arrondi au plus proche).

```
{{ 87.67 | number:1 }} // 87.7
```

```
{{ 87.67 | number:3 }} // 87.670
```

- currency : permet de préciser la monnaie.

```
{{ 87.67 | currency:'$' }} // $87.67
```

- date : permet de formatter l'affichage des dates, en précisant un pattern. On retrouve l'écriture classique de pattern :

```
{{ today | date:'yyyy-MM-dd' }} // 2013-06-25
```

Un certain nombre de patterns sont disponibles (avec un rendu différent selon la locale) :

```
{{ today | date:'longDate' }} // June 25, 2013
```

- lowercase/uppercase : de façon assez évidente, ces filtres vont convertir l'expression en majuscules ou minuscules.

```
{{ "Cedric" | uppercase }} // CEDRIC
{{ "Cedric" | lowercase }} // cedric
```

- json : moins connu, ce filtre permet d'afficher l'objet au format JSON. Il est également moins utile, car, par défaut, afficher un object avec la notation '{{ }}' convertit l'objet en JSON.

```
{{ person | json }} // { name: 'Cedric', company: 'Ninja Squad'}
```

```
{{ person }} // { name: 'Cedric', company: 'Ninja Squad'}
```

- limitTo : ce filtre s'applique quant à lui à un tableau, en créant un nouveau tableau ne contenant que le nombre d'éléments passés en paramètre. Selon le signe de l'argument, les éléments sont retenus depuis le début ou la fin du tableau.

```
{{ ['a','b','c'] | limitTo:2 }} // ['a','b']
{{ ['a','b','c'] | limitTo:-2 }} // ['b','c']
```

- orderBy : là encore un filtre s'appliquant à un tableau. Celui-ci va trier le tableau selon l'accesseur passé en paramètre. L'accesseur peut être une chaîne de caractères représentant une propriété des objets à trier ou une fonction. L'accesseur sera appliqué sur chaque élément du tableau pour donner un résultat, puis le tableau de ces résultats sera trié selon l'ordre défini par les opérateurs <, >, =. Une propriété peut être précédée du signe - pour indiquer que le tri doit être inversé. A la place d'une simple propriété, il est possible de passer un tableau de propriétés ou de fonctions (chaque propriété ou fonction supplémentaire servant à affiner le tri primaire). Un second paramètre, booléen, permet quant à lui d'indiquer si le tri doit être inversé.

```
var jb = {name: 'JB', gender: 'male'};
var cyril = {name: 'Cyril', gender: 'male'};
var agnes = {name: 'Agnes', gender: 'female'};
var cedric = {name: 'cedric', gender: 'male'};
$scope.ninjas = [jb, cyril, agnes, cedric];
```

On peut ordonner la liste par genre :

```
// order by the property 'gender'
{{ ninjas | orderBy:'gender'}} // Agnes,JB,Cyril,Cédric
```

Ou par nom :

```
// order by the property 'name'
{{ ninjas | orderBy:'name'}} // Agnes,Cédric,Cyril,JB
```

Pour faire un tri plus compliqué, il est possible de définir son propre comparateur :

```
// order by a function (lowercase last)
$scope.lowercaseLast = function(elem){
  return elem.name === elem.name.toLowerCase()
};
```

```
{{ ninjas | orderBy:lowercaseLast }} // Agnes,JB,Cyril,cedric

// order by an array of properties or functions
{{ ninjas | orderBy:['-gender','name'] }} // cedric,Cyril,JB,Agnes
```

4.3. Filtre dynamique

Nous avons vu les principaux filtres, mais il manque un qui se nomme … 'filter'. Et c'est l'un de nos préférés !

Ce filtre agit sur un tableau pour en retourner un sous-ensemble correspondant à l'expression passée en paramètre. L'expression la plus simple consiste à passer une chaîne de caractères : le filtre va alors retenir tout élément du tableau dont une propriété contient la chaîne de caractères en question.

Il devient alors très simple de faire une recherche dans un tableau dans votre application AngularJS.

Supposons que vous ayez un liste de personnes. Au hasard une équipe de ninjas.

```
$scope.ninjas = [
  { 'name': 'Agnes', 'superpower': 'Java Champion', 'skills':
 ['Java', 'JavaEE', 'BDD']},
  { 'name': 'JB', 'superpower': 'Stack Overflow Superstar', 'skills':
 ['Java', 'JavaScript', 'Gradle']},
  { 'name': 'Cyril', 'superpower': 'VAT specialist' /*I'm joking
 buddy*/, 'skills': ['Java', 'Play!']},
  { 'name': 'Cédric', 'superpower': 'Hype developer', 'skills':
 ['Java', 'JavaScript', 'Git']},
];
```

Maintenant vous voulez les afficher dans un tableau. Facile, un coup de 'ng-repeat' et l'affaire est pliée.

```
<tr ng-repeat="ninja in ninjas">
```

```
  <td>{{ ninja.name }}</td>
  <td>{{ ninja.superpower }}</td>
  <td>{{ ninja.skills.join(',') }}</td>
</tr>
```

Maintenant nous voulons ajouter notre filtre, qui s'utilise comme les autres filtres, sur notre tableau de ninjas. Pour cela nous ajoutons le pipe suivi du filtre 'filter' avec un paramètre nommé search qui contiendra la chaîne de caractères à rechercher dans le tableau (et ne retiendra donc que les ninjas qui satisfont la recherche).

```
<tr ng-repeat="ninja in ninjas | filter:search">
  <td>{{ ninja.name }}</td>
  <td>{{ ninja.superpower }}</td>
  <td>{{ ninja.skills.join(',') }}</td>
</tr>
```

Nous allons ajouter un input qui aura comme modèle 'search' et rendra donc la recherche dynamique.

```
<input ng-model='search' placeholder='filter...'/>
```

Et voilà! Le tableau est filtré dynamiquement ! Essayez, on vous attend ici ![1]

C'est quand même la super classe, non ?

Il est également possible de limiter la recherche à certaines propriétés de l'objet, en passant un objet au filtre plutôt qu'une chaîne de caractères. Par exemple on peut chercher seulement le nom, en transformant l'input comme suit :

```
<input ng-model='search.name' placeholder='filter...'/>
```

Il est également possible de passer une fonction à évaluer contre chaque élément du tableau plutôt qu'une chaîne de caractères ou un objet.

Enfin, ce filtre peut prendre un deuxième paramètre, pouvant être un booléen indiquant si la recherche doit être sensible à la casse (par défaut, elle ne l'est pas) ou une fonction définissant directement comment comparer l'expression avec les objets du tableau. Si vous voulez une recherche sensible à la casse :

[1] http://embed.plnkr.co/qgaSkx/

```
<tr ng-repeat="ninja in ninjas | filter:search:true">
  <td>{{ ninja.name }}</td>
  <td>{{ ninja.superpower }}</td>
  <td>{{ ninja.skills.join(',') }}</td>
</tr>
```

Avec une combinaison de 'filter' et de 'orderBy', vous avez toutes les cartes en main pour faire de superbes listes ou tableaux en HTML, parfaitement dynamiques, et, vous l'avez remarqué, sans écrire une ligne de JavaScript !

4.4. Créer ses propres filtres

Il est également possible de créer ses propres filtres et cela peut être parfois très utile. Il existe une fonction pour enregistrer un nouveau filtre dans votre module (comme nous l'avons vu avec les controllers). Elle se nomme 'filter' et prend comme argument le nom du filtre que vous utiliserez et une fonction. Cette fonction doit renvoyer une fonction prenant comme paramètre l'élément à filtrer. Oui, une fonction qui retourne une fonction, quand on ne fait pas souvent de JavaScript, ça fait bizarre !

Par exemple, nous utilisons souvent la librairie moment.js[2] pour la gestion des dates en JavaScript. Vous ne connaissez pas ? Elle est vraiment très bien faite, et nous est devenue indispensable dès qu'un projet veut faire des choses un peu avancée avec des dates (le filtre par défaut d'Angular sur les dates est assez limité…).

Nous avons donc créé un filtre 'moment' pour Angular, qui nous permet d'utiliser facilement cette librairie dans nos templates.

```
app.filter('moment', function() {
  return function(input, format) {
    format = format || 'll';
    return moment(input).format(format);
  }
});
```

Notre filtre se nomme donc 'moment' et prend 2 paramètres possibles :

- un input, la date à formatter (tous les filtres ont au moins ce paramètre).
- un format, qui est optionnel, et est initialisé à 'll' (l'un des formats offerts par moment.js) si il n'est pas défini.

[2] http://momentjs.com/

Nous pouvons ensuite utiliser ce filtre dans nos templates :

```
{{ '2013-10-15' | moment }} // Oct 15 2013
{{ '2013-10-15' | moment:'LL' }} // October 15 2013
```

L'input est formaté avec 'll' par défaut ou celui précisé, 'LL' dans le deuxième exemple.

Vous savez tout sur les filtres !

Chapter 5. Modules

Angular utilise une logique modulaire pour déclarer les différents composants de l'application et la façon dont ils sont câblés. Il n'y a pas une méthode principale, comme un 'main', qui remplirait ce rôle. Ici chaque module va déclarer quelles dépendances il utilise. Cette méthode a l'avantage d'être simple à comprendre et à utiliser. Elle permet également des tests unitaires simplifiés, en ne chargeant que certains modules, ou en chargeant des modules spécifiques aux tests. Le chargement des modules peut se faire dans un ordre quelconque, et il est également possible de packager des librairies externes sous forme de module.

5.1. Déclarer et récupérer un module

Rien de plus simple, Angular offre une méthode module. On peut ainsi déclarer un module contenant les controllers de l'application :

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

Le deuxième paramètre est un tableau des dépendances du module : il est important de le spécifier, même vide (c'est une erreur fréquente de l'oublier).

Si notre module contenant l'application possède des dépendances, par exemple un module contenant les services et un module contenant les controllers, alors on écrira :

```
// déclaration du module 'services'
var services = angular.module('services', []);
// ajout des services dans le module services
services.service('PersonService', ...);

// déclaration du module 'controllers'
var controllers = angular.module('controllers', []);
// ajout des controllers dans le module 'controllers'
controllers.controller('PersonCtrl', ...);

// ajout des modules à l'application
var app = angular.module('app', ['controllers', 'services']);
```

Pour récupérer un module, on utilise la même méthode, cette fois sans le tableau de dépendances :

```
var services = angular.module('services');
```

Une fois récupéré, il est possible d'y ajouter des composants (controllers, services, filtres, etc.), ou de configurer le module. C'est ce que nous verrons plus bas.

Tous ces modules peuvent être ensuite enregistrés comme dépendances de l'application (qui est elle-même un module). L'application est récupérée par le nom donné à la directive ngApp :

```
<body ng-app="poneyracer">
  ...
</body>

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

En revanche, si vous tentez de récupérer un module pas encore déclaré, Angular lancera une erreur.

5.2. Bonnes pratiques

Il est généralement recommandé d'utiliser plusieurs modules dans votre application, et notamment de séparer les services, des controllers etc… On se retrouve donc généralement avec :

- un module pour les controllers
- un module pour les services
- un module pour les filtres
- un module pour les directives
- enfin, un module pour l'application, qui dépend des modules précédents, et possède la partie du code nécessaire à la configuration de l'application.

Ce découpage permet de faire des tests unitaires n'utilisant que les modules nécessaires, et ignorant généralement le module de l'application, avec le code d'initialisation, difficile à tester. Certains adoptent également une convention de nommage pour leurs modules, en les préfixant avec le nom de l'application (par exemple 'poneyracer.services').

Ce système est un bon moyen de débuter, mais il ne faut pas hésiter à l'adapter à vos besoins lorsque ceux-ci évoluent.

5.3. Utiliser des modules externes

Angular fournit quelques modules optionnels que vous pouvez utiliser :

- ngCookies, pour la gestion des cookies
- ngRoute, nous verrons le principe des routes plus loin
- ngResource, pour interagir avec un serveur REST
- ngMessages, pour les messages dans les templates
- ngAnimate, pour les animations
- ngSanitize, module particulier pour vérifier le HTML.
- ngTouch, pour les évènements liés au mobile (swipe par exemple).

L'équipe Angular maintient ces modules et les package lors de chaque release. Ces modules ajoutent généralement des nouveaux services et/ou directives à utiliser.

Pour ajouter ces modules à votre application, on commence par charger le script dédié au module :

```
<script src="angular.js">
<script src="angular-touch.js">
```

Puis on indique à notre application qu'elle doit utiliser ce module :

```
angular.module('app', ['ngTouch']);
```

La magie d'Angular réside aussi dans sa vaste communauté qui propose de nombreux modules prêts à l'emploi. Nous reviendrons sur les modules les plus indispensables mais on peut d'ores et déjà noter AngularUI Bootstrap si vous voulez utiliser les plugins Javascript de Bootstrap, AngularTranslate pour une meilleure gestion de l'i18n, AngularUI Router pour un module de gestion de route plus puissant que celui par défaut…

La liste est longue, et vous pouvez faire une recherche sur ngmodules.org[1].

[1] http://ngmodules.org

Chapter 6. Tests

Quel que soit le projet, et les langages utilisés, pouvoir tester le code et l'application de manière automatisée est essentiel. L'utilisation de JavaScript ne fait qu'exacerber encore ce besoin. En effet, le code JavaScript n'est pas compilé, n'offre aucune vérification de type, et seule l'exécution du code permet de détecter des erreurs de programmation. Il est donc extrêmement utile de pouvoir, au moindre changement, vérifier que le code fonctionne toujours comme il le devrait, sans avoir à naviguer manuellement dans une série d'écran jusqu'à exécuter le code modifié.

AngularJS a été conçu par des développeurs qui aiment les tests et ont voulu les rendre indispensables à toute bonne application Angular.

Nous distinguerons deux niveaux de tests automatisés :

- les tests unitaires
- les tests end-to-end

6.1. Tests unitaires

Les tests unitaires consistent à vérifier qu'une unité de code (un controller, un filtre, un service, une directive), en isolation des autres, fonctionne correctement. Ecrire un test unitaire d'un controller revient à exécuter chacune de ses méthodes, et à vérifier que les sorties attendues sont obtenues, et que l'état du scope est correct. Dans ces tests, on peut aussi vérifier que les collaborateurs du controller (les services qu'ils utilisent, par exemple) sont utilisés de manière adéquate. Par exemple, on peut vérifier qu'un POST est envoyé via le service $http chaque fois qu'une méthode d'un controller est invoqué. Et cela, sans avoir à réellement mettre en place un serveur web qui va recevoir ce POST.

Les tests unitaires n'offrent que des garanties parcellaires sur le bon fonctionnement de l'application, mais ils ont de gros avantages :

- ils sont très rapides (quelques millisecondes par test)
- ils permettent de facilement couvrir la totalité du code, y compris les cas particuliers qui ne sont pas nécessairement faciles à simuler avec l'application réelle.

L'un des principes de fonctionnement des tests unitaires est l'utilisation de 'mock objects', grâce à l'injection de dépendances. Pour tester un controller utilisant le service $http, par exemple, on lui fournira une fausse implémentation de ce service, qui

permettra simplement de simuler le comportement du service, et de vérifier qu'il est invoqué comme attendu.

6.2. Tests end-to-end

Les tests end-to-end, ou e2e, permettent, comme leur nom l'indique, de tester l'application de bout en bout, comme un testeur humain le ferait : lancer un navigateur, aller à l'adresse de l'application, remplir un formulaire, cliquer sur un bouton, etc. Ils ont l'avantage de tester l'application dans son ensemble, mais ont des inconvénients par rapport aux tests unitaires :

- ils sont bien plus lents (une ou plusieurs secondes par test)
- ils permettent plus difficilement de tester les cas aux limites

Comme vous le devinez, il ne s'agit pas de choisir entre tests unitaires et tests end-to-end, mais de combiner les deux pour obtenir à la fois une couverture de code importante et rapide, et des garanties de bon fonctionnement de l'application dans son ensemble.

Les tests end-to-end peuvent eux-même employer deux stratégies différentes.

La première, la plus naturelle, consiste à réellement faire des tests de bout en bout. L'application réelle est déployée sur un serveur. Des données de test sont insérées dans la base de données, et les tests utilisent l'application comme un humain le ferait. Cette stratégie teste donc non seulement l'interface graphique de l'application (qui est le domaine d'AngularJS), mais aussi le backend, contenant toute la logique métier, l'accès aux données, etc. de l'application. Cette stratégie n'est pas sans difficultés. En effet, il s'agit de mettre en place un jeu de données de test adapté à tous les scénarios à tester. De plus, selon l'ordre d'exécution des tests, les résultats peuvent varier : un test pourrait par exemple tester la consultation d'un utilisateur qu'un autre test vient de supprimer de la base de données pour tester la suppression des utilisateurs.

La deuxième stratégie consiste à se concentrer sur l'interface graphique de l'application, en simulant le backend. Elle consiste à déployer l'application en y ajoutant un module dédié aux tests, et simulant le back-end. Ce module permettra de dire, par exemple: chaque fois qu'un GET est reçu sur l'URL /users, retourne le tableau JSON suivant. Et donc, plutôt que de réellement aller chercher la liste des utilisateurs en base de données, le module retournera une liste, en dur, d'utilisateurs fictifs.

Là encore, les deux stratégies ne sont pas nécessairement mutuellement exclusives. La deuxième, bien que moins réaliste, permet de tester plus rapidement et simplement

l'interface graphique, voire de la tester avant même que le backend soit réellement implémenté.

6.3. Outillage des tests unitaires

Plusieurs librairies de test existent dans l'écosystème JavaScript. L'une des plus utilisées, notamment par AngularJS, est Jasmine[1], que nous allons également utiliser. Jasmine permet d'écrire des tests, et fournit également un exécuteur de ces tests. Mais nous lui préférerons l'exécuteur écrit par l'équipe AngularJS : Karma[2]. Karma est un exécuteur permettant de lancer les tests sur un ou plusieurs navigateurs, et donc de tester l'exécution du code sur tous les navigateurs que votre application doit supporter. Karma offre d'autres avantages :

- il peut lancer les navigateurs automatiquement, ou attendre que vous vous connectiez au serveur Karma avec le navigateur de votre choix pour y exécuter les tests. Vous pourriez donc facilement tester que le code fonctionne sur le navigateur de votre téléphone portable, par exemple.
- il peut surveiller les fichiers JavaScript, et exécuter automatiquement les tests chaque fois qu'une modification est détectée.

La documentation de Jasmine, ainsi que celle de Karma, sont très claires et complètes. Nous n'allons donc pas nous étendre ici sur la manière de les utiliser. Nous allons cependant décrire les étapes nécessaires au test unitaire du composant le plus utilisé dans AngularJS : le controller.

6.4. Test unitaire d'un controller

Supposons que nous ayons écrit le controller suivant, volontairement minimaliste :

```
angular.module('controllers').controller('MessageCtrl', function($scope) {
  $scope.getMessage = function(name) {
    return 'Hello ' + name;
  };
});
```

Il faut commencer par créer un fichier de test. Par convention, nous le nommerons comme le controller, mais avec l'extension .spec.js pour le distinguer facilement dans

[1] http://pivotal.github.io/jasmine/
[2] http://karma-runner.github.io/0.10/index.html

notre IDE : MessageCtrl.spec.js. Ce fichier de test contiendra une suite de tests pour le controller à tester :

```
describe('MessageCtrl', function() {
  var $scope;
});
```

Le controller ajoute des fonctions dans un scope, et il nous faudra donc une variable pour ce scope, afin de pouvoir appeler ces fonctions pour les tester. Un controller plus réaliste stockerait aussi son état dans le scope, et le test pourrait examiner l'état du scope pendant le test.

Avant chaque méthode de test, nous demanderons à Angular de charger le module contenant notre controller :

```
describe('MessageCtrl', function() {
  var $scope;

  beforeEach(module('controllers'));
});
```

Tel est le rôle de la fonction angular.mock.module()[3] de ngMock : charger un module pour l'injecteur. Malheureusement, Angular a choisi d'utiliser le même nom que pour la fonction angular.module()[4], qui est, elle, utilisée pour définir ou référencer un module dans l'application elle-même, ce qui est source de confusion.

Une fois le module chargé, nous allons demander à Angular de créer un injecteur pour le test, et d'injecter les services dont nous avons besoin pour instancier et tester notre controller :

```
describe('MessageCtrl', function() {
  var $scope;

  beforeEach(module('controllers'));

  beforeEach(inject(function($rootScope, $controller) {
    $scope = $rootScope.$new();
    $controller('MessageCtrl', {
      $scope: $scope
```

[3] http://docs.angularjs.org/api/angular.mock.module
[4] http://docs.angularjs.org/api/angular.module

```
    });
  }));
});
```

Examinons ensemble le code ci-dessus.

Il crée une fonction anonyme dépendant de $rootScope et $controller. Ces services sont tous deux fournis directement par Angular. Mais nous pourrions très bien injecter nos propres services, pour peu qu'ils aient été définis dans un module préalablement chargé grâce à la fonction module().

Cette fonction anonyme est passée à la fonction inject()[5]. C'est cette fonction qui se charge de résoudre les dépendances et de les injecter dans notre fonction anonyme.

Notre fonction anonyme, exécutée avant chaque test, effectue les opérations suivantes:

- initialiser la variable $scope avec un nouveau scope, créé par le service $rootScope
- créer une instance du controller nommé 'MessageCtrl'. L'injecteur trouve ce controller dans le module préalablement chargé. Il lui injecte toutes les dépendances dont il a besoin, et déclarées dans le module, excepté la dépendance $scope, qui est le scope créé spécifiquement pour notre test et qui est injecté directement grâce à l'objet passé en argument de $controller().

C'est ce mécanisme d'injection qui permet d'injecter une dépendance réelle (un service, par exemple), définie dans le module, ou un mock, défini dans le test.

Enfin, maintenant qu'un controller est créé pour le test, il faut créer des méthodes de test :

```
describe('MessageCtrl', function() {
  var $scope;

  beforeEach(module('controllers'));

  beforeEach(inject(function($rootScope, $controller) {
    $scope = $rootScope.$new();
    $controller('MessageCtrl', {
      $scope: $scope
    });
```

[5] http://docs.angularjs.org/api/angular.mock.inject

```
  }));

  it('should get the correct message', function() {
    var message = $scope.getMessage('Cedric');
    expect(message).toBe('Hello Cedric');
  });
});
```

Examinons cette méthode de test. Elle appelle la fonction $scope.getMessage(\'Cedric \'). Cette fonction getMessage() est celle qui a été ajoutée au scope lorsque le controller a été initialisé, lors de l'appel à $controller(\'MessageCtrl', ...). La fonction est très simple : elle retourne 'Hello' suivi du nom passé en argument. C'est ce que teste expect(message).toBe(\'Hello Cedric\'). Si le message retourné est bien 'Hello Cedric', le test réussit. Sinon, il échoue.

Et voilà, vous avez votre premier test unitaire ! La syntaxe de l'initialisation est un peu perturbante lorsque l'on débute, et nous reviendrons sur les principes sous-jacents plus tard. Il faut retenir que l'on peut déclarer et tester n'importe quel composant Angular, en lui passant éventuellement de faux objets, que l'on pourra manipuler comme nous voulons.

6.5. Outillage des tests end-to-end

Après avoir longtemps utilisé ngScenario, AngularJS recommande à présent d'utiliser Protractor[6], qui est une API JavaScript permettant de piloter, via Selenium, un navigateur.

Protractor n'a pas encore la maturité d'Angular et de ngScenario et, si le projet est tout à fait utilisable, sa documentation laisse encore à désirer. Heureusement, les exemples de code de la documentation Angular utilisent Protractor, et il vous est donc possible de vous inspirer de ces exemples pour écrire vos propres tests avec Protractor.

Protractor est non seulement une API, mais aussi un exécuteur de test. Au contraire de Karma, qui exécute les tests au sein d'un navigateur, Protractor, lui, exécute les tests dans Node.js. Ces tests communiquent avec un serveur Selenium, qui lance un navigateur. Les tests permettent ensuite, via une API JavaScript, de piloter le navigateur et de lui demander des informations, afin de vérifier que l'application se comporte comme elle le devrait. Il est également possible de débugger l'application pendant un test, ou de prendre des captures d'écran.

[6] https://github.com/angular/protractor

L'installation de Protractor et du serveur Selenium est décrite dans le README[7] du projet, et nous n'allons pas la décrire ici. Nous allons néanmoins détailler les étapes de l'écriture d'un test protractor.

6.6. Test end-to-end avec Protractor

Commençons par créer un fichier de test. L'organisation est libre. Vous pouvez utiliser un fichier de test par écran, ou par groupe d'écrans, ou par scénario fonctionnel. Comme pour les tests unitaires, nous utiliserons Jasmine pour exprimer nos assertions dans le test.

Commençons par écrire une suite de tests:

```
describe('User List', function() {
});
```

Cette suite de test va permettre de tester l'écran de notre application, affichant la liste des utilisateurs. Protractor met à notre disposition une variable globale nommée browser, permettant de piloter le navigateur.

Comme tous les tests de la suite vont tester le même écran, nous pouvons directement naviguer vers cet écran.

```
beforeEach(function() {
  browser.get('/users');
});
```

Dans le code ci-dessus, nous utilisons le pilote pour naviguer vers l'URL /users. L'appel à get() correspond à un utilisateur qui saisirait l'URL /users dans la barre d'adresse de son navigateur, et appuierait sur Enter.

> Seule la fin de l'URL est utilisée, parce que le début, commun à tous les tests, est renseigné dans le fichier de configuration de protractor (par exemple: http://localhost).

On peut ensuite ajouter des tests. Le principe est toujours le même : on recherche un ou plusieurs élément(s) dans la page, on vérifie des assertions sur ces éléments, et on interagit avec eux. Voici un premier test vérifiant simplement que la page contient un tableau de 5 utilisateurs :

[7] https://github.com/angular/protractor/blob/master/README.md

```
it("should display 5 users in the table", function() {
  browser.findElements(by.css("#users tbody tr")).
    then(function(rows) {
      expect(rows.length).toBe(5);
    });
});
```

Examinons ce code ensemble. Il recherche des éléments dans la page (browser.findElements()), en utilisant un sélecteur. Ici, le sélecteur est un sélecteur CSS. Mais nous pourrions aussi rechercher un élément par ID, ou par nom de tag HTML, ou par chemin XPath, ou même en utilisant les propriétés Angular avec une recherche par model (by.model) ou par repeater (by.repeater).

> Plusieurs sélecteurs sont disponibles. La plupart du temps, les développeurs web préfèrent les sélecteurs CSS, couramment utilisés avec jQuery. Souvenez-vous cependant que c'est le navigateur qui va rechercher les éléments grâce à ce sélecteur, et non jQuery. Tous les sélecteurs disponibles avec jQuery ne sont donc pas utilisables ici.

Le sélecteur CSS utilisé ci-dessus recherche donc tous les éléments tr au sein de l'élément tbody de l'élément ayant l'id users. N'hésitez pas à assigner des identifiants (uniques) aux éléments clés de la page. L'écriture des tests s'en trouve grandement simplifiée.

La valeur retournée par findElements() n'est pas un tableau d'éléments. Il s'agit en réalité d'une promesse. L'API est en effet asynchrone. Pour vérifier des assertions sur les éléments, il faut donc les exécuter lorsque la promesse est résolue, en les passant à la méthode then() de la promesse. On peut donc lire le code de la manière suivante : 'Trouve les lignes du corps de la table users, et, lorsque tu les auras trouvées, vérifie que leur nombre est égal à 5'.

Comme les sélecteurs CSS sont les plus utilisés, et que compter les éléments retournés est une opération fréquente, Protractor met à notre disposition des raccourcis :

- $$(selector) permet de trouver 0, un ou plusieurs éléments via un sélecteur CSS. L'objet retourné est une promesse qui a des méthodes supplémentaires: count(), get(), first(), etc. Cette promesse est résolue par un tableau de WebElement.
- $(selector) permet de trouver le premier élément obéissant à un sélecteur CSS. L'objet retourné est une promesse de WebElement.

Le test décrit ci-dessus peut être réécrit plus simplement, et de manière plus concise :

```
it("should display 5 users in the table", function() {
  expect($$("#users tbody tr").count()).toBe(5);
});
```

Ajoutons à présent un test vérifiant qu'il est possible d'ajouter un utilisateur en cliquant sur un lien de la page :

```
it("should allow creating a user", function() {
  browser.findElement(by.linkText("Add user")).click();
  expect($("h1").getText()).toBe("Enter the new user information below");
});
```

Le test ci-dessus trouve le premier lien ayant le texte **Add user**, et clique dessus. Ce lien, théoriquement, provoque l'affichage d'une autre page permettant de créer un nouvel utilisateur. Le test vérifie ce comportement en recherchant le premier élément h1 de la page, et en vérifiant que son texte est égal à 'Enter the new user information below'.

Voilà pour l'essentiel des tests e2e !

Chapter 7. Services

Les services Angular sont des fonctions ou objets avec un seul but bien précis. De nombreux services sont disponibles dans le framework et nous allons pouvoir nous appuyer dessus, comme le service $http par exemple, qui permet de communiquer avec un serveur. Il est bien sûr possible de créer nos propres services. Pour distinguer les services fournis de nos propres services, le nom des services fournis commence toujours par $. L'approche Angular de découpage par service atomique permet aussi de tester très facilement en créant des mocks.

7.1. Injection de dépendances

Derrière ce nom mystérieux si vous ne l'avez jamais utilisé se cache un principe très pratique et répandu dans plusieurs frameworks de différents langages !

7.1.1. Fonctionnement

Commençons par voir comment utiliser un service, par exemple le service $http. Si nous voulons l'utiliser dans un controller, nous écrirons :

```
var MainCtrl = function($scope, $http){
  ...
}
```

On peut ensuite utiliser les objets `$scope` et `$http` dans notre controller. Ok. Angular va, pour nous, rechercher les services dont nous avons besoin, récupérer leur instance et les injecter dans notre controller pour les rendre disponibles. C'est un 'design pattern' très connu et qui mérite un livre à lui tout seul.

Si on écrit les paramètres de notre controller dans l'autre sens, cela fonctionne toujours :

```
var MainCtrl = function($http, $scope){
  ...
}
```

Ce qui est un peu surprenant (habituellement quand on change l'ordre des paramètres d'une fonction, ça a plutôt tendance à mal se passer) !

En revanche, si on change le nom de l'un des paramètres

```
var MainCtrl = function($http, $s){
  ...
}
```

Là plus rien ne fonctionne… En JavaScript l'ordre des paramètres est important et le nom des paramètres ne l'est pas, or il semblerait qu'en Angular le fonctionnement soit différent, et même exactement l'inverse. Angular se base sur le nom des paramètres pour fonctionner et non pas sur leur ordre !

Le principe utilisé est connu sous le nom d'**injection de dépendances**, utilisé dans nombre de langages, plus rarement en JavaScript.

Comment fait donc Angular pour rendre les bons objets disponibles ? Il va procéder en 3 étapes :

- reconnaître les paramètres de la fonction.
- récupérer les objets correspondants auprès de `providers`.
- appeler le controller avec les objets récupérés.

La première étape, la reconnaissance des paramètres, se fait grâce à l'une des particularités de JavaScript. Saviez-vous que les fonctions JS sont des objets et possèdent une méthode `toString()` ?

Si nous appelons le `toString()` sur notre controller, on obtient :

```
MainCtrl.toString() // function($http, $scope){ ... }
```

Angular va alors analyser (à l'aide d'une belle expression régulière que l'on peut voir dans le code de $injector[1]) le résultat du `toString()` et extraire le nom des objets à injecter. Vous pouvez même appeler directement Angular pour vérifier :

```
angular.injector().annotate(MainCtrl); // ["$http", "$scope"]
```

La seconde étape consiste à récupérer les objets en question. Angular utilise pour cela les providers. Si vous demandez l'objet $http, Angular va rechercher un provider nommé $httpProvider et lui demander de créer l'instance de service. Une fois cette instance créée, elle est toujours réutilisée. Tous les controllers partagent donc la même instance du service : c'est un 'singleton'.

[1] https://github.com/angular/angular.js/blob/master/src/auto/injector.js#L64

7.1.2. Le problème de la minification

Les plus avertis auront remarqué que si le nom des paramètres est important, alors on s'expose à de sérieux problèmes si l'on minifie le JavaScript. En effet, lors de la minification, votre controller JavaScript va ressembler à quelque chose comme :

```
var MainCtrl = function(a, b){
  ...
}
```

Et vous l'avez compris, cela va nous poser problème, car Angular va chercher un aProvider et un bProvider qui n'existeront pas ! Pour pallier à ce problème, Angular supporte une autre forme de déclaration :

```
var MainCtrl = ['$http', '$scope', function($http, $scope){
  ...
}];
```

En listant les arguments en chaîne de caractères, vous avez la garantie que vos paramètres seront bien injectés même une fois le code minifié.

```
var MainCtrl = ['$http', '$scope', function(a, b){
  ...
}];
angular.injector().annotate(MainCtrl); // ["$http", "$scope"]
```

Cette fois notre code fonctionne à nouveau ! Cette deuxième forme de déclaration est à privilégier pour éviter tout problème, même si elle est un peu plus verbeuse et contraignante, puisqu'il faut préserver l'ordre entre les chaînes de caractères représentant les services et les paramètres de la fonction où l'on veut les utiliser. Un projet nommé ngmin[2], maintenu par Brian Ford, l'un des membres de la team Angular chez Google, propose de conserver la forme simple de déclaration et d'automatiser la conversion à la seconde forme avant l'étape de minification. C'est en effet ce qui a le plus de sens, mais le projet n'est pas exempt de petites limitations pour l'instant qui empêche de s'appuyer complètement dessus. D'autres projets, comme ngAnnotate[3] apparaissent et sont de plus en plus fiables et rapides.

[2] https://github.com/btford/ngmin
[3] https://github.com/olov/ng-annotate

7.2. Services disponibles

Beaucoup de services sont disponibles dans le framework, depuis les services un peu internes que nous utiliserons très rarement, jusqu'aux utilitaires dont vous ne vous passerez pas. Commençons par ces derniers !

7.2.1. $http

Peut être le service le plus connu, $http nous permet de communiquer avec un serveur. Il permet de cacher toute la plomberie AJAX dans un service facilement utilisable sans avoir à manipuler les XMLHttpRequests. Ce service utilise l'API Promise que nous verrons plus en détail avec le service $q.

Cette API permet de déclarer les callback de succès et d'échec sur l'objet Promise renvoyé par la méthode $http. Par exemple :

```
$http({method: 'GET', url: '/serverUrl'})
  .success(function(data, status, headers, config){ ... })
  .error(function(data, status, headers, config){ ... });
```

La méthode $http va alors appeler l'URL 'serverUrl' avec la méthode 'GET' et renvoyer une 'Promise' qui exécutera soit le callback de succès si tout se passe bien, soit le callback d'erreur. Une requête est considérée en succès si son code HTTP de retour (status) est entre 200 et 299. A noter que les redirections sont gérées de façon transparente sans appeler le callback d'erreur.

Plusieurs méthodes plus simples sont disponibles pour appeler directement la méthode HTTP de votre choix. Ainsi la requête précédente peut s'écrire :

```
$http.get('/serverUrl')
  .success(function(data, status, headers, config){ ... })
  .error(function(data, status, headers, config){ ... });
```

Il est bien sûr possible d'appeler les méthodes put, post, delete, mais aussi head et jsonp.

Les méthodes put et post peuvent recevoir en plus un objet à passer au serveur :

```
$http.post('/serverUrl', { username: 'Cédric' })
  .success(successCallback)
  .error(errorCallback);
```

Il est possible de passer des paramètres. Angular se chargera pour vous de les convertir en chaîne de caractères et de les ajouter à l'url :

```
$http.get('/serverUrl', { params:{key1: value1} })...
```

ce qui donnera l'url 'http://…/serverUrl?key1=value1'.

Il est bien évidemment aussi possible de passer des headers HTTP particuliers. Par défaut, Angular va ajouter un certain nombre de headers pour nous, définis dans $httpProvider.defaults.headers, sur le content-type de la requête pour les put et post (application/json) et le type de la réponse (application/html, application/json, */*). Pour ajouter un header à une requête GET par exemple, il suffit de passer un objet config contenant les headers à la méthode $http :

```
$http.get('/serverUrl', { headers: {'Custom-Header': 'Value'} })...
```

Pour l'ajouter à toutes les requêtes GET, il suffit de faire :

```
$http.defaults.headers.get = {'Custom-Header': 'Value'};
```

A noter également, Angular dispose d'un système de cache interne et il est possible de cacher les requêtes GET :

```
$http.get('/serverUrl', { cache: true })...
```

Il est même possible d'avoir complètement la main sur le cache utilisé en passant une instance d'un cache grâce à $cacheFactory, un autre service chargé de la gestion des caches.

Enfin un timeout en millisecondes peut être donné à la requête et celle-ci sera abandonnée une fois le délai passé.

```
$http.get('/serverUrl', { timeout: 1000 })...
```

La sécurité n'est pas en reste avec notamment un mécanisme intégré pour contrer les attaques Cross Site Request Forgery (XSRF). Pour toute requête, le service va lire un cookie nommé XSRF-TOKEN et en fait un header HTTP (X-XSRF-TOKEN), ce qui assure le serveur que la requête provient bien de son domaine. Ce header ne sera bien sûr pas utilisé pour les requêtes cross-domain. Pour utiliser ce mécanisme, votre

serveur devra donc générer un cookie contenant un XSRF-TOKEN lors de la première requête. Cet identifiant doit être unique par utilisateur, par exemple une combinaison de votre cookie d'authentification avec un 'salt' de sécurité.

7.2.2. $resource

Ce service est à un niveau d'abstraction plus haut niveau que $http (et s'appuie sur lui) et permet d'interagir de façon transparente avec des ressources REST, en procurant directement des méthodes 'get', 'save', 'query', 'remove' et 'delete', qui sont converties par le service en appel à la méthode HTTP adéquate. A noter qu'il fait partie d'un module annexe, ngResource, déclaré dans un fichier JavaScript séparé, et doit donc être ajouté à l'application :

```
<script src="lib/angular-resource.js"></script>
```

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

L'instanciation d'un service $resource se fait en passant l'URL du serveur exposant l'API REST. Ici, on récupère des poneys du serveur, leur id étant défini par le champ id de l'objet.

```
var Poneys = $resource('/poneys/:poneyId', { poneyId: '@id'});
```

On peut éventuellement définir les paramètres par défaut et les méthodes supplémentaires offertes par l'API sur chaque resource (en plus des 'get', 'save', etc…).

```
var Poneys = $resource('/races/:raceId/poneys/:poneyId', { raceId: 24,
poneyId: '@id'}, { run: { method: 'PUT' }});
```

Cette ressource aura donc une méthode supplémentaire run qui correspond à un PUT.

Les paramètres de l'URL sont indiqués avec un préfixe :, et si une valeur de paramètre est préfixée par @ cela indique à Angular que cette valeur sera dans la resource manipulée, au champ en question (par exemple, l'id du poney est donné par son champ 'id').

Ce service permet donc de récupérer les poneys de la course 24, et expose une méthode 'run' sur le poney récupéré.

On peut ensuite créer un poney, le modifier, etc…

```
    var fury = Poneys.save({ name: 'Fury Red'});
    // POST /races/24/poneys/ { name: 'Fury Red'} -> { id: 57, name: 'Fury
Red'}
```

La méthode supplémentaire run est disponible sur chaque ressource avec $run :

```
    fury.$run();
    // PUT /races/24/poneys/57
```

On peut modifier une ressource et la sauvegarder :

```
    fury.name = 'Fury Red Jr';
    fury.$save();
    // POST /races/24/poneys/57 { name: 'Fury Red Jr'} -> { id: 57, name:
'Fury Red Jr'}
```

On peut récupérer une ressource existante et la supprimer :

```
    var cyclone = Poneys.get({ poneyId: 56});
    // GET /races/24/poneys/56 -> { id: 56, name: 'Cyclone'}
    cyclone.$delete();
```

Ou encore récupérer la liste complète des ressources :

```
    // DELETE /races/24/poneys/56
    var poneys = Poneys.query()
    // GET /races/24/poneys/ -> [{id: 57, name: 'Fury Red Jr'}, ...]
```

$resource est donc très pratique si votre serveur suit une API assez proche des concepts REST, sinon, privilégiez l'utilisation du service $http.

7.2.3. $httpBackend

Les services $http et $resource utilisent le service $httpBackend pour faire leurs requêtes. Ce service bas niveau n'est quasiment jamais utilisé par les développeurs directement, à l'exception des tests.

En effet, plutôt que d'utiliser un vrai serveur HTTP, nous allons utiliser une instance mockée de $httpBackend et simuler les requêtes HTTP faites par nos controllers ou services dans les tests !

Prenons l'exemple d'un controller permettant de lister et d'enregistrer les utilisateurs (avec donc une méthode `list` qui fait un appel HTTP GET pour récupérer la liste des utilisateurs et l'affecter à une variable `users` du scope, et une méthode `register`, qui, elle, fait un HTTP POST pour enregistrer un utilisateur) .

Créons un test :

```
describe('Controller: MainCtrl', function () {

  // load the controller's module
  beforeEach(module('controllers'));

  var scope, $httpBackend;

  // Initialize the controller and a mock scope
  beforeEach(inject(function ($controller, $rootScope, _$httpBackend_) {
    scope = $rootScope.$new();

    $httpBackend = _$httpBackend_;

    $controller('MainCtrl', {
      $scope: scope
    });
  }));
});
```

Nous allons avoir besoin du module 'controllers'. Ce module est donc chargé grâce à la méthode beforeEach(module(...)).

Nous aurons également besoin de faire un faux backend HTTP pour simuler les requêtes serveurs : c'est ici qu'intervient le service $httpBackend.

Pour pouvoir les manipuler dans nos tests nous déclarons des variables locales avec leurs noms, puis nous injectons les valeurs dans ces variables, grâce à inject(). Angular propose une astuce pour injecter $httpBackend dans $httpBackend (il faudrait sinon une notion comme le this en Java), en nommant la variable à injecter avec des '_' de part et d'autres. A la fin du 2ème beforeEach, nous avons donc nos variables initialisées.

Le service `$rootScope` permet de créer un scope sur lequel nous pourrons appeler les méthodes exposées par le controller et vérifier si les valeurs du scope sont bien celles attendues.

Le service `$controller` injecté permet quand à lui de créer le controller que nous voulons tester en lui passant le scope créé.

Nous pouvons donc écrire notre premier test :

```
it('should fetch 2 users when list is called', function () {
  // given an http backend that receives a request and return 2 users
  $httpBackend.expectGET('http://localhost:8080/poneyserver/users')
    .respond([ced, jb]);

  scope.list();
  // we need to flush the request
  $httpBackend.flush();

  // we should have users
  expect(scope.users.length).toBe(2);
});
```

Le test commence par mocker l'appel serveur : un appel GET à l'URL donnée est attendu et renverra maintenant systématiquement les 2 users définis.

Le test fait donc un appel HTTP et affecte le résultat à la variable users. L'appel à la méthode flush permet de déclencher les réponses de toutes les requêtes en attente (et simplifie l'écriture de nos tests pour ne pas avoir à gérer l'asynchronisme). De plus, `flush` vérifiera que la requête attendue a bien été reçue. Ainsi, si la requête GET n'a pas été faite par le controller, alors le test échouera.

On vérifie ensuite que nous avons bien deux users sur le scope.

On peut ensuite écrire un test équivalent pour l'enregistrement :

```
it('should register user', function () {
  // given
  $httpBackend.expectPOST('http://localhost:8080/poneyserver/users',
 ced).respond('ced');
  var login;

  // when
  login = scope.register(ced);
  $httpBackend.flush();

  // then
  expect(login).toBe('ced');
```

```
});
```

Ce test fonctionne sensiblement de la même manière : nous définissons l'attente d'une requête POST cette fois, puis nous appelons la méthode register, nous déclenchons le flush et enfin nous vérifions que l'utilisateur enregistré est bien celui retourné par l'appel HTTP.

L'utilisation du $httpBackend est très pratique : le seul piège lorsque l'on débute est d'oublier l'appel au flush. Dans ce cas, la réponse n'est pas retournée, et l'on perd du temps à comprendre pourquoi. Maintenant, vous saurez !

7.2.4. $location

Le service $location rend accessible l'URL de la page à notre application et donne un certain nombre de méthodes utilitaires pour le parsing. Il est également possible de changer l'URL, sans déclencher de rechargement (pratique, puisque nous sommes dans une application Single Page). C'est un wrapper amélioré de window.location, nous donnant des getters et setters, connaissant le cycle de vie Angular et étant compatible avec les API HTML5 de navigation. Dès que vous avez besoin de connaître l'URL ou de la changer, c'est ce service qu'il vous faut!

Pour Angular, l'URL est constitué de 6 parties : 'protocol', 'host', 'port', 'path', 'search' et 'hash'. Par exemple dans http://ninja-squad.com/training?subject=angular#agenda, Angular considère le protocole comme étant 'http', le 'host' comme étant 'ninja-squad.com', le port comme étant '80', le path comme étant '/training', 'search' comme étant 'subject=angular' et le 'hash' comme 'agenda'. L'URL complète peut être récupérée grâce à 'absUrl()', et chacune des sous-parties est également accessible grâce aux méthodes éponymes.

Pour récupérer le 'path' courant :

```
    $location.path();
```

Ou le changer

```
    $location.path("/newPath");
```

Toutes les valeurs qui sont passées aux méthodes de $location sont encodées correctement par Angular. Utilisé conjointement avec un routeur, ce service permettra de déclencher une navigation entre nos vues depuis nos controllers.

7.2.5. $timeout

Il est fréquent de devoir différer l'exécution d'une action en JavaScript. La fonction habituellement utilisée est 'setTimeout', et Angular propose un wrapper pour celle-ci, sous la forme du service $timeout. Le wrapper gère correctement les exceptions, connaît le cycle de vie Angular et renvoie une Promise qu'il est possible d'annuler grâce à la méthode cancel(). Cette dernière fonction renvoie vrai si le timeout ne s'est pas encore exécuté et a bien été annulé.

```
    var delayedFn = $timeout(function(){ ... }, 1000) // set a timeout of
1 sec
    var canceled = $timeout.cancel(delayedFn); // cancel the timeout
```

7.2.6. $interval

La fonction JavaScript 'setInterval' a également son wrapper : #0#. Ce service déclenchera l'exécution de la fonction toutes les X millisecondes, un certain nombre de fois ou indéfiniment. Son enregistrement renvoie également une Promise qu'il est possible d'annuler. Il faut également penser à annuler ce genre de fonction récursive lorsqu'elle n'est plus utile (à la destruction du controller appelant par exemple [4]).

```
    var recurringFn = $interval(function(){ ... }, 1000, 0) // every 1
sec, infinitely
    var canceled = $interval.cancel(recurringFn); // cancel the recurring
function
```

7.2.7. $filter

Nous avons déjà vu son utilisation, sans savoir que c'était un service : voir le paragraphe sur l'utilisation des filtres.

7.2.8. $locale

Le support i18n d'Angular est très rudimentaire et ne concerne que les dates et les nombres. Pour plus de possibilités, le projet le plus adapté est angular-translate[5]. Toujours est-il qu'il est possible de changer la locale utilisée par Angular, pour le format des nombres par exemple, grâce au service $locale.

[4] Voir le chapitre 'Evénements'
[5] https://github.com/PascalPrecht/angular-translate

7.2.9. $log

Pratique, ce service wrappe la console du navigateur (si elle existe). Vous ne devriez jamais utiliser console.log directement dans votre application, mais plutôt utiliser le wrapper. Cela permet d'écrire nos logs de debug par exemple :

```
$log.debug('Log de debug');
```

Le logger fournit les méthodes 'debug', 'info', 'warn', 'error', 'log'. Il est possible de désactiver les logs de debug grâce à $logProvider.debugEnabled(false).

7.2.10. $cookies/$cookieStore

Ce service permet d'écrire et de les lire les cookies du navigateur. Très simple à utiliser, il est disponible dans un module séparé (`ngCookies`) à inclure :

```
// read
$scope.login = $cookies.login;
// write
$cookies.login = $scope.login;
```

Il est également possible d'utiliser $cookieStore, qui permet de stocker des objets dans des cookies en les sérialisant/désérialisant en JSON :

```
// read
$scope.user = $cookieStore.get('user');
// write
$cookieStore.put('user', $scope.user);
// remove
$cookieStore.remove('user');
```

Les fonctionnalités sont très limitées, si vous cherchez plus de possibilités, comme gérer l'expiration par exemple, il existe d'autres modules comme angular-cookie[6].

7.2.11. $route/$routeParams

Une partie spécifique est dédiée à ces services et au concept de route de façon plus générale.

[6] https://github.com/ivpusic/angular-cookie

Tous ces services sont très pratiques, mais vous pouvez aussi bien sûr développer les vôtres, qu'il s'agisse de services techniques (manipuler le localStorage par exemple) ou fonctionnels (gérer le métier de votre application).

7.3. Créer ses services

7.3.1. Construire un service

Construire un service se résume à créer une fonction ou un objet exposant des fonctions, à lui donner un nom et à l'enregistrer auprès du framework (comme nous l'avons fait avec les controllers et les filters) pour le rendre accessible au reste de l'application. Cette fonction ne sera instanciée qu'une seule fois pendant la vie de votre application : tous les controllers qui utiliseront votre service partageront la même instance de celui-ci.

Prenons le cas d'un service permettant de stocker et de lire des valeurs dans le localStorage du navigateur. L'objet localStorage est accessible grâce au wrapper `$window` du framework.

Le service va donc ressembler à :

```
function($window){

  var save = function(key, value){
    $window.localStorage.setItem(key, angular.toJson(value));
  };

  var read = function(key){
    var json = $window.localStorage.getItem(key);
    return angular.fromJson(json);
  };

  return {
    save: save,
    read: read
  }
}
```

Les deux méthodes sont exposées grâce au Revealing Module Pattern. C'est à dire que l'on peut choisir de n'exposer que certaines méthodes aux consommateurs de notre service : vous pouvez y voir l'équivalent JavaScript de méthodes publiques.

Il serait possible de les exposer avec un nom différent également :

```
function($window){

  var s = function(key, value){
    $window.localStorage.setItem(key, angular.toJson(value));
  };

  var r = function(key){
    var json = $window.localStorage.getItem(key);
    return angular.fromJson(json);
  };

  return {
    save: s,
    read: r
  }
}
```

Les méthodes s'appellent 's' et 'r' mais sont exposées publiquement sous les noms 'save' et 'read'.

Si l'on voulait garder une méthode secrète, nous pourrions simplement ne pas la retourner :

```
function($window, $log) {

  var save = function(key, value){
    logAccess(key, 'saved');
    $window.localStorage.setItem(key, angular.toJson(value));
  };

  var read = function(key) {
    logAccess(key, 'read');
    var json = $window.localStorage.getItem(key);
    return angular.fromJson(json);
  };

  var logAccess = function(key, op) {
    $log.debug('The key ' + key + ' has been ' + op + ' in local
 storage');
  };

  return {
    save: save,
    read: read
```

```
  }
}
```

Ainsi, la fonction nommée 'logAccess' ne sera pas accessible à l'extérieur du service, mais quand même utilisable en interne : c'est l'équivalent d'une méthode privée dans la plupart des langages orientés objets.

Notre service est terminé, il faut maintenant procéder à son enregistrement. Comme nous le verrons avec la partie consacrée aux modules, il est de bon ton de déclarer un module 'services' dans lequel nous pourrons enregistrer notre service.

```
angular.module('services').factory('localStorage',
 ['$window', function($window){
  var save = function(key, value){
    $window.localStorage.setItem(key, angular.toJson(value));
  };

  var read = function(key){
    var json = $window.localStorage.getItem(key);
    return angular.fromJson(json);
  };

  return {
    save: save,
    read: read
  }
}]);
```

Nous utilisons la méthode factory pour enregistrer notre service sous le nom localStorage. Il est maintenant possible de l'utiliser dans les controllers, directives ou autres services :

```
function UsersCtrl($scope, localStorage){
  $scope.name = localStorage.read('name');
  $scope.save = function(){
    localStorage.save('name', $scope.name);
  }
}
```

7.3.2. Tester un service

Il est important de conserver une bonne couverture de tests sur les services qui sont généralement utilisés de façon transverse dans l'application. Angular permet de faire très facilement des tests de service, profitons-en !

Créons un test :

```
describe('Service: localStorage', function () {

  // load the service's module
  beforeEach(module('services'));

  var localStorage, $window, fakeStore;

  // ask the service to angular
  beforeEach(inject(function(_localStorage_, _$window_) {
    localStorage = _localStorage_;
    $window = _$window_;
    fakeStore = {};

    spyOn($window.localStorage, 'setItem').andCallFake(function(key,
 value) {
      fakeStore[key] = value;
    });
    spyOn($window.localStorage, 'getItem').andCallFake(function(key) {
      return fakeStore[key];
    });
  }));

  it('should save a value', function() {
    var value = { key1: 'value1' };
    localStorage.save('key', value);

 expect($window.localStorage.setItem).toHaveBeenCalledWith('key', '{"key1":"value1"}');
    expect(fakeStore['key']).toBe('{"key1":"value1"}');
  });

});
```

L'object `$window.localStorage` est un objet mocké par Jasmine, déclarant les fonctions `getItem` et `setItem` dont on remplace le comportement. Utiliser un spy Jasmine permet de court-circuiter les vrais appels et faire des assertions du type 'toHaveBeenCalled' pour vérifier l'appel ou les paramètres.

Le test est ensuite très simple : on appelle la méthode de notre service et on vérifie que celui-ci a bien effectué la sérialisation JSON et le stockage.

7.3.3. Enregistrer un service

On enregistre généralement un service dans un module dédié. Le module expose des méthodes d'enregistrement différentes selon le type de service que l'on veut créer : nous avons vu la méthode factory() mais il en existe d'autres.

Un service Angular est un singleton, créé par une 'factory' (design pattern très fréquemment utilisé) : c'est à dire que toute utilisation d'un service se fera sur une seule instance de ce service à travers toute l'application, contrairement à un controller par exemple qui sera instancié à chaque fois que la vue en aura besoin.

Cette factory va être fournie par un 'provider', c'est à dire un constructeur de service. Lorsque Angular (plus précisément l'injecteur) veut un nouveau service, il cherche le provider adéquat, récupère la factory et construit une instance du service. Un provider doit donc exposer une fonction '$get' qui constitue sa 'factory'.

En théorie, pour créer un service, il faudrait donc faire un provider, qui fournit une factory, qui elle serait chargée de créer le service. Dans la réalité, bien que tout cela soit possible, on fait bien plus simple, comme vous l'avez vu dans la partie précédente.

Mais, pour être exhaustif, pour enregistrer un nouveau composant dans l'application, le module expose les méthodes suivantes :

- 'provider()' la méthode de plus bas niveau, pour enregistrer un provider.
- 'factory()' pour enregistrer directement une factory sans s'occuper du 'provider' enveloppant.
- 'service()' pour enregistrer un service via son constructeur.
- 'constant()' pour enregistrer une constante accessible par les services et providers.
- 'value()' pour un objet accessible par les services mais pas les providers

Donc, en vrai, vous n'allez utiliser que 'service()' ou 'factory()' et éventuellement 'constant()' de temps en temps. Le reste est de l'ordre du très exceptionnel.

On peut donc enregistrer une constante (valeur ou fonction) :

```
angular.module('services').constant('HOST', 'localhost');
```

```
angular.module('services').constant('square', function(x){
  return x*x;
});
```

Pour utiliser la méthode 'service', on définit un type custom, dont le constructeur sera appelé (et injecté) par Angular :

```
function LoginService($http, SERVER) {
  this.login = function(user, pwd) {
    return $http.post(SERVER + '/login', {user: user, password: pwd});
  };
}
angular.module('services').service('LoginService', ['$http', 'SERVER',
 LoginService]);
```

La seconde méthode consiste à utiliser une factory. Une factory doit renvoyer le service à enregistrer. Pour créer un service 'LoginService' avec une méthode 'login', on peut écrire :

```
angular.module('services').factory('LoginService',
 ['$http', 'SERVER',function($http, SERVER){
  return function login(user, pwd) {
    return $http.post(SERVER + '/login', {user: user, password: pwd});
  };
}]);
```

On utilise plus fréquemment le 'revealing module pattern' qui retourne un objet avec les méthodes exposées :

```
angular.module('services').factory('LoginService',
 ['$http', 'SERVER', function($http, SERVER){
  function login(user, pwd){
    return $http.post(SERVER + '/login', {user: user, password: pwd});
  };
  return {login: login};
}]);
```

Nous aimons bien utiliser la méthode 'factory()' avec ce pattern plutôt que 'service()', mais les deux sont sensiblement équivalentes.

7.4. Promises

Parlons un peu des Promises, qui sont utilisées par tous les services Angular qui font des traitements asynchrones, et que vous voudrez peut-être utiliser dans vos propres services. Même si ce n'est pas le cas, les promises sont un concept très intéressant, et seront bientôt disponibles dans le langage JavaScript, même si vous ne faites pas du AngularJS.

A la différence d'un callback, les promises permettent la composition. C'est d'ailleurs l'un de leurs avantages sur les callbacks, puisque nous allons pouvoir enchaîner les appels sans tomber dans le 'Callback Hell', l'enfer des callbacks imbriqués les uns dans les autres, bien connu des développeurs JavaScript.

Lorsqu'un appel asynchrone est créé, on peut l'envelopper dans une promise qui peut avoir deux résultats :

- la promise est 'fulfilled' (appel avec succès)
- la promise est 'rejected' (appel en échec)

Elle peut être dans deux états :

- 'pending', en attente
- 'settled', terminée (en échec ou en succès)

Une promise est un objet 'thenable', c'est à dire qu'elle possède une méthode 'then'. Une méthode 'then' prend comme arguments une fonction pour le cas 'fulfilled' et une fonction pour le cas 'rejected'. Chaque méthode 'then' renvoie une promise, on peut donc chaîner (composer) les appels.

```
var promise1 = new Promise(function(resolve, reject){
  //async stuff returning data
  if(success){
    resolve(data);
  } else {
    reject(errorData);
  }
});
var promise2 = promise1.then(fulfilledCallback, rejectedCallback);
```

Exemple pratique :

```
function fetchTime(delay) {
  return new Promise(function(resolve, reject){
    window.setTimeout(function(){
      resolve(new Date());
    }, delay);
})};
function logTime(time){
  console.log(time);
  return time;
};

function logSeconds(time){
  console.log(time.getSeconds());
};

// chaining
fetchTime(2000).then(logTime).then(logSeconds)
```

En JavaScript classique, l'API Promise utilisée dans l'exemple ci-dessus vient d'être introduite dans le langage pour la prochaine version (EcmaScript 6).

La librairie Q existe depuis quelques temps et Angular l'a réécrite, en version plus réduite, dans un service, nommé $q, afin de pouvoir utiliser ce concept de promise sans attendre que tous les navigateurs supportent EcmaScript 6. Cette librairie n'est pas exactement comme l'API JavaScript, mais s'en rapproche fortement. Voici à quoi ressemble un appel avec promise et le service $q, dont le service $http se sert :

```
$http.get(...)
  .then(function(data){
    // success, promise resolved
  }, function(error){
    // error, promise rejected
  });
```

$http.get renvoie une promise et l'on peut donc passer un callback de succès et un callback d'échec à la méthode 'then'. Les promesses retournées spécifiquement par le service $http ont un peu de sucre syntaxique supplémentaire, et permettent d'écrire :

```
$http.get(...)
  .success(function(){
    // success, promise resolved
  })
```

```
  .error(function(){
    // error, promise rejected
  });
```

Mais fondamentalement, quand on utilise un service qui fait de l'asynchrone en AngularJS, on manipule des promises !

Mais peut-être voudriez vous créer des promises dans vos services ?

7.4.1. $q

Ce service sert beaucoup en interne du framework pour utiliser l'API Promise : tous les appels asynchrones passent par cette API.

Il est possible de créer des promises dans nos services à l'aide de $q. Créons un service permettant de lancer un dé sur une table. Le dé renvoie son résultat 2 secondes après qu'il ait été lancé.

Pour créer une promise, on utilise la méthode 'defer' :

```
function($q, $timeout) {
  var roll = function() {
    var deferred = $q.defer();

    $timeout(function() {
      var value = Math.floor(Math.random() * 7) + 1;
      if (value < 7) {
        deferred.resolve(value);
      } else {
        deferred.reject('The dice fell off the table');
      }
    }, 2000);

    return deferred.promise;
  };

  return {
    roll: roll
  };
}
```

Ce service expose une méthode asynchrone 'roll' qui renvoie une promise. Elle attend 2 secondes avant d'appeler la méthode 'resolve', c'est-à-dire la méthode qui passe la

promise du statut 'pending' à 'fulfilled' si la valeur du dé est inférieure à 7, ou 'rejected' si la valeur est supérieure ou égale à 7.

Pour appeler ce service, il faudra donc adopter la syntaxe 'then' :

```
dice.roll()
  .then(function(value) {
    $scope.value = value;
    return "You're a good dice roller";
  }, function(message) {
    $scope.value = undefined;
    $scope.error = message;
  })
  .then(function(successMessage) {
    $scope.success = successMessage;
  });
```

On peut également renvoyer des updates sur l'avancement de l'appel grâce à 'notify'. On peut ainsi indiquer au bout d'une seconde que le dé est toujours en train de rouler, et passer à la fonction 'then' un 3ème callback pour faire quelque chose de ces updates :

```
function($q, $timeout) {
  var roll = function() {
    var deferred = $q.defer();

    $timeout(function() {
      deferred.notify('Rolling...');
    }, 1000);

    $timeout(function() {
      var value = Math.floor(Math.random() * 7) + 1;
      if (value < 7) {
        deferred.resolve(value);
      } else {
        deferred.reject('The dice fell off the table');
      }
    }, 2000);

    return deferred.promise;
  };

  return {
    roll: roll
  };
```

```
    }
```

On passe alors un 3ème callback à 'then' pour profiter de cette notification :

```
    dice.roll()
      .then(function(value) {
        $scope.value = value;
      }, function(message) {
        $scope.value = undefined;
        $scope.error = message;
      }, function(notification) {
        $log.debug(notification);
      });
```

L'API $q propose aussi une méthode 'all' pour attendre le résultat de plusieurs promises. Admettons que nous voulions lancer deux dés, et attendre que chacun d'eux ait fini de rouler, il serait alors possible d'écrire :

```
    $q.all([dice.roll(), dice.roll()])
      .then(function(values){
        // values is an array of 2 values, each returned by one of the
dices
        $scope.value = values[0] + values[1];
      });
```

L'objet promise offre également les méthodes 'catch' et 'finally'. Sans surprise, 'catch' revient à passer un callback d'erreur, uniquement, et 'finally' permet de passer un callback appelé quel que soit le résultat de l'exécution asynchrone (succès ou erreur).

Au delà de leur utilisation dans AngularJS, les promises sont maintenant une réalité en JavaScript et seront bientôt utilisées dans vos librairies préférées : c'est donc un point important à connaître même si vous ne les utilisez que peu dans vos applications AngularJS.

7.5. Configuration d'un service

Un module peut posséder un ensemble de blocs de configuration, 'config', et un ensemble de blocs d'exécution, 'run'. Quelle est la différence entre ces deux types ? Les premiers blocs, 'config', seront exécutés pendant la phase d'enregistrement des services et ne pourront d'agir que sur des providers (par exemple $httpProvider). Les

seconds, 'run', seront exécutés une fois tous les providers enregistrés et configurés, et ne pourront agir que sur des services (par exemple $http), et plus sur leurs providers.

7.5.1. config

C'est l'endroit où vous allez pouvoir agir sur les providers. Par exemple, si vous désirez changer '{{' pour le remplacer par '[[' dans vos templates, il faut paramétrer le provider $interpolateProvider.

```
var app = angular.module('poneyracer');

app.config(function($interpolateProvider){
  $interpolateProvider.startSymbol = '[[';
  $interpolateProvider.endSymbol = ']]';
});
```

7.5.2. run

La deuxième méthode est elle une configuration au runtime de l'application, lorsque les providers ont été configurés et les services prêts à être utilisés. Ici, il ne sera plus possible d'agir sur les providers, mais seulement sur les services. Ce code contiendra ce qui est nécessaire au démarrage de votre application et doit être effectué à chaque chargement.

Par exemple, vous pouvez vouloir ajouter systématiquement un header HTTP à vos requêtes, ce header étant peut-être stocké dans un cookie (mécanisme fréquent pour la gestion de l'authentification).

```
app.run(function($http, $cookies){
  $http.defaults.headers.common['Custom-Authentication'] =
 $cookies.customAuth;
});
```

Ainsi, à chaque redémarrage de l'application, le cookie 'customAuth' sera relu et ajouté comme header HTTP 'Custom-Authentication' à toutes les requêtes.

Vous connaissez maintenant les principaux services, comment les configurer et comment créer vos propres services!

Chapter 8. Routes

Une des parties essentielles de l'application que vous rêvez d'écrire concerne bien sûr la navigation entre les écrans. Nous avons vu que nous pouvions nous appuyer sur des templates pour nos vues et que les controllers régissaient leurs comportements. Il est peu probable que votre application ne comprenne qu'un seul écran. Mais comment indiquer à Angular que telle URL doit afficher tel template avec tel controller ? Rappelez-vous que l'idée est de réaliser une application 'single-page'. On ne va donc pas recharger complètement une nouvelle page et redémarrer une autre application Angular. On va en revanche simplement rester dans la même application, mais changer de vue principale et de controller. C'est ici que le router d'Angular entre en jeu.

Dès que l'application se complexifie un peu, vous allez avoir à jongler entre différents écrans et le module ngRoute vous deviendra indispensable. Du moins jusqu'à ce que vos attentes dépassent les fonctionnalités offertes par le router du framework et que vous ayez peut-être besoin du routeur de AngularUI, plus complet. Dans un premier temps, le routeur d'Angular nous suffit.

Ce routeur est présent dans un module annexe, nommé ngRoute qu'il faudra donc charger dans votre application. Une fois disponible, de nouveaux services nous sont accessibles, $route et son provider $routeProvider, routeParams, ainsi qu'une nouvelle directive ngView.

8.1. Déclarer une route

Le provider $routeProvider, qu'il s'agit donc de configurer grâce à un bloc 'config', nous permet d'enregistrer les routes de notre application : en fait un lien entre une URL, un template et un controller. Le template adéquat sera ensuite inséré dans la page lorsque l'URL correspondra à celle d'une route, et le controller sera instancié.

Pour savoir quelle partie de la page doit être remplacée par le template, Angular utilise la directive ngView qui sert de point d'insertion.

```
<div ng-view></div>
```

La déclaration des routes est très simple grâce au provider :

```
$routeProvider
```

```
  .when('/races', {
    templateUrl: 'races.html',
    controller: RacesCtrl
  })
```

Ainsi si l'utilisateur accède à l'URL '/index.html#/races' alors la la vue 'races.html' sera chargée et le controller 'RacesCtrl' instancié. La navigation vers une telle URL pourra se faire depuis un autre controller avec location.path("/races") ou depuis le HTML avec un simple lien Races.

Note sur les URL : par défaut, lorsqu'on navigue dans une application Angular, la partie de l'URL qui est modifiée est le hash. Ainsi, passer de la liste des courses au détail d'une course fera passer l'URL de '/index.html#/races' à '/index.html#/races/12' par exemple. Ce mode de navigation a l'avantage d'être supporté même par de vieux navigateurs. Un autre avantage est qu'un rafraichissement de la page rechargera la page index.html depuis le serveur (l'unique page de notre application), puis affichera la vue correspondant au chemin derrière le hash.

Il est possible de configurer le $locationProvider en mode 'html5', afin que les URLs soient plus jolies: '/races' et '/races/12', par exemple. Attention cependant: si l'utilisateur rafraichit la page, une requête HTTP sera alors envoyée au serveur pour l'URL '/races/12'. Il faudra donc que votre serveur soit configuré pour renvoyer le contenu de index.html pour cette URL (et toutes les autres URLs de votre application), afin que l'application redémarre et réaffiche ensuite la vue correspondant à cette URL.

Il est possible d'avoir des URLs paramétrées, et de récupérer ensuite les paramètres dans les controllers (nous verrons comment) :

```
$routeProvider
  .when('/races', {
    templateUrl: 'races.html',
    controller: RacesCtrl
  })
  .when('/races/:raceId', {
    templateUrl: 'race.html',
    controller: RaceCtrl
  })
```

Dans cette deuxième route, un identifiant de course raceId sera nécessaire, comme '/races/12' ou '/races/55'.

Il est aussi possible de matcher un ensemble de routes :

```
$routeProvider
  .when('/races/:raceId/p*', {
    templateUrl: 'poneys.html',
    controller: PoneysCtrl
  })
```

Dans ce cas, les URLs '/races/12/poneys' ou '/races/12/pi' déclencheront cette route.

Enfin, la méthode otherwise permet de définir une route lorsque l'URL ne déclenche aucune route :

```
$routeProvider
  .otherwise({
    redirectTo: '/'
  })
```

Les paramètres d'une route sont donc :

- le template (par son URL ou directement) et le controller associé (éventuellement par son alias)
- ou l'URL de redirection
- éventuellement une propriété 'resolve' qui, comme son nom l'indique, va résoudre ses différentes propriétés avant de d'instancier la vue. Cela peut être pratique pour charger des données nécessaires à l'affichage de la vue, puisque si la propriété est une promise, Angular va attendre sa résolution.
- la propriété 'caseInsensitiveMatch' est par défaut fausse, mais il est possible de la paramétrer pour matcher des routes sans tenir compte de la casse
- enfin la propriété 'reloadOnSearch' est vraie par défaut, mais peut être passée à faux pour déclencher un rechargement lorsque la partie 'search' de l'URL change.

8.2. Récupérer les paramètres

Le service $route expose une méthode reload qui force le rechargement de la vue, et la réinstanciation du controller. Ce service possède également deux propriétés :

- 'current' qui est la route courante, sur laquelle on peut récupérer les paramètres de la route.
- 'routes' qui contient la liste de toutes les routes.

Mais ce service est peu utilisé, car les controllers peuvent récupérer les paramètres en utilisant le service $routeParams qui donne un dictionnaire clé/valeur des paramètres de l'URL ('path' et 'search').

Par exemple, avec l'URL 'http://ninja-squad.com/training/angular?topic=agenda', si nous avons une route avec le chemin '/training/:subject', alors l'objet $routeParams contiendra :

```
{ subject: 'angular', topic: 'agenda'};
```

Si l'on reprend notre exemple de courses :

```
$routeProvider
  .when('/races', {
    templateUrl: 'races.html',
    controller: RacesCtrl
  })
  .when('/races/:raceId', {
    templateUrl: 'race.html',
    controller: RaceCtrl
  })
```

Alors 'RaceCtrl' pourrait très simplement récupérer l'identifiant de la course comme suit :

```
angular.module('controllers').controller('RaceCtrl', function($scope,
 $routeParams){
  $scope.raceId = $routeParams.raceId; // URL /races/12 -> raceId = 12
});
```

Ce module devient rapidement indispensable quand l'application se développe. Encore une fois, il n'offre que des fonctionnalités minimales, et vous devrez peut-être vous tourner vers le routeur d'AngularUI pour des choses plus avancées.

Chapter 9. Directives

Ha, les directives ! Nous en avons déjà vu plusieurs, vous savez, les mot clés à mettre dans nos templates. Voyons maintenant plus en détail comment elles fonctionnent, quelles sont celles qui nous sont fournies et comment créer les nôtres.

Une directive est un comportement que vous voulez ajouter ou un composant réutilisable dans votre application. Elle consiste en un ensemble de méthodes qui régissent son comportement et éventuellement en un template qui sera injecté dans votre HTML.

Le futur du Web passera probablement par les Web Components. Vous en avez peut-être entendu parler, cette spécification[1], encore en cours d'élaboration, pourrait devenir un standard dans les prochaines années et changer notre façon de concevoir des applications Web. Google pousse fortement dans ce sens, avec notamment le projet Polymer[2], et Angular n'est pas en reste avec sa philosophie très orientée composants réutilisables à travers les directives. Une très grande partie du framework se base sur les directives, parfois même sans que l'on s'en rende compte : une balise `<a href="">` dans votre HTML est une directive, un `<input>` en est une autre…

C'est, à l'heure actuelle, la partie la plus puissante mais la plus complexe de tout le framework. Heureusement la documentation officielle s'améliore rapidement (après avoir été pendant longtemps très… spartiate). Alors, faisons un tour de ce que nous proposent les directives !

Angular propose de nombreuses directives intégrées dans le framework. On peut distinguer différents types de directives :

- les directives du framework
- les directives de templating (dont des tags HTML enrichis)
- les directives de style
- les directives de binding
- les directives d'action

[1] http://www.w3.org/TR/2013/WD-components-intro-20130606/
[2] http://www.polymer-project.org/

9.1. Directives du framework

9.1.1. ngApp

```
<html>
  <head>
    <script src="angular.min.js"></script>
  </head>
  <body ng-app></body>
</html>
```

Il ne peut y avoir qu'un seul ngApp par HTML, cette directive indique la racine de l'application. Les applications ne peuvent pas s'imbriquer.

Plusieurs syntaxes sont possibles pour l'utiliser dans votre HTML : ng-app, x-ng-app, data-ng-app…, nous reviendrons sur le pourquoi plus tard. En attendant, nous utiliserons ici la notation de la documentation, ng-app.

Le premier ng-app trouvé par Angular sera 'bootstrappé' : pour utiliser plusieurs applications Angular dans une même page, il faut donc manuellement 'bootstrapper' les autres avec angular.bootstrap(). Cette fonction permet de démarrer manuellement une application Angular.

La fonction prend plusieurs arguments :

- l'élément DOM racine de l'application (équivalent à la balise où l'on aurait appliqué 'ngApp').
- une liste éventuelle de modules.

Cette fonction renvoie l'injecteur créé pour l'application.

```
<html>
  <head>
    <script src="angular.js"></script>
  </head>
  <body>{{ 2 + 2 }}</body>
  <script>
    angular.bootstrap(angular.element(document).find('body'));
  </script>
</html>
```

La fonction bootstrap() demandant un élément du DOM, il est nécessaire de sélectionner l'élément 'body' à l'aide de angular.element().

Cet élément, ou celui sur lequel est appliqué la directive ngApp, est nommé $rootElement par Angular.

9.1.2. ngStrictDI

AngularJS offre un système d'injection de dépendances qui permet d'injecter les services voulus dans un composant :

```
app.controller('MyCtrl', function($scope, $http) {
  // $scope and $http are injected by AngularJS
});
```

C'est très pratique, mais cela a un "léger" problème si vous minifiez votre application comme nous l'avons vu : `$scope` va s'appeler `a`, `$http` va s'appeler `b`, et là c'est le drame, l'injection ne fonctionne plus car Angular se base sur les noms.

Pour éviter ça une autre syntaxe est disponible :

```
app.controller('MyCtrl', ["$scope", "$http", function($scope, $http) {
  // $scope and $http are injected by AngularJS
  // and now handles minification
}]);
```

Et là, plus de problème. Il faut donc penser à utiliser cette syntaxe (ou s'appuyer sur un plugin dans votre build qui fasse tout ça automatiquement comme ng-min[3] ou ng-annotate[4], ou encore une autre syntaxe, basée sur l'attribut `$inject`). Mais il arrive qu'un développeur oublie de l'utiliser à un seul endroit et l'application ne démarre pas, ce qui est bien sûr un peu pénible.

C'est là ou la version 1.3 apporte une nouveauté : il est désormais possible de démarrer Angular en mode 'strict-di' (injection de dépendance stricte), et un beau message d'erreur pour indiquer que tel composant n'utilise pas la bonne syntaxe apparaît.

```
<body ng-app="app" ng-strict-di>...</body>

app.controller('MyBadCtrl', function($scope, $http) {
```

[3] https://github.com/btford/ngmin
[4] https://github.com/olov/ng-annotate

```
  // not allowed with strict DI
});
// --> 'MyBadCtrl is not using explicit annotation and cannot be invoked
 in strict mode'
```

9.1.3. ngInit

Cette directive permet d'évaluer une expression dans le scope courant et donc d'initialiser une variable dans le scope, sans passer par le controller.

```
    <div ng-init="world = 'world'">Hello {{ world }}!</div>
```

Ce n'est cependant pas une pratique recommandée, à l'exception de cas très particuliers où il est impossible de faire autrement, par exemple lors de l'imbrication de deux `ngRepeat`, où l'on veut conserver des variables très particulières créées par celui-ci, comme `$index`.

```
    <div ng-repeat="innerList in list" ng-init="outerIndex = $index">
      <div ng-repeat="value in innerList" ng-init="innerIndex = $index">
         <span class="example-init">list[ {{outerIndex}} ]
[ {{innerIndex}} ] = {{value}};</span>
      </div>
    </div>
```

9.1.4. ngController

Déjà vue, cette directive permet de déclarer un controller associé à un élément, voir le paragraphe dédié pour plus d'informations. Si vous utilisez un routeur, cette directive n'est alors que peu ou pas du tout utilisée.

9.2. Directives de template

Les directives les moins connues lorsque l'on débute sont certainement les tags HTML enrichis car ils sont relativement transparents à utiliser pour le développeur. D'autres directives permettent également au développeur de se simplifier l'écriture de template.

9.2.1. a

Par exemple, la directive `a` va empêcher la navigation si le `href` est vide. Les développeurs ajoutent alors généralement un `ngClick` pour définir l'action à effectuer :

```
<a href="" ng-click="click()">href vide {{ clicked }}</a>

$scope.click = function(){
  $scope.clicked = "cliqué";
}
```

A noter : la façon correcte de variabiliser les URLs consiste à utiliser `ngHref` plutôt que `href`. Cela évite que les URLs soient invalides tant qu'elles ne sont pas compilées, et qu'un utilisateur, plus rapide que votre navigateur, ne clique sur le lien avant que la variable n'ait été remplacée.

```
<a ng-href="/products/{{productId}}">Detail</a>
```

De la même façon, pour variabiliser les URLs de vos images, il existe `ngSrc` et `ngSrcset` pour définir la source :

```
<img ng-src="http://www.gravatar.com/avatar/{{id}}.png"/>
<img ng-srcset="http://www.gravatar.com/avatar/{{id}}.png 2x"/>
```

C'est ici nécessaire d'utiliser ngSrc et non pas src. En effet, le navigateur cherchera à charger l'image immédiatement s'il trouve un attribut src, et tombera donc sur une erreur 404 tant que Angular n'a pas compilé l'expression contenant la variable. Avec ngSrc, on est tranquille : Angular ajoutera pour nous l'attribut src avec la bonne URL une fois qu'elle sera compilée.

9.2.2. Champs et formulaires

La directive `input` permet d'ajouter de la validation HTML5 à vos champs, même pour les navigateurs qui ne le supportent pas. Au minimum, il est nécessaire de lier l'input à un model (`ngModel`). Vous pouvez également demander à ce que le champ soit obligatoire avec `required`.

```
<input type="text" ng-model="color" required="true"/>
```

Attardons nous un peu sur ngModel.

9.2.3. ngModel

Lorsque vous ajoutez cette directive à un champ, vous indiquez à AngularJS que chaque modification de l'utilisateur entraînera la mise à jour du modèle. Cela veut dire

que, pour nous, AngularJS se charge d'ajouter les listeners nécessaires sur le champ, et de mettre à jour le modèle à chaque événement.

Tout ce travail est fait par un controller lié à la directive : NgModelController. Il est de plus chargé de gérer la validation du champ, qui dépend du type du champ et des directives de validation ajoutée au champ. Par exemple, Angular validera qu'un champ de type 'number' contient effectivement un nombre, et on pourra spécifier une valeur minimale via un attribut supplémentaire.

Ce controller expose un ensemble de méthode et de propriétés :

- `$pristine` : permet de savoir si l'utilisateur a déjà touché au champ. On pourrait traduire en français par vierge. Ce booléen sera donc vrai tant qu'il n'y aura pas eu d'interaction, puis faux ensuite.
- `$dirty` : à l'inverse de `$pristine`, `$dirty` sera vrai si l'utilisateur a commencé à interagir avec le champ. Pour résumer : $dirty == !$pristine.
- `$valid` : sera vrai si le champ est valide.
- `$invalid` : sera vrai si le champ est invalide.
- `$error` : représente un dictionnaire des erreurs, avec comme clé le nom de l'erreur, par exemple `required`.
- `$untouched` : vrai si le champ n'a pas encore perdu le focus.
- `$touched` : vrai si le champ a perdu le focus (même sans avoir changé le modèle).

Là où Angular est fabuleux, c'est que cette validation automatique va non seulement mettre à jour les propriétés ci-dessus, mais aussi mettre à jour les classes CSS portées par le champ.

Voyons en détail ce fonctionnement dans la partie consacrée aux formulaires ci-dessous.

9.2.4. form

Toute application web contient son lot de formulaires. Un formulaire est un ensemble de champs, chaque champ étant une manière pour l'utilisateur de saisir des informations. Ces champs groupés ensemble dans un formulaire ont un sens, que ce soit une page d'enregistrement pour vos utilisateurs, ou une formulaire de login.

Les formulaires en Angular étendent les formulaires HTML en leur ajoutant un certain nombre d'états et en donnant aux développeurs de nouvelles façon d'agir.

Les formulaires, comme beaucoup de composants en Angular, sont des directives. Chaque fois que vous utilisez un formulaire en Angular, la directive `form` va instancier un controller nommé `FormController`. Il est possible d'accéder au controller dans votre code en utilisant comme nom l'attribut `name` de votre formulaire. Celui-ci connaît à son tour les controllers de chacun des champs (les fameux NgModelController vus précédemment).

Ce controller expose un ensemble de méthode et de propriétés :

- `$pristine` : permet de savoir si l'utilisateur a déjà touché au formulaire. On pourrait traduire en français par vierge. Ce booléen sera donc vrai tant qu'il n'y aura pas eu d'interaction, puis faux ensuite.
- `$dirty` : à l'inverse de `$pristine`, `$dirty` sera vrai si l'utilisateur a commencé à interagir avec le formulaire. Pour résumer : $dirty == !$pristine.
- `$valid` : sera vrai si l'ensemble des champs (et éventuellement des formulaires imbriqués) sont valides.
- `$invalid` : sera vrai si au moins un champ (ou formulaire imbriqué) est invalide.
- `$error` : représente un dictionnaire des erreurs, avec comme clé le nom de l'erreur, par exemple `required`, et comme valeur la liste des champs avec cette erreur, par exemple login et password.
- `$submitted` : vrai si le formulaire a été soumis.

Ces états sont disponibles sur le formulaire global mais également sur chaque champ du formulaire. Si votre formulaire se nomme `loginForm`, et contient un champ `password` alors vous pouvez voir si ce champ est valide grâce au code suivant :

```
loginForm.password.$valid
```

Vous vous voyez déjà ajouter du code JS pour surveiller ces états et changer le CSS en cas d'erreur. Pas besoin ! Angular ajoute ces états comme classe CSS directement sur chaque champ et le formulaire. Ainsi lorsque le formulaire encore vierge s'affiche, il contient déjà la classe CSS `ng-pristine` :

```
<form name='loginForm' class='ng-pristine'>
  <input name='login' ng-model='user.login' class='ng-pristine'/>
  <input name='password' ng-model='user.password' class='ng-pristine'/
>
```

```
    </form>
```

Dès la saisie du premier caractère dans le champ `input`, le formulaire devient dirty :

```
    <form name='loginForm' class='ng-dirty'>
      <input name='login' ng-model='user.login' class='ng-dirty'/>
      <input name='password' ng-model='user.password' class='ng-pristine'/
>
    </form>
```

A noter que le formulaire, ainsi que l'input, reste dirty une fois modifié, même si la modification est annulée.

Si l'un des champs présente une contrainte (comme par exemple `required` ou `email`), alors le formulaire sera invalide tant que cette contrainte ne sera pas satisfaite. Le champ comme le formulaire aura donc la classe `ng-invalid` ainsi que le champ en question. Une fois la condition satisfaite, la classe `ng-valid` remplace la classe `ng-invalid`.

Vous pouvez donc très facilement customiser votre CSS pour ajouter un style selon l'état de votre formulaire. Cela peut être un bord rouge en cas de champ invalide :

```
    input.ng-dirty.ng-invalid {
      border: 1px solid red;
    }
```

Ou encore afficher un message d'erreur dans votre HTML :

```
    <form name='loginForm'>
      <input name='login' type='email' ng-model='user.login'/>
      <span ng-show='loginForm.login.$invalid'>Your login should be a
 valid email</span>
      <input name='password' ng-model='user.password'/>
    </form>
```

Ajouter un type `email` sur un champ place une contrainte sur celui-ci, obligeant l'utilisateur à entrer un login sous la forme d'une adresse email valide. Dès que l'utilisateur commencera sa saisie de login, le champ deviendra invalide, rendant l'expression `loginForm.login.$invalid` vraie. La directive `ng-show` activera alors l'affichage du message d'avertissement. Dès que le login saisi sera un email

valide, l'expression deviendra fausse et l'avertissement sera caché. Plutôt pas mal pour une ligne de HTML non ?

Vous pouvez bien sûr cumuler les conditions d'affichage de l'avertissement, ou faire un avertissement par type d'erreur :

```
    <form name='loginForm'>
      <input name='login' required type='email' ng-model='user.login'/>
      <span ng-show='loginForm.login.$dirty && loginForm.login.
$error.required'>A login is required</span>
      <span ng-show='loginForm.login.$error.email'>Your login should be a
 valid email</span>
      <input name='password' ng-model='user.password'/>
    </form>
```

Ainsi si l'utilisateur, après avoir entré un login (rendant ainsi le champ "dirty"), l'efface, un message d'avertissement apparaîtra pour indiquer que le login est nécessaire. Les combinaisons ne sont limitées que par votre imagination !

Plusieurs méthodes sont disponibles sur le FormController :

- `addControl()`, `removeControl()` permettent d'ajouter ou de supprimer des champs du formulaire. Par défaut tous les inputs "classiques" (input, select, textarea) que vous utilisez avec un `ng-model` sont ajoutés au formulaire pour vous. Ils ont tous un controller nommé NgModelController, qui gère justement les états ($pristine, $valid, etc…), la validation, l'ajout au formulaire pour vous ainsi que le binding des vues au modèle. Les méthodes `addControl()` et `removeControl()` peuvent être intéressantes si vous voulez ajouter un autre type de champ à votre formulaire.
- `setDirty()`, `setPristine()` vont respectivement mettre le formulaire dans un état `dirty` ou `pristine` avec les classes CSS associées positionnées. A noter que les méthodes se propagent vers les formulaires parents si ils existent. En revanche, seule la méthode `setPristine()` s'applique sur chaque champ du formulaire. Vous pouvez donc faire un 'reset' du formulaire et de ses champs grâce à `$setPristine()`.
- `setValidity(errorType, isValid, control)` va, comme son nom l'indique, passer un champ de votre formulaire comme $valid ou $invalid pour le type d'erreur précisé, affectant par le même coup la validité du formulaire ainsi que celles des éventuels formulaires parents.

Ces méthodes sont aussi accessibles sur l'objet `NgModelController` de chaque champ et sont particulièrement pratiques pour créer des validations 'custom', non fournies par le framework et spécifiques à votre domaine fonctionnel par exemple. Nous verrons plus tard en détail comment faire pour créer une directive ajoutant cette contrainte de validation, et implementer la validation de cette contrainte.

La dernière chose à retenir sur les formulaires concerne la soumission. Angular étant conçu pour les "single page applications", on cherche à éviter les rechargements non nécessaires. Ainsi l'action par défaut du formulaire sera désactivée, à moins que vous le vouliez explicitement en précisant une `action` dans la balise `form`. Angular cherchera plutôt à vous faire gérer vous-même la soumission du formulaire, soit en utilisant la directive `ngSubmit` dans la balise `form` soit en utilisant la directive `ngClick` sur un `button` ou un `input` de type `submit` (mais pas les deux à la fois, sinon vous avez une double soumission !).

Vous avez maintenant toutes les clés pour faire une bonne validation côté client de vos formulaires en Angular. Mais que cela ne vous empêche de vérifier côté serveur ! ;-)

Avant de passer à la suite, voyons en détail quelles sont les directives de validation sur les différents types de champs.

9.2.5. input

La directive `input` possède des attributs généraux qui permettent de valider la longueur de la valeur saisie (`ngMinlength` et `ngMaxlength`). Le controller ajoutera les classes CSS correspondantes en cas d'erreur (`ng-invalid-minlength`, `ng-invalid-maxlength`) et de validité (`ng-valid-minlength`, `ng-valid-maxlength`).

```
input.ng-dirty.ng-invalid-minlength { border: 6px solid red; }
input.ng-dirty.ng-invalid-maxlength { border: 6px solid orange; }
input.ng-dirty.ng-valid { border: 6px solid green; }

<input type="text" name="firstname" ng-model="firstname"
ng-minlength="3" ng-maxlength="10" placeholder="3 <= word <= 10">
```

Il est possible de valider un pattern, par exemple pour s'assurer que tous les caractères sont en minuscules :

```
input.ng-dirty.ng-invalid-pattern { border: 6px solid red; }
input.ng-dirty.ng-valid { border: 6px solid green; }
```

```
    <input type="text" name="lowercase" ng-model="lowercase" ng-pattern="/
^[a-z]+$/" placeholder="lowercase only">
```

L'attribut `ng-change` quant à lui permet d'appeler une fonction à chaque changement fait par l'utilisateur. En revanche, un changement programmatique du modèle n'appelle pas la fonction; on parle bien seulement des changements utilisateurs.

```
    $scope.change = function(){
      $scope.counter++;
    }

    <input type="text" name="changed" ng-model="changed" ng-
change="change()" placeholder="type to change">
    <div>Counter {{ counter }}</div>
```

input text

Dernier attribut utilisable sur un `input` de type `text` seulement : `ng-trim`. Par défaut Angular supprime les espaces supplémentaires à la fin des valeurs saisies (et ne déclenche pas la fonction `ng-change` si des espaces sont ajoutés). Si vous voulez conserver ces espaces, il suffit de passer la valeur `false` à `ng-trim`.

```
    $scope.change = function(){
      $scope.counter++;
    }

    <input type="text" name="trimmed" ng-model="trimmed" ng-
change="change()" placeholder="type to change">
    <div>Length {{ trimmed.length }}</div>
    <div>Counter {{ counter }}</div>
    <div>Make space count</div>
    <input type="text" name="trimmed" ng-model="trimmed" ng-
change="change()" ng-trim="false" placeholder="type to change">
```

Enfin, petite directive pratique en conjonction avec un `input`, `ngList` va convertir une liste de chaînes de caractères séparée par des virgules en un tableau de chaînes de caractères.

```
    <input type="text" name="names" ng-model="names" ng-
list placeholder="name, name, name">
```

```
    <div>{{ names }}</div> <!-- Cédric, JB -> ["Cédric", "JB"] -->
```

Vous pouvez également préciser le séparateur qui vous intéresse :

```
    <input type="text" name="names" ng-model="names" ng-
list=";" placeholder="name; name; name">
    <div>{{ names }}</div> <!-- Cédric; JB -> ["Cédric"; "JB"] -->
```

Voilà pour le type `text` !

input number

Mais la spécification HTML5 permet également de nouveaux types comme par exemple `number`. Angular permet bien sûr de l'utiliser, ainsi que les attributs `min` et `max` associés et nous donne les classes CSS d'erreur et de validité associées.

```
    input.ng-dirty.ng-invalid.ng-invalid-min { border: 6px solid red; }
    input.ng-dirty.ng-invalid.ng-invalid-max { border: 6px solid orange; }
    input.ng-dirty.ng-valid { border: 6px solid green; }

    <input type="number" name="age" ng-
model="age" placeholder="012" min="18" max="130">
```

input email

Idem pour le type `email` : pas d'attributs spécifiques disponibles mais les classes CSS d'erreurs et de validité sont ajoutées. Pas besoin d'écrire vous la regex incompréhensible de validation des emails !

```
    input.ng-dirty.ng-invalid.ng-invalid-email { border: 6px solid red; }
    input.ng-dirty.ng-valid { border: 6px solid green; }

    <input type="email" name="email" ng-
model="email" placeholder="mail@mail.com">
```

input url

Et enfin, même chose pour le type `url`.

```
input.ng-dirty.ng-invalid.ng-invalid-url { border: 6px solid red; }
input.ng-dirty.ng-valid { border: 6px solid green; }
```

```
<input type="url" name="url" ng-model="url" placeholder="url">
```

input date

La version 1.3 apporte la gestion du type 'date', en utilisant le support HTML5 si disponible (et vous aurez alors le plus ou moins beau date-picker de votre navigateur), ou un champ texte sinon, dans lequel il faudra entrer une date au format ISO-8601, par exemple 'yyyy-MM-dd'. Le modèle lié doit être un object JS de type Date.

```
<form name="userForm">
  <input name="birthDate" type="date" ng-required="true" ng-
model="user.birthDate"
    min="1900-01-01" max="2014-01-01">
  <span ng-show="userForm.birthDate.$error.date">Date is incorrect</span>
  <span ng-show="userForm.birthDate.$error.min">Date should be after
 1900</span>
  <span ng-show="userForm.birthDate.$error.max">You should be born before
 2014</span>
</form>
```

S'il y a les dates, il y a également les heures. Si ce type n'est pas supporté par le navigateur, un champ texte sera utilisé et le format sera HH:mm :

```
<form name="eventForm">
  <input name="startTime" type="time" ng-required="true" ng-
model="event.startTime"
    min="06:00" max="18:00">
  <span ng-show="eventForm.startTime.$error.time">Time is incorrect</span>
  <span ng-show="eventForm.startTime.$error.min">Event should be after
 6AM</span>
  <span ng-show="eventForm.startTime.$error.max">Event should be before
 6PM</span>
</form>
```

Le modèle lié est une Date JS avec la date du 1 Janvier 1900 et l'heure saisie.

La version 1.3 supporte également les champs 'dateTimeLocal', qui sont donc une date et une heure. Si ce type n'est pas supporté par le navigateur, un champ texte sera utilisé et le format ISO sera yyyy-MM-ddTHH:mm :

```
<form name="eventForm">
```

```
  <input name="startDate" type="dateTimeLocal" ng-required="true" ng-
model="event.startDate"
    min="2014-06-01T00:00" max="2014-06-30T23:59">
  <span ng-show="eventForm.startDate.$error.dateTimeLocal">Date is
 incorrect</span>
  <span ng-show="eventForm.startDate.$error.min">Event should be after
 May</span>
  <span ng-show="eventForm.startDate.$error.max">Event should be before
 July</span>
</form>
```

Enfin, il est possible d'utiliser les champs 'month' ou 'week', stockés également dans un modèle de type Date, qui permettent évidemment de choisir un mois ou une semaine. Si ce type n'est pas supporté par le navigateur, un champ texte sera utilisé et le format ISO à utiliser sera yyyy- *ou yyyy-W* pour les semaines :

```
<form name="eventForm">
  <input name="week" type="week" ng-required="true" ng-
model="event.weekNumber"
    min="2014-W01" max="2014-W52">
  <span ng-show="eventForm.week.$error.week">Week is incorrect</span>
  <span ng-show="eventForm.week.$error.min">Event should be after 2013</
span>
  <span ng-show="eventForm.week.$error.max">Event should be before 2015</
span>
</form>
```

input checkbox

Les checkbox bénéficient également d'un traitement particulier, avec la possibilité de donner une valeur différente au modèle associé si la valeur est vraie ou fausse. Par exemple :

```
    <label>Boolean</label><input type="checkbox" name="checkbox" ng-
model="checkbox"><br/>
    <label>Gender</label><input type="checkbox" name="checkbox" ng-
model="checkbox" ng-true-value="Male" ng-false-value="Female">
    <div>{{ checkbox }}</div>
```

Dans le premier cas, le modèle 'checkbox' sera une valeur booléenne. Dans le second cas, l'utilisation des attributs `ng-true-value` et `ng-false-value` permet d'attribuer les valeurs 'male' ou 'female' au modèle.

input radio

Les radio buttons n'ont quant à eux pas d'attribut spécifique hormis ngValue, qui permet de déterminer la valeur que doit prendre le modèle quand le bouton radio est coché:

```
    <label ng-repeat="role in roles">
      <input type="radio" ng-model="user.roleId" ng-
value="role.id" name="role">
      {{role.name}}
    </label>
```

Les inputs ne sont pas les seuls champs disponibles et bénéficiant d'une directive : c'est aussi le cas des selects et des textareas.

9.2.6. select

Les selects permettent d'itérer sur un tableau d'élément ou sur les clés/valeurs d'un objet (syntaxe '.. for .. in' du ngOptions). Il est possible de grouper ses éléments dans le select (syntaxe '.. group by .. for .. in') et même d'affecter seulement une partie de l'objet au modèle associé (syntaxe '.. as .. for .. in').

```
    <select ng-model="member" ng-options="people.name for people in
team"></select>
    <select ng-model="member" ng-options="people.name group by
people.gender for people in team"></select>
    <select ng-model="member" ng-options="people.name as people.name group
by people.gender for people in team"></select>
    <div>{{ member }}</div>

    $scope.team = [
      {name: 'Agnes', gender: 'female'},
      {name: 'Cédric', gender: 'male'},
      {name: 'Cyril', gender: 'male'},
      {name: 'JB', gender: 'male'}
    ];
```

Le premier select affiche la liste des prénoms des ninjas, et une sélection affectera le ninja en question au modèle 'member'. Le second, la même liste, groupée par genre. Le troisième affiche la même chose que le second, mais la sélection d'un ninja entraînera l'affectation de son prénom à 'member', plutôt que l'objet complet.

9.2.7. textarea

Enfin les textareas permettent de contrôler la longueur de la valeur saisie :

```
    textarea.ng-dirty.ng-invalid-minlength { border: 6px solid red; }
    textarea.ng-dirty.ng-invalid-maxlength { border: 6px solid orange; }
    textarea.ng-dirty.ng-valid { border: 6px solid green; }

    <textarea ng-model="bio" ng-minlength="10" ng-
maxlength="20" placeholder="10 <= word <= 20"></textarea>
```

9.2.8. ngMessages

Un nouveau module est apparu en version 1.3, c'est le module `ngMessages`.

Comme je l'expliquais plus haut, il est possible en Angular d'afficher des messages d'erreur sur vos formulaires en fonction de différents critères : si le champ est vierge ou si l'utilisateur l'a touché, si le champ est obligatoire, si le champ viole une contrainte particulière… Bref vous pouvez afficher le message que vous voulez !

Le reproche fait à cette version est que la condition booléenne à écrire pour afficher le message est souvent verbeuse du type :

```
<span ng-if="eventForm.eventName.$dirty && eventForm.eventName.
$error.required">The event name is required</span>
<span ng-if="eventForm.eventName.$dirty && eventForm.eventName.
$error.minlength">The event name is not long enough</span>
<span ng-if="eventForm.eventName.$dirty && eventForm.eventName.
$error.maxlength">The event name is too long</span>
<span ng-if="eventForm.eventName.$dirty && eventForm.eventName.
$error.pattern">The event name
must be in lowercase</span>
```

Ce n'est pas insurmontable mais on a souvent plusieurs messages par champ, et plusieurs champs par formulaire : on se retrouve avec un formulaire HTML bien rempli ! On peut aussi vouloir n'afficher qu'un seul message à la fois, par ordre de priorité.

C'est ici qu'entre en jeu ce module, avec lequel l'exemple précédent pourra alors s'écrire :

```
<div ng-if="eventForm.eventName.$dirty" ng-messages="eventForm.eventName.
$error">
```

```
  <div ng-message="required">The event name is required</div>
  <div ng-message="minlength">The event name is too short</div>
  <div ng-message="maxlength">The event name is too long</div>
  <div ng-message="pattern">The event name should be in lowercase</div>
</div>
```

C'est un peu plus lisible et cela gère pour nous le fait de n'afficher qu'un seul message et de les afficher par ordre de priorité. Si vous voulez afficher plusieurs messages à la fois, c'est facile, vous ajoutez la directive `ng-messages-multiple`.

Il est possible d'écrire tout ça différemment :

```
<ng-messages for="eventForm.eventName.$dirty" multiple>
  <ng-message when="required">...</ng-message>
  <ng-message when="minlength">...</ng-message>
  <ng-message when="maxlength">...</ng-message>
  <ng-message when="pattern">...</ng-message>
</ng-messages>
```

Le dernier avantage à utiliser ngMessages réside dans la possibilité d'externaliser les messages d'erreur dans des templates.

```
<!-- error-messages.html -->
<ng-message when="required">...</ng-message>
<ng-message when="minlength">...</ng-message>
<ng-message when="maxlength">...</ng-message>
<ng-message when="pattern">...</ng-message>
```

Puis dans votre formulaire :

```
<ng-messages for="eventForm.eventName.$dirty" ng-include-messages="error-
messages.html"/>
```

Et si jamais les messages d'erreur sont trop génériques, vous pouvez les surcharger directement dans votre formulaire :

```
<ng-messages for="eventForm.eventName.$dirty" ng-include-messages="error-
messages.html">
  <ng-message when="required">An overloaded message</ng-message>
</ng-messages>
```

A noter que cela fonctionne également avec vos directives de validation custom !

Là encore, amusez vous avec le Plunker associé[5].

9.2.9. script

Dernier tag amélioré par Angular : `script`. Angular reconnaît un type particulier `text/ng-template`. Le template peut alors être utilisé par `ngInclude`, `ngView` ou une directive.

Voyons maintenant les autres directives de template, celles qui ne sont pas des tags HTML enrichis.

9.2.10. ngIf

A la différence de `ngHide`, `ngIf` va réellement agir sur le DOM et pas simplement cacher l'élément : si l'expression renvoie faux, l'élément sera supprimé du DOM, ou recréé si l'expression est vraie. A noter que l'élément est recréé dans son état originel et pas avec les éventuelles classes qui auraient pu lui être ajoutées depuis.

```
<input type="checkbox" ng-model="isVisible"/><br/>
<div ng-if="isVisible">I'm gonna disappear</div>
```

9.2.11. ngRepeat

`ngRepeat` est l'une des directives les plus utilisées puisqu'elle permet de répéter un élément HTML en parcourant un tableau d'objet :

```
$scope.team = [
  {name: 'Agnes', gender: 'female'},
  {name: 'Cédric', gender: 'male'},
  {name: 'Cyril', gender: 'male'},
  {name: 'JB', gender: 'male'}
];
<div ng-repeat="member in team">{{ member.name }}</div>
```

Ou également un objet, en parcourant ses clés et valeurs :

```
$scope.ninja = {name: 'Agnes', gender: 'female'};
```

[5] http://plnkr.co/edit/jUkOtx30Etb1IbscxiJh?p=preview

```
<div ng-repeat="(key, value) in ninja">{{ key }} : {{ value }}</div>
```

Chaque élément aura son propre scope, de façon transparente pour le développeur. Quelques propriétés bien pratiques sont ajoutées sur ce scope :

- `$index` pour la position de l'itérateur
- `$first` vrai pour le premier élément, faux ensuite
- `$middle` vrai du second à l'avant dernier élément
- `$last` vrai pour le dernier élément
- `$even` / `$odd` si l'élément est pair ou impair

```
<input type="text" ng-model="teamFilter"/>
<div ng-repeat="member in team | filter:teamFilter">{{ $index }} -
{{ member.name }}</div>
```

Petite subtilité : `ngRepeat` ne permet pas d'avoir des éléments dupliqués dans les tableaux. En effet, pour suivre le lien entre un objet du modèle et l'élément DOM qui lui est associé, Angular calcule un hash (grâce à la fonction `$id`) qu'il ajoute à l'objet dans la propriété `$$hashKey`. Mais si le tableau contient des nombres par exemple, vous risquez de vous retrouver confronté à l'erreur `ngRepeat:dupes`. Il faut en effet indiquer à Angular comment suivre les éléments grâce à 'track by'.

```
$scope.numbers = [1,2,1];
<div ng-repeat="number in numbers track by $index">{{ number }}</div>
```

Vous savez qu'il est possible de filtrer la collection que vous allez afficher dans un `ngRepeat`, par exemple :

```
<div ng-repeat="poney in poneys | filter:search">{{ poney }}</div>
```

C'est très pratique, mais si l'on veut utiliser le nombre de résultats affichés, on est obligé de réappliquer le même filtre à la même collection :

```
<div ng-repeat="poney in poneys | filter:search">{{ poney }}</div>
<div ng-if="(poneys | filter:search).length === 0">No results.</div>
```

La version 1.3 introduit un alias `as`, qui permet justement de stocker le résultat du filtre dans le `ngRepeat` et de le réutiliser plus tard.

```
    <div ng-repeat="poney in poneys | filter:search as
  poneysFiltered">{{ poney }}</div>
    <div ng-if="poneysFiltered.length === 0">No results.</div>
```

9.2.12. ngSwitch

Cette directive permet de remplacer un élément selon la valeur d'une expression. Par exemple on peut passer une `div` en input si l'utilisateur veut passer en édition :

```
  <div ng-switch on="isEditable">
    <label>Editable</label><input type="checkbox" ng-model="isEditable"/>
    <div ng-switch-when="false">Name : {{ member.name }}</div>
    <div ng-switch-when="true"><input type="text" ng-model="member.name"/
  ></div>
  </div>
```

9.2.13. ngInclude

Cette directive permet d'inclure un template (après l'avoir compilé) dans un élément du DOM. L'attribut `onload` permet d'exécuter une expression après chargement.

```
    <div ng-include="template.url"></div>
```

9.3. Directives de binding

Angular offre quelques directives pour lier le modèle à la vue.

9.3.1. ngBind

Cette directive est équivalente à l'utilisation d'une expression classique, à ceci près qu'elle garantit l'affichage de la valeur sans l'effet de 'flickering', c'est-à-dire l'affichage du template Angular non compilé au chargement de la page.

```
    <!-- peut éventuellement 'scintiller' si la compilation Angular est
  lente -->
    Hello {{ world }}<br/>
    <!-- ne scintillera pas -->
    Hello <span ng-bind="world"></span>
```

9.3.2. ngBindHtml

Angular propose également une directive pour inclure directement du HTML dans un élément. `ngBindHtml` prendra le résultat de l'expression pour le mettre à l'intérieur de l'élément :

```
$scope.helloHtml = '<strong>Hello</strong>'
<div ng-bind-html="helloHtml"></div>
```

9.3.3. ngBindTemplate

`ngBind` ne permet pas de recevoir plusieurs valeurs, si le besoin se présente, il est possible d'utiliser `ngBindTemplate` :

```
{{ hello }} {{ world }}<br/>
<span ng-bind-template="{{ hello }} {{ world }}"></span>
```

9.3.4. ngNonBindable

`ngNonBindable` indique à Angular d'ignorer le contenu de l'élément (cela permet d'afficher des '{{ }}' par exemple) :

```
<div ng-non-bindable>Hello {{ world }}</div>
```

9.4. Directives de style

Une fois le template écrit, il faut souvent le rendre un peu plus esthétique, lui appliquer un style, cacher des éléments : c'est le rôle de cet ensemble de directives.

9.4.1. ngCloak

Une alternative possible à `ngBind` pour éviter le 'flickering' consiste à utiliser une expression normale et à lui adjoindre la directive `ngCloak`. Le template ne s'affichera que lorsqu'il aura été compilé. Cette directive ajoute simplement une classe CSS `.ng-cloak { display: none !important; }` puis la supprime lorsque le template est compilé. A noter que c'est l'une des très rares règles CSS ajoutées par Angular.

```
<div ng-cloak>Hello {{ world }}</div>
```

9.4.2. ngHide/ngShow

Restons dans les classes CSS ajoutées par Angular avec `ngHide` qui permet de cacher un élément selon une condition (si l'expression est vraie, l'élément est caché). Son pendant est `ngShow` qui est exactement l'inverse (l'élément sera affiché si l'expression est vraie).

```
<input type="checkbox" ng-model="hidden"/><br/>
<div ng-hide="hidden">Hello hide</div>
<div ng-show="hidden">Hello show</div>
```

9.4.3. ngClass

Cette directive permet d'ajouter dynamiquement toutes les classes renvoyées par l'expression, que ce soit une chaîne de caractères avec les classes CSS séparées par des virgules, un tableau de classes CSS ou, plus intéressant, une map avec comme clés les classes et comme valeur un booléen qui permet de savoir si elle doit être appliquée ou non.

```
.green { color: green;}
.bold { font-weight: bold;}
<input type="checkbox" ng-model="greenChecked"/>
<input type="checkbox" ng-model="boldChecked"/>
<div ng-class="{green: greenChecked, bold: boldChecked }">I'm classy</
div>
```

9.4.4. ngClassEven/ngClassOdd

Même chose que précédemment, mais utilisé en conjonction avec `ngRepeat`, ce qui permet de ne les appliquer que sur les lignes paires ou impaires.

```
<div ng-repeat="row in rows" ng-class-odd="{bold: isBold}" ng-class-
even="{green: isGreen}">I'm classy {{ row }}</div>
```

9.4.5. ngDisabled

Cette directive va désactiver l'élément si l'expression est vraie. Pratique par exemple pour désactiver la soumission d'un formulaire tant que certains champs sont incorrects.

```
<input type="checkbox" ng-model="isDisabled"/><br/>
```

```
<button ng-disabled="isDisabled">Button</button>
```

9.4.6. ngReadonly

Il est possible de passer dynamiquement un élément en lecture seule grâce à `ngReadonly`.

```
<input type="checkbox" ng-model="isReadonly"><br/>
<input type="text" ng-readonly="isReadonly" value="Read only">
```

9.4.7. ngStyle

`ngStyle` permet d'ajouter dynamiquement un style à un élément sous la forme d'un dictionnaire clé/valeur :

```
$scope.style = { color: 'red' };
<div>{{ style }}</div>
<label>Style color </label><input type="text" ng-model="style.color"/>
<div ng-style="style">I've got some style</div>
```

9.5. Directives d'action

Cette dernière catégorie regroupe tout ce qui s'apparente à la gestion d'événement et que l'on ferait traditionnellement avec jQuery.

9.5.1. ngClick

Probablement la plus utilisée, `ngClick` permet d'appeler une fonction au clic de souris.

```
<div ng-click="click()">Click me!</div>
```

9.5.2. ngDblClick

La petite soeur de la précédente, elle, appelle la fonction au double clic.

```
<div ng-dbl-click="dblclick()">Double Click me!</div>
```

9.5.3. ngCut/ngCopy/ngPaste

Les fonctions associées à ces directives seront appelées lors d'une action couper/copier/coller, que ce soit au clavier ou à la souris.

```
    <input ng-cut="cut()" ng-copy="copy()" ng-paste="paste()" value="Cut/
Copy/Paste me!">
```

9.5.4. ngFocus/ngBlur

Ces directives permettent de jouer avec les événements de prise de focus (`ngFocus`) et de perte de focus (`ngBlur`).

```
    <div ng-focus="focus()" ng-blur="blur()">Focus me!"</div>
```

9.5.5. ngKeydown/ngKeypress/ngKeyup

Ces trois directives sont liées aux événements clavier (keydown, keypress, keyup).

```
    <input ng-keydown="down()" ng-keyup="up()" ng-
keypress="press()" value="Keyboard!">
```

9.5.6. ngMousemove/ngMouseenter/ngMouseleave/ngMouseover/ngMousedown/ngMouseup

Enfin, ces six directives gèrent les différents événements liés à la souris (curseur entré, présent ou sorti de la zone, curseur déplacé, bouton appuyé ou relâché).

```
    <div ng-mouseenter="enter()" ng-mouseover="over()" ng-
mouseleave="leave" ng-mousedown="down()" ng-mouseup="up()">Mouse!"</div>
```

Et voilà pour les directives existantes ! Voyons maintenant comment créer nos propres directives.

9.6. Créer ses directives

Il faut d'ores et déjà noter qu'il existe 4 façons d'utiliser une directive dans vos templates HTML :

- comme un élément : si nous avons une directive `poney` (que serait JavaScript sans ses poneys ?) définie comme un élément, alors vous pouvez écrire `<poney></poney>` dans votre HTML et l'élément sera votre directive.
- comme un attribut : en reprenant notre exemple, il est possible de déclarer votre directive comme `<div poney></div>`.
- comme une classe CSS : votre directive peut être une classe CSS également : `<div class='poney'></div>`
- comme un commentaire : la dernière façon, la moins répandue, consiste à déclarer votre directive par un commentaire de la forme `<!-- directive: poney -->`

Il est généralement recommandé de préférer les déclarations par élément (si vous avez créé un composant à vous) ou attribut (si la directive n'est pas un composant mais plutôt un ajout de comportement).

Le compilateur HTML d'Angular va rechercher ces directives dans vos templates pour effectuer le remplacement de la déclaration par le composant que vous avez défini.

Il y a cependant une petite subtilité à connaître sur le parser HTML : il commence en effet par normaliser les attributs qu'il croise. La littérature Angular référence souvent les directives avec leur nom en camelCase (i.e. ngModel), mais HTML n'étant pas sensible à la casse, le parser cherche les noms en version "dash-limited" (i.e. ng-model). Le parser va même normaliser les noms des éléments et attributs en supprimant les éventuels 'x-' ou 'data-' qui précèdent (ces préfixes permettant d'avoir un HTML valide, les attributs Angular ne faisant, bien sûr, pas partie du standard HTML), et en convertissant les ':', '-' ou '_' en version camelCase. Donc, ng_model, ng:model, x-ng-model ou data-ng_model sont équivalents.

9.6.1. Template

Vous devez ensuite déclarer votre directive dans votre code JavaScript, pour indiquer quel template elle contient, ainsi que son comportement. Sa déclaration est assez simple : une méthode `directive` permet d'enregistrer votre composant et prend comme argument son nom (normalisé, donc en camelCase) et une méthode de création (factory method).

```
app.directive('poney', function() {
  return {
    template: '<h1>Poney</h1>'
  };
});
```

Voilà une directive très simple qui remplacera l'élément sur lequel l'attribut `poney` sera placé (comme rien n'est précisé ici, c'est la façon par attribut, qui est le défaut, qui sera utilisée[6]), par un titre contenant "poney".

Si votre template HTML est comme suit :

```
<div poney>This text will be removed</div>
```

Alors, comme le texte contenu dans la `div` le laisse deviner, une fois la directive appliquée, le texte sera remplacé par le template.

Les templates peuvent également inclure des expressions Angular : les expressions seront alors évaluées contre le scope englobant par défaut (nous reviendrons là-dessus plus tard).

```
app.directive('poney', function() {
  return {
    template: '<h1>{{color}} Poney</h1>'
  };
});
```

Dans cet exemple, Angular va chercher à résoudre la couleur dans le scope courant et remplacera l'expression par la valeur trouvée.

9.6.2. Template Url

Vous pouvez donc définir un template dans la directive. Il est également possible de passer une URL de template grâce à `templateUrl`.

```
app.directive('poney', function() {
  return {
    templateUrl: '/directives/poney.html'
  };
});
```

Lorsque l'on utilise un template chargé par URL, celui-ci sera téléchargé de façon asynchrone, compilé et mis en cache dans le service Angular `$templateCache` pour des utilisations ultérieures. Vous pouvez donner directement le lien vers le template, ou le wrapper dans une balise `script` et donner son id :

[6] Dans la version 1.3, le défaut est attribut ou élément.

```
<script type='text/ng-template' id='poneyTemplate'>
  <h1>Poney</h1>
</script>
```

```
app.directive('poney', function() {
  return {
    templateUrl: 'poneyTemplate'
  };
});
```

N'oubliez pas de mettre cette balise `script` dans le scope de l'application Angular, bien sûr.

9.6.3. Restrict

Par défaut la directive sera donc restreinte à la forme par attribut (ou par élément depuis la version 1.3). Il est possible de changer le comportement par l'utilisation de l'option `restrict` en utilisant les valeurs :

- `A` pour attribut
- `E` pour élément
- `C` pour classe CSS
- `M` pour… commentaire.

Il est important que toute l'équipe utilise la même règle pour éviter les confusions : chaque option a ses défenseurs et ses détracteurs, choisissez celle que vous trouvez le plus à votre goût. Les éléments sont cependant les plus utilisés pour définir de nouveaux composants et les attributs pour ajouter du comportement à un élément.

9.6.4. Transclude

Vous avez dû remarquer que, jusque là, le contenu originel de la directive est perdu lors du remplacement. Il est en fait possible de le conserver grâce à l'option `transclude`.

```
app.directive('poney', function() {
  return {
    transclude: true,
    template: '<h1><div ng-transclude></div> Poney</h1>'
  };
});
```

Avec cette directive :

```
<div poney>Mega</div>
```

donnera

```
<h1><div>Mega</div> Poney</h1>
```

Attention cependant à quelques subtilités avec le scope lors de l'utilisation de transclude[7].

Vous pouvez également jouer avec les priorités des différentes directives (savoir laquelle sera exécutée avant les autres). Cela peut être utile si vous envisagez de cumuler plusieurs directives sur un même élément. Les options utiles dans ce cas là sont 'priority' et 'terminal'.

9.6.5. Link

Pour l'instant une directive ressemble plutôt à une solution de template overkill, mais les directives permettent également de modifier le DOM et de gérer des événements. Vous lirez partout que l'on code différemment en Angular qu'en jQuery et que l'on ne manipule pas le DOM, parce que c'est le mal. Et c'est vrai, mais parfois, on est obligé de modifier le DOM, et c'est dans ce cas-là que les directives se rendent utiles.

Les directives Angular ont une option nommée `link` (rien à voir avec Zelda) qui permet de définir une fonction ayant accès au scope, à l'élément racine de la directive, et aux attributs de l'élément portant la directive. L'élément est un élément jQuery Lite, avec donc quelques fonctions[8] de manipulation à la jQuery. Nous allons voir ça un peu plus loin. L'accès au scope vous permet aussi d'initialiser certaines valeurs :

```
app.directive('poney', function() {
  return {
    link: function(scope, element, attributes){
      scope.poney = { color: 'Red'};
    },
    template: '<h1>{{poney.color}} Poney</h1>'
  };
```

[7] http://stackoverflow.com/questions/14481610/angularjs-two-way-binding-not-working-in-directive-with-transcluded-scope/14484903#14484903
[8] http://docs.angularjs.org/api/angular.element

```
});
```

ou de définir des fonctions :

```
app.directive('poney', function() {
  return {
    link: function(scope, element, attributes) {
      scope.turnBlue = function(){
        scope.poney = { color: 'Blue'};
      }
    },
    template: '<h1 ng-click="turnBlue()">{{poney.color}} Poney</h1>'
  };
});
```

Et cela permet donc de modifier le DOM si besoin. Imaginons que nous voulions une directive qui nous affiche le nombre de secondes écoulées (quoi, il vous plaît pas mon exemple ?). Il faut donc qu'à chaque seconde, nous changions le contenu de la directive. Pour avoir une fonction appelée chaque seconde, nous pouvons utiliser le service Angular `$interval` et l'injecter dans la directive. Puis, à chaque seconde, utiliser `element.text()` pour modifier le contenu avec le nombre de secondes. On obtient :

```
app.directive('secondsElapsed', function($interval) {
  return {
    link: function(scope, element, attributes) {
      var seconds = 0;
      var timeoutId = $interval(function() {
        element.text(seconds++)
      }, 1000);
      element.on('$destroy', function() {
        $interval.cancel(timeoutId);
      });
    }
  };
});
```

Il est important de noter que l'injection du scope, de l'élément et des attributs se fait selon leur ordre dans les paramètres et non selon leur nom. Si vous déclarez les paramètres dans un autre ordre, vous risquez d'être surpris ! En revanche vous pouvez utiliser les noms qui vous plaisent, mais pour aider vos collègues, mieux vaut utiliser ceux-ci.

Vous aurez probablement remarqué que l'exemple contient quelques lignes encore jamais vues :

```
element.on('$destroy', function() {
  $interval.cancel(timeoutId);
});
```

Il s'agit tout simplement d'éviter les fuites mémoires : si on lance une fonction grâce à $interval, le navigateur l'exécutera toutes les X millisecondes, même si la directive n'est plus affichée à l'écran. Pour éviter d'exécuter des traitements inutiles, on écoute sur un événement particulier, $destroy, émis par AngularJS lorsque la directive est supprimée du DOM. A la réception de cet événement, on annule alors la promise : fuite évitée !

9.6.6. Scope

Nous avons vu que la directive pouvait utiliser le scope englobant dans ses templates. Mais cela devient un peu problématique si l'on veut faire un composant réutilisable : il faudrait changer le scope englobant autour de chaque directive. Heureusement, il existe une possibilité pour isoler le scope des directives et seulement transmettre des paramètres. Un scope isolé, contrairement aux scopes des controllers imbriqués, n'hérite pas prototypalement du scope englobant.

Pour isoler le scope :

```
app.directive('poneyColorIsolate', function(){
  return {
    scope: {},
    template: '<h1>{{poney.color}} Poney</h1>'
  };
});
```

La directive déclare son propre scope et ne verra donc pas la couleur initialisée par le scope parent. Mais comment transmettre des paramètres à la directive ? Il existe trois façons de procéder, selon ce que vous voulez transmettre.

Pour passer une valeur, comme une chaîne de caractères, vous pouvez utiliser '@', comme suit :

```
<div poney-color-arobase color="Blue">This text will be removed</div>

app.directive('poneyColorArobase', function(){
  return {
```

```
      scope: {
        poneyColor: '@color'
      },
      template: '<h1>{{ poneyColor }} Poney</h1>'
    };
  });
```

La variable `poneyColor` de la directive sera alors initialisée avec l'attribut `color` passé en paramètre. Si la variable du scope de la directive se nommait `color`, alors on pourrait simplifier la déclaration du scope en :

```
scope: {
  color: '@'
},
```

et Angular ferait le lien avec l'attribut de même nom. A noter que les attributs doivent s'écrire avec la notation "dash-limited" et les objets du scope en "camelCase", de la même façon que les noms des directives (voir l'exemple ci-dessous). Relisez cette phrase, vous allez vous faire avoir.

Si vous souhaitez passer une référence vers un objet, et que toute modification faite sur cet objet par la directive soit vue par le scope parent et inversement, vous devez utiliser '=', comme suit :

```
<div poney-color-equal="poney">This text will be removed</div>

app.directive('poneyColorEqual', function(){
  return {
    scope: {
      poney: '=poneyColorEqual'
    },
    template: '<h1>{{ poney.color }} Poney</h1>'
  };
});
```

La variable `poney` de la directive sera alors initialisée avec l'attribut `poney-color-equal` passé en paramètre. Cette relation est différente de la précédente car la variable poney est alors "bindée" à la variable passé en paramètre. On peut voir ce passage de paramètre comme un passage par référence :

- si le controller change la valeur de `poney.color`, alors le poney affiché par la directive changera également de couleur

- si le controller remplace le poney de son scope par un autre poney, la directive référencera et affichera ce nouveau poney
- si la directive change la valeur de `poney.color`, alors le poney contenu dans le scope du controller changera également de couleur
- si la directive remplace le poney de son scope par un autre poney, alors le scope du controller référencera ce nouveau poney

Enfin, vous pouvez également passer une fonction à une directive avec la notation '&'.

```
<div poney-color-ampersand turn-color="turnBlue()">This text will be
removed</div>
```

```
$scope.turnBlue = function(){
  alert('blue!');
}

app.directive('poneyColorAmpersand', function(){
  return {
    scope: {
      turnColor: '&'
    },
    template: '<h1 ng-click="turnColor()">Click Poney!</h1>'
  };
});
```

Pratique ! Cela permet à vos directives d'accepter des fonctions qui définiront leur comportement selon l'endroit où vous les utilisez.

Si vous voulez une description encore plus détaillée et imagée du fonctionnement des scopes en fonction des différentes options de la directive, consultez cette page du wiki Angular[9].

9.6.7. Controller

Il est également possible d'extraire le comportement de la fonction `link` pour le factoriser dans un `controller` qui sera plus simple à tester unitairement. C'est d'ailleurs une pratique très largement utilisée dans l'excellent module Angular UI.

```
app.directive('poney', function(){
```

[9] https://github.com/angular/angular.js/wiki/Understanding-Scopes

```
    return {
      link: function(scope, element, attributes){
        scope.turnBlue = function(){
          scope.poney = { color: 'Blue'};
        }
      },
      template: '<h1 ng-click="turnBlue()">{{poney.color}} Poney</h1>'
    };
  });
```

pourrait devenir

```
  app.directive('poney', function(){
    return {
      controller: 'poneyController',
      template: '<h1 ng-click="turnBlue()">{{poney.color}} Poney</h1>'
    };
  });

  app.controller('poneyController', function($scope){
    $scope.turnBlue = function(){
      $scope.poney = { color: 'Blue'};
    }
  });
```

9.6.8. Require

L'attribut `require` permet d'injecter le controller d'autres directives à la fonction `link` comme quatrième argument. Cela permet donc a plusieurs directives de collaborer. L'utilisation la plus répandue consiste à ajouter `ngModel` pour récupérer le controller associé au modèle, et pouvoir manipuler ses méthodes de validation. C'est un excellent moyen de faire une directive de validation custom pour un champ de formulaire.

```
  <form name='poneyForm'>
    Black Mamba is forbidden.
    <input name='name' required ng-model='poney.name' forbidden-name/
><br/>
    <span ng-show='poneyForm.name.$dirty && poneyForm.name.
$error.forbiddenNameError'>This poney name is forbidden</span>
  </form>
```

```
app.directive('forbiddenName', function(){
  return {
    require: 'ngModel',
    link: function(scope, elem, attributes, modelCtrl){
      // view to model update (i.e view has changed)
      modelCtrl.$parsers.unshift(function(value) {
        // test the name
        var valid = value !== 'Black Mamba';
        console.log('is name forbidden?', valid);
        modelCtrl.$setValidity('forbiddenNameError', valid);
        // if it's valid, return the value to the model,
        // otherwise return undefined.
        return valid ? value : undefined;
      });

      // model to view update (i.e model has changed)
      modelCtrl.$formatters.unshift(function(value) {
        var valid = value !== 'Black Mamba';
        modelCtrl.$setValidity('forbiddenNameError', valid);
        // return the value or nothing will be written to the DOM.
        return value;
      });
    }
  }
})
```

modelCtrl est le fameux NgModelController détaillé plus haut. Il expose un tableau de $parsers, appelés lorsque la vue change la valeur, et un tableau de $formatters, appelés lorsque le modèle a changé et qu'il faut mettre à jour la vue. On peut donc ajouter nos propres parsers et formatters dans la liste, pour interdire certaines valeurs.

Non seulement le modèle ne sera pas mis à jour si la directive l'interdit, mais en plus notre erreur 'custom' se comportera comme une erreur de validation standard, règles CSS s'appliquant automatiquement, etc.

Si vous êtes un vrai aventurier, vous croiserez des directives qui utilisent require pour injecter des directives en passant leur nom précédé d'un \^, d'un ? ou des deux ?^. Cela surprend un peu mais cela a une vraie utilité :

- par défaut si l'on ajoute require: 'ngModel', la directive lancera une erreur si ngModel n'est pas appliquée sur le même élément.
- avec require: '?ngModel', la directive reçoit null si ngModel n'est pas appliquée sur le même élément.

- avec require: '^ngModel', la directive cherche ngModel sur les éléments parents et lance une erreur si elle ne trouve pas ngModel.
- enfin, avec require: '^?ngModel', vous avez compris : la directive reçoit null si ngModel n'est trouvé sur aucun des parents de l'élément.

A noter, certains changements depuis la version 1.3. Le mécanisme de validation a été légèrement revu et reprend le principe de validateurs de AngularDart. L'idée est de découpler la validation des champs qui était pour l'instant à la charge des `$formatters` ou des `$parsers`. Ceux-ci existent toujours et peuvent transformer les valeurs comme ils le souhaitent, mais c'est maintenant les `$validators` qui choisissent si une valeur est valide ou non pour le champ, en prenant en compte les différentes directives présentes sur le champ en question (`ngMinLength`, `ngPattern`, nos propres directives, etc…).

Concrètement, cela va permettre d'avoir des champs qui ne passent pas la validation mais que l'on voudrait quand même soumettre parce que leur valeur est "assez" bonne. Il est par exemple impossible pour l'instant de stocker une adresse email incorrecte. Si l'on écrit le prochain Gmail, il est donc impossible de sauver un draft de mail, pour corriger plus tard l'adresse du destinataire !

Maintenant le modèle sera mis à jour quand même, et le controller saura si la valeur est valide ou non : à nous de décider si cela nous convient.

Cela impacte également la façon d'écrire une directive de validation custom, pour les rendre plus simples. Par exemple, pour n'autoriser que des logins avec au moins un chiffre :

```
app.directive('atLeastOneNumber', function(){
  return {
    require: 'ngModel',
    link: function(scope, element, attributes, modelCtrl){
      modelCtrl.$validators.atLeastOneNumber = function(value){
        return (/.*\d.*/).test(value);
      }
    }
  }
});
```

Plus besoin de s'embêter avec les `$parsers` et `$formatters` !

Voilà l'essentiel pour les directives. On ne va pas se voiler la face, cela ne vous empêchera de galérer à faire votre première directive un peu velue (transclusion,

récursivité, cycle de vie avec compile, pre-link, post-link…). La construction de directives est compliquée dans la vraie vie. Si le sujet vous intéresse, les meilleurs exemples de code, et de très loin, sont dans le module Angular UI Bootstrap, qui transforme les composants de Twitter Bootstrap en directives.

9.6.9. Tests

Il faudra bien sûr tester vos directives. Pour cela vous aurez besoin d'un élément du DOM portant la directive et d'un scope. Il faudra ensuite compiler l'élément avec le scope, grâce au service $compile, puis comparer le HTML généré.

```
it('should be a poney image', inject(function ($compile) {
    element = angular.element('<poney name="{{ name }}"></poney>');
    scope.name = 'Pink';
    element = $compile(element)(scope);
    scope.$digest();
    expect(element.html()).toBe('<div><img ng-src="images/poney-pink.png"
 alt="Pink" src="images/poney-pink.png"></div>');
  }));
```

Chapter 10. Concepts avancés

Si vous arrivez ici, félicitations, vous avez les bases et un peu plus pour vous lancer. Dans cette partie, nous verrons des techniques un peu plus avancées ou nous jetterons un œil sous le capot d'AngularJS pour comprendre les mécanismes internes.

10.1. Intercepteurs HTTP

Il est parfois nécessaire de systématiquement intercepter les requêtes ou les réponses HTTP, par exemple dans le cas de la sécurité. Pour cela, Angular fournit le mécanisme d'intercepteur. Il est possible de les ajouter à la variable $httpProvider.interceptors et ceux-ci sont de plusieurs types selon l'action que vous désirez effectuer (sur un envoi de requête, une erreur de requête, une réponse ou une erreur de réponse).

Par exemple, vous pouvez envoyer votre utilisateur sur la page de login si la requête qu'il a effectué a échoué en raison d'un problème d'authentification (erreur 401).

```
    angular.module('interceptors').factory('authenticationInterceptor',
['$q', '$location', function($q, $location) {
      return {
        responseError: function(rejection) {
          if (rejection.status == 401) {
            $location.path('/login');
          }
          return $q.reject(rejection);
        }
      };
    }]);

    $httpProvider.interceptors.push('authenticationInterceptor');
```

On déclare donc un nouvel intercepteur grâce à la méthode factory, avec pour dépendances le service de promises ($q) et le service de changement d'URL ($location). La factory crée un nouvel intercepteur nommé 'authenticationInterceptor', de type 'responseError', et qui, selon le code d'erreur, soit redirige vers la page de login, soit poursuit son chemin à travers les promises (puisqu'il est possible d'en enchaîner plusieurs). Enfin l'intercepteur est ajouté à la liste des intercepteurs de l'application.

10.2. Événements

Angular possède un système d'événements que l'on peut écouter ou émettre dans son application. Cela peut se révéler un très bon moyen de communication entre différentes parties de l'application, en envoyant nos propres événements, ou pour surveiller certains événements envoyés par le framework ou des modules tiers. Le scope expose plusieurs méthodes utilitaires :

- `$on(name, listener)` qui permet d'attacher un listener à un événement précis
- `$emit(name)` qui permet d'émettre l'événement correspondant vers les scopes parents.
- `$broadcast(name)` qui permet d'émettre l'événement correspondant vers les scopes enfants.

Les événements permettent d'éviter un couplage fort entre divers éléments de l'application, et il est donc fréquent qu'on veuille émettre un événement à destination de tous les controllers (enfants ou parents) ou même de services ou de directives. Dans ce cas, on peut utiliser la méthode `$broadcast` sur l'object `$rootScope`. Tous les listeners, y compris ceux ajoutés au $rootScope lui-même, seront ainsi notifiés.

Certains événements sont envoyés par le framework sans actions de notre part : c'est par exemple le cas de l'événement `$destroy` qui est émis lorsqu'un élément est détruit. Il est parfois intéressant d'écouter cet événement pour éviter d'éventuelles fuites mémoires, suite à un traitement récurrent que l'on aurait oublié d'arrêter :

```
app.directive('secondsElapsed', function($interval){
  return {
    link: function(scope, element, attributes){
      var seconds = 0;
      timeoutId = $interval(function() {
        element.text(seconds++)
      }, 1000);
      element.on('$destroy', function() {
        $interval.cancel(timeoutId);
      });
    },
    template: '<div></div>'
  };
});
```

Dans l'exemple ci-dessus, lorsque la directive est détruite (c'est-à-dire qu'elle disparaît du navigateur, suite à la navigation vers une autre page par exemple), l'événement `$destroy` est lancé sur l'élément. Nous pouvons ainsi nous abonner à l'événement pour détruire proprement notre traitement récurrent et ainsi éviter toute fuite mémoire.

D'autres événements sont lancés par le framework au changement d'URL (`$locationChangeSuccess`, `$locationChangeStart`), de route (`$routeUpdate`, `$routeChangeStart`, `$routeChangeSuccess`, `$routeChangeError`) ou de contenu (`$includeContentLoaded`, `$includeContentError`, `$viewContentLoaded`).

10.3. Elément

Vous avez pu constater dans certains exemples l'utilisation de `angular.element` : cet objet représente l'élément DOM manipulé. Angular délègue toute la partie manipulation de DOM à jqLite ou à jQuery si la librairie est présente (elle doit être chargée avant Angular pour être détectée). Plusieurs méthodes sont disponibles sur cet objet, que vous connaissez sans doute si vous avez déjà utilisé jQuery. La documentation de jQuery vous donnera une bonne vue de toutes les possibilités, notons ici les plus couramment utilisées : `addClass()`, `removeClass()`, `toggleClass()`, `text()`, `val()`, `attr()`, `css()`, `children()`, `parent()`, `append()`, `prepend()`...

jqLite n'est qu'un sous-ensemble de jQuery : si vous avez besoin de fonctionnalités un peu plus avancées, il vous sera nécessaire d'inclure jQuery.

10.4. Dirty checking

Mais comment fonctionne Angular pour savoir si un objet du modèle a été mis à jour, afin de mettre à jour la vue ? Il vérifie tout simplement à chaque événement !

Il faut savoir que JavaScript s'exécute par cycle :

1. Le navigateur attend un événement utilisateur, un timer ou une réponse serveur.
2. Il exécute le callback correspondant à l'événement. Ce callback est le JavaScript exécuté et peut modifier la structure DOM de la page.
3. Chaque modification du DOM met à jour la page instantanément.

Par exemple, avec jQuery :

```
var btn = $('#btn-save');
```

```
btn.on('click', save);

var save = function() {
  $.post(url, data, success);
};

var success = function() {
  alert("success!");
};
```

Au chargement de la page, le navigateur exécute ce bout de script. Il récupère le bouton puis lui ajoute un listener de click. Une fois ces actions réalisées, le navigateur se met en attente d'une action utilisateur. Si vous ne faites rien, le navigateur ne fait rien, aucun JavaScript n'est exécuté.

Si l'utilisateur clique sur le bouton de sauvegarde, alors le navigateur est réveillé, rentre dans un nouveau cycle d'exécution et appelle le gestionnaire de clic : la fonction 'save'. Celle-ci fait un simple appel serveur, puis le JavaScript stoppe et le navigateur se rendort.

Lorsqu'une réponse est renvoyée par le serveur, le navigateur entre dans un nouveau cycle d'exécution JavaScript et exécute la fonction 'success' (callback de l'appel serveur précédent), puis se rendort.

Ce fonctionnement par cycle est simple et transparent pour les développeurs. A chaque fin de cycle, la page retourne dans un état stable jusqu'au prochain événement provoquant une modification du DOM.

Comment font donc les différents frameworks pour surveiller l'état des différents objets ? Pour l'instant, aucune technique native ne permet de savoir si un objet a été modifié. Cela changera dans le futur, avec notamment la possibilité d'avoir une fonction 'Object.observe'[1] dans l'API JavaScript (reportée de ES6 à ES7, mais déjà disponible dans certains navigateurs).

En attendant, plusieurs solutions sont employées par les frameworks les plus connus :

- utiliser des objets spéciaux, où chaque changement se fait par un setter obligatoirement. C'est la technique utilisée par Ember.js par exemple[2], où les modèles héritent d'une classe spéciale[3]. Le framework sait ainsi lorsqu'un attribut

[1] http://wiki.ecmascript.org/doku.php?id=harmony:observe
[2] http://emberjs.com/guides/models/working-with-records/#toc_modifying-attributes
[3] http://emberjs.com/guides/object-model/classes-and-instances/

est modifié et peut impacter la vue en conséquence. Cette technique possède l'avantage d'être performante, mais contraint le développeur à utiliser des objets spéciaux et à toujours mettre à jour les attributs par setter et non directement.

- l'autre technique, beaucoup plus brutale dans son approche, consiste à vérifier l'état d'un objet et des attributs à chaque fin de cycle d'exécution. Ainsi si une valeur a été modifiée, le framework met à jour la vue en conséquence. Il est possible de surveiller n'importe quel objet, de se passer de l'utilisation d'objets spéciaux et de setters. Cette stratégie, plus confortable mais moins efficace en terme de performance, est néanmoins bien optimisée par l'équipe Angular et permet de surveiller beaucoup de valeurs simultanément dans un navigateur moderne. En pratique, ce nombre est largement suffisant et aucun ralentissement ne se fait sentir. Il faut cependant être vigilant sur des applications d'envergure et ne pas surveiller des attributs inutiles, ou penser à ne plus les surveiller lorsque ce n'est plus nécessaire.

Angular vient s'insérer dans cette boucle et sépare l'exécution JavaScript en deux phases distinctes : la phase classique et la phase Angular. La phase Angular est appelé par la fonction $apply(). En fait vous appellerez rarement cette fonction vous-même, votre code Angular le fera pour vous sans appel explicite. Par exemple, prenons un simple 'input' mettant à jour une 'div' :

```
<input type="text" ng-model="name"/>
<div>Hello {{name}}!</div>
```

Même si cela n'est pas forcément évident, 'input' est ici une directive Angular et pas un simple 'input'. Ainsi, chaque frappe clavier de l'utilisateur dans le champ appelle la directive 'input' et plus particulièrement un listener sur ce champ (si l'on raisonnait en JavaScript classique, avoir une directive 'input' est un peu comme avoir un 'input' HTML avec un listener de l'événement input[4], supporté par la plupart des navigateurs ou simulé par plusieurs listeners simultanés sur 'keyDown', 'change', 'paste', 'cut'). Ce listener va lui-même appeler $apply(), entrainant ainsi le cycle d'exécution Angular, sans travail de votre part. Tout ce code est caché dans la directive, magique non ?

Le code exécuté par $apply(), à l'inverse de ce que fait du code JavaScript plus classique (comme jQuery), ne modifie pas le DOM. Il ne fait que modifier des objets JavaScripts placés dans le scope. Ici, notre objet 'name'.

La fonction $apply() rentre alors dans la phase suivante, celle où toutes les valeurs qui ont changé depuis la dernière exécution sont collectées. Cette phase est appelée

[4] https://developer.mozilla.org/en-US/docs/Web/Reference/Events/input

la phase 'digest'. Cette phase est en réalité une boucle qui va itérer jusqu'à ce que le modèle se stabilise (le changement d'un champ peut déclencher le changement d'un autre champ, etc…). Ainsi notre objet 'name' est bien à jour, mais si nous avons une expression qui dépend de 'name' alors cette expression n'est pas encore à jour tant que $digest() n'est pas appelé une deuxième fois (vous pouvez voir en détail le fonctionnement du digest dans le chapitre consacré aux concepts avancés).

Pour garder la trace de toutes les variables à observer, Angular utilise des 'watchers'. A chaque 'digest', les différents watchers sont appelés un à un tant qu'aucun listener n'est appelé. Un watcher pouvant changer le modèle, ce changement génèrera un appel à un listener et donc un nouveau tour de 'digest', ainsi de suite, jusqu'à ce que le modèle soit stable.

Il est ainsi possible d'obtenir une boucle infinie (un watcher1 modifiant un champ observé par un watcher2 qui modifie le champ observé par le watcher1…). Pour éviter ce risque, le nombre maximal d'itération est fixé à 10 et vous obtiendrez le message '"10 $digest() iterations reached. Aborting!"' si vous dépassez ce seuil. $digest() n'est jamais appelé directement, et $apply() très rarement. Il est cependant nécessaire de le faire si nous modifions le modèle en dehors de l'application Angular, par exemple à la réception d'un message par Websocket (puisqu'Angular ne fournit aucun service pour gérer cette communication et faire le $apply pour nous). Un appel à $apply() permettra alors à Angular de savoir que le modèle a changé et qu'il doit lancer un nouveau tour de 'digest' pour mettre à jour les différents champs et vues en conséquence.

Ainsi une application Angular chargeant l'exemple précédent, a procédé comme suit :

1. Compilation
    a. La directive 'input' ajoute un listener sur l'événement 'input'.
    b. Chaque action sur le input déclenche un appel à $apply().
    c. $apply met le modèle à jour et appelle $digest().
    d. Tous les watchers remarquent le changement.
    e. Les expressions qui utilisent la valeur de l'input sont rafraîchies.

10.5. Watchers

Il est possible d'ajouter manuellement des watchers, si vous avez par exemple une variable qui dépend d'une autre. Supposons que tous les changements de ma variable 'name' doivent incrémenter un compteur 'updates'. On pourrait très bien avoir le code suivant :

```
$scope.$watch('name', function(newValue, oldValue) {
  $scope.updates = $scope.updates + 1;
});
```

Un watcher prend une expression en entrée et un listener. L'expression est appelée à chaque '$digest' et renvoie la valeur à surveiller. Cette expression doit renvoyer un résultat consistant, non dépendant de l'exécution, puisqu'à chaque phase de '$digest', elle peut être appelée plusieurs fois. Le listener sera quand à lui appelé seulement si la valeur a changé, c'est à dire si oldValue et newValue ne sont pas égales. Pour définir cette égalité, il est possible de passer un troisième argument, un booléen pour indiquer au watcher si la comparaison se fait par égalité (c'est la fonction 'angular.equals' qui sera appelé) plutôt que par référence.

Une copie profonde de chaque valeur sera effectuée afin de la conserver et de la comparer au prochain cycle : attacher un watcher à des objets complexes n'est donc pas sans conséquence sur les performances de votre application. Le corps du listener peut éventuellement changer le modèle, comme c'est le cas dans notre exemple : 'updates' change de valeur. Un autre tour de '$digest' est alors effectué, afin de vérifier qu'aucun listener n'est déclenché par le changement de valeur de 'updates'. Lorsqu'un tour de '$digest' ne déclenche aucun listener, le modèle est stable pour Angular.

A noter que les watchers sont exécutés dans l'ordre de leur ajout.

Enfin, il est possible d'arrêter de surveiller une valeur. Pour cela, la déclaration d'un watcher renvoie une fonction de désenregistrement, qu'il suffit d'appeler pour arrêter la surveillance.

```
var unregisterNameWatcher = $scope.$watch('name', function(newValue,
 oldValue) {
  $scope.updates = $scope.updates + 1;
});
// quand on ne veut plus écouter:
unregisterNameWatcher();
```

Depuis la version 1.3, le `$scope` est enrichi d'une nouvelle méthode pour observer un ensemble de valeurs : `watchGroup`. Elle fonctionne sensiblement comme la méthode `watch` déjà existante, à la nuance près qu'au lieu d'observer une seule valeur, elle en observe une collection :

```
$scope.$watchGroup(['user1', 'user2', 'user3'], function(newUsers,
 oldUsers) {
```

```
  // the listener is called any time one of the user is updated
  // with newUsers representing the new values and oldUsers the old ones.
);
```

10.5.1. Alternative

Utiliser des watchers rend le code plus complexe, et plus difficile à tester. La plupart du temps, plutôt que de mettre à jour un attribut 'B' lorsqu'un attribut 'A' change, il est plus simple de simplement utiliser une fonction. Par exemple, supposons qu'on ait un attribut 'name' et un attribut 'message', dont la valeur doit toujours être 'Hello <name>'. On pourrait utiliser un watcher sur 'name', qui met à jour l'attribut 'message' :

```
$scope.$watch('name', function(newValue, oldValue) {
  $scope.message = "Hello " + newValue;
});
```

```
<input type="text" ng-model="name"/>

{{ message }}
```

Mais une solution bien plus simple et testable consiste à simplement utiliser une fonction au lieu de l'attribut message :

```
$scope.getMessage = function() {
    return "Hello " + $scope.name;
}
```

```
<input type="text" ng-model="name"/>

{{ getMessage() }}
```

En effet la valeur de l'expression getMessage() dans la vue est elle aussi surveillée automatiquement par un watcher. Et donc, tout changement de la valeur de 'name' entrainera la réévaluation de l'expression getMessage(), et affichera donc la nouvelle valeur du message retournée par la fonction.

Il est assez rare de devoir utiliser des watchers dans une application, si c'est la solution qui vous vient à l'esprit, creusez un peu plus pour trouver la bonne alternative.

10.6. ngModelOptions

Jusqu'ici la directive `ngModel` lançait une mise à jour du model à chaque action utilisateur (donc à chaque frappe clavier par exemple, les validations étaient exécutées, et une boucle `$digest` se déroulait). Cela fonctionne très bien, mais peut être pénalisant si vous avez une page avec beaucoup de bindings surveillés et à mettre à jour. C'est ici qu'intervient cette nouvelle directive depuis la version 1.3 : `ngModelOptions`.

Il devient ainsi possible de changer ce comportement par défaut et de déclencher ces différents événements selon votre envie, en passant un objet représentant les options à cette directive. La pull-request était ouverte depuis plus d'un an, comme quoi il ne faut jamais désespérer quand on contribue à un projet populaire ! Tous les inputs portant un `ngModel` vont alors chercher ces options sur l'élément portant la directive ou sur ses ancêtres.

Ces options peuvent contenir un champ `updateOn` qui définit sur quel événement utilisateur la mise à jour du modèle doit être effectuée. Vous pouvez par exemple ne la déclencher que sur la perte de focus du champ :

```
<input name="guests" type="number" ng-required="true" ng-
model="event.guests" max="10" ng-model-options="{ updateOn: 'blur' }">
```

Il est possible de passer plusieurs événements par exemple 'blur' pour la perte de focus et 'paste' pour déclencher la mise à jour si l'utilisateur colle une valeur dans le champs.

```
<input name="guests" type="number" ng-required="true" ng-
model="event.guests" max="10"
ng-model-options="{ updateOn: 'blur paste' }">
```

Pour conserver le comportement normal, utilisez l'événement 'default' : utile si vous voulez simplement ajouter de nouveaux événements.

Les options peuvent également contenir un champ `debounce` qui spécifie un temps d'attente depuis le dernier événement avant de lancer la mise à jour. Par exemple, seulement une seconde après la dernière action utilisateur :

```
<input name="guests" type="number" ng-required="true" ng-
model="event.guests" max="10"
ng-model-options="{ debounce: 1000 }">
```

La valeur peut être un nombre si le même temps doit être attendu pour tous les événements, ou un objet avec comme attribut chacun des événements pour lequel vous voulez spécifier un debounce et la valeur de celui-ci.

```
<input name="guests" type="number" ng-required="true" ng-
model="event.guests" max="10"
ng-model-options="{ debounce: { default: 0, paste: 500 }}">
```

On peut donc également définir cette option pour toute une page ou tout un formulaire, puisque chaque `ngModel` recherchera les options également sur ses ancêtres :

```
<form name="eventForm" ng-model-options="{ debounce: 1000 }">
  <input name="week" type="date" ng-required="true" ng-
model="event.weekNumber"
    >
  <input name="guests" type="number" ng-required="true" ng-
model="event.guests" max="10">
</form>
```

Cette nouvelle option est assez pratique, notamment pour un cas très précis qui arrive parfois. Imaginez que vous ayez un formulaire d'inscription et que le champ login vérifie si la valeur entrée par l'utilisateur est disponible côté serveur : on voit ici l'intérêt du debounce pour attendre que l'utilisateur termine son entrée de valeur, ou d'attendre la perte de focus pour faire la vérification, plutôt que de lancer une requête HTTP à chaque frappe clavier !

```
<!-- poney-unique-name is a custom validator, implemented as a directive,
 checking with the server if the value is not already taken by another
 poney -->
<input name="name" type="text" ng-required="true" poney-unique-name ng-
model="poney.name" ng-model-options="{updateOn: 'blur'}">
```

Un problème peut cependant apparaître avec cette approche. Imaginez un bouton 'Clear' sur votre formulaire qui vide les champs. Si jamais l'utilisateur remplit un champ avec un debounce, puis clique sur 'Clear', les champs vont se vider, puis le debounce s'exécuter et remplir à nouveau le champ ! Il faut donc penser à supprimer tous les debounces en attente dans le code de la méthode `clear()` appelée par le clic sur le bouton, en utilisant une nouvelle méthode exposée par le controller du champ, appelée `$rollbackViewValue`. Celle-ci est également disponible sur le formulaire pour agir sur tous les champs d'un coup.

Vous pouvez jouer avec cette nouvelle fonctionnalité, et ses limites, avec ce Plunker[5].

Une autre option est disponible pour `ngModelOptions` : celle d'activer un mode getter/setter pour votre modèle.

```
<input ng-model="todo.name" ng-model-options="{ getterSetter: true }" />
```

Votre controller devra alors posséder un attribut du scope `todo.name` qui sera une fonction. Si celle-ci est appelée sans paramètres, elle renverra la valeur de l'attribut, sinon elle modifiera éventuellement la valeur de l'attribut avec la valeur du paramètre, de la façon dont vous le déciderez. Un getter/setter qui fonctionnerait sans modification ressemblerait à celui-ci :

```
var _name = 'internal name'
$scope.todo {
  name: function(newName){
    _name = angular.isDefined(newName) ? newName : _name;
    return _name;
  }
}
```

L'intérêt principal réside dans le fait que l'on peut alors avoir une représentation interne différente de la représentation envoyée à l'utilisateur. Cela peut remplacer ce qui aurait été précédemment fait avec un watcher, voir cet exemple avec $location[6].

Vous pouvez alors accéder à votre attribut par son getter avec `todo.name()` ou changer sa valeur avec `todo.name('new name')`.

10.7. One time binding

Il est, depuis la version 1.3, possible de demander à Angular de cesser d'observer une expression une fois celle-ci évaluée. Il suffit pour cela de la précéder de `::` dans vos templates. Ainsi tout changement ultérieur ne sera pas répercuté à l'affichage.

```
$scope.name = 'Cédric';
```

```
<!-- updating the input will not affect the displayed div -->
<input type="text" ng-model="name">
```

[5] http://plnkr.co/edit/94oLhzYOZeMcJcUBrXKq?p=preview
[6] https://github.com/angular/angular.js/commit/5963b5c69f5dc145c9535f734c43ee6027ae24bd

```
<div>Bind once {{ ::name }}</div>
```

Cela fonctionne aussi avec les collections, par exemple lorsqu'elles sont utilisées dans un `ng-repeat` :

```
$scope.tasks = ["laundry", "running", "shopping"];
$scope.remove = function(){ $scope.tasks.pop(); };

<!-- removing tasks will not affect the displayed list -->
<button ng-click="remove()">remove</button>
<div ng-repeat="task in ::tasks">
  {{ task }}
</div>
```

Bien sûr, c'est plus sympa à essayer dans le Plunker associé[7].

10.8. Les services internes

Pour votre culture, voici quelques-uns des services utilisés en interne par le framework, mais très rarement par les développeurs.

10.8.1. $document/$window

Les objets JavaScript window et window.document sont wrappés dans des services si vous en avez besoin dans votre application Angular. Par exemple, si vous voulez un petit côté années 90 dans votre application, vous pourriez être tenter de créer des alertes.

```
    $window.alert("I'm an alert!");
```

Les alertes étant difficiles à tester, il sera possible de créer un faux service $window pour nos tests, et de simuler un appel à la méthode alert() !

10.8.2. $templateCache

Lorsqu'un template est chargé par Angular, il est mis dans un cache spécifique nommé $templateCache. Il est possible de charger de nouveaux templates manuellement ou de les récupérer avec ce service :

[7] http://plnkr.co/edit/bwX7SLqpUNv9Q5KXVgOj?p=preview

```
    // adding template in html
    <script type="text/ng-template" id="hello-template">
      Hello {{ world }}
    </script>
    // loading in html
    <div ng-include="hello-template"></div>
    // adding template in JS
    $templateCache.put('hello-template', 'Hello {{ world }}');
    // loading in JS
    $templateCache.get('hello-template');
```

10.8.3. $compile

Si ce service ne sera que très rarement utilisé par les développeurs, il est en revanche très utilisé à l'intérieur d'Angular. Il sert en effet à compiler un élément du DOM avec un scope pour donner le résultat attendu, avec les variables remplacées, les directives introduites etc.

Il est possible de passer une chaîne de caractères pour l'élément ou directement un élément du DOM.

Le rare cas pratique d'utilisation qui vient à l'esprit est le test des directives, où l'on va devoir compiler manuellement un élément de DOM pour voir si la directive produit le résultat escompté.

```
it('should be a poney image', inject(function ($compile) {
    element = angular.element('<poney name="{{ name }}"></poney>');
    scope.name = 'Pink';
    element = $compile(element)(scope);
    scope.$digest();
    expect(element.html()).toBe('<div><img ng-src="images/poney-pink.png"
 alt="Pink" src="images/poney-pink.png"></div>');
  }));
```

10.8.4. $interpolate

Ce service, à usage interne et utilisé par $compile, est chargé de localiser les '{{ }}' de nos templates.

```
    var scope = { poney : { name: 'Red Fury'} };
    var interpolated = $interpolate(' {{ poney.name | uppercase }}')
(scope);
```

```
    // assert that interpolated === 'RED FURY'
```

La particularité intéressante de ce service est qu'il est configurable, par l'intermédiaire de son provider $interpolateProvider, pour accepter d'autres symboles que '{{' et '}}' !

```
    app.config(function($interpolateProvider){
      $interpolateProvider.startSymbol = '[[';
      $interpolateProvider.endSymbol = ']]';
    });
    var scope = { poney : { name: 'Red Fury'} };
    var interpolated = $interpolate(' [[ poney.name | uppercase ]]')
(scope);
    // assert that interpolated === 'RED FURY'
```

Pas tellement conseillé, mais vous saurez que vous pouvez remplacer les traditionnelles 'moustaches' par un autre symbole plus à votre goût.

10.8.5. $parse

Ce service est un peu similaire au précédent, qui l'utilise d'ailleurs, puisqu'il prend une expression et un contexte, et remplace l'expression par la valeur du contexte, si elle est trouvée.

```
    var scope = { poney : { name: 'Red Fury'} };
    var parsed = $parse('poney.name')(scope);
    // assert that parsed === 'Red Fury'
```

10.8.6. $injector

Ce service gère l'injection de dépendance de l'application : il est chargé de lire toutes les déclarations de controllers, directives, filtres, etc… afin de voir quelles dépendances sont utiles, puis d'injecter ces dépendances en question.

L'injecteur contient une map de tous les services de l'application, et il est possible de les récupérer :

```
    var http = $injector.get('$http'); // returns $http service
```

Il est possible d'interroger l'injecteur avec 'has' :

```
    $injector.has('$http'); // returns true
```

```
$injector.has('unknownService'); // returns false
```

Trois façons existent pour appeler l'injecteur sur une fonction. Par inférence, l'injecteur devinant le nom des dépendances par le nom des arguments, façon pratique mais qui ne résiste pas à la minification :

```
$injector.invoke(function($http){});
```

Par annotation, en utilisant une propriété explicite $inject qui liste les dépendances à injecter :

```
function myService($http){};
myService.$inject = ['$http'];
$injector.invoke(myService);
```

La dernière, souvent préférée, avec un tableau dont le dernier argument est la fonction à appeler :

```
$injector.invoke(['$http', function($http){}]);
```

La fonction 'annotate' permet à l'injecteur de lister les dépendances (voir le paragraphe sur l'injection de dépendances).

Pour trouver une dépendance, l'injecteur tente de trouver un provider associé ($http → $httpProvider).

10.8.7. $exceptionHandler

Toutes les exceptions non catchées atterrissent dans ce service, ce qui nous permet de gérer leur traitement. Par défaut, le service appelle simplement $log.error. Mais on peut imaginer une gestion beaucoup plus évoluée, par exemple en envoyant les erreurs au serveur, à des fins analytiques ou de débugging !

Surchargeons le handler d'exception avec le nôtre :

```
    angular.module('handlers').factory('$exceptionHandler', function
($log, $injector) {
      return function (exception, cause) {
        $log.error.apply($log, arguments);
        var http = $injector.get('$http');
```

```
        http.post('/errors', exception);
      };
    });
```

Une nouveauté dans cet exemple : l'utilisation de $injector pour récupérer le service $http. En effet, $http dépend de $exceptionHandler donc Angular ne sait pas lequel injecter dans l'autre, on ne peut donc injecter le service $http directement. On utilise $injector, qui lui n'appelle pas notre handler en cas d'exception, et l'on récupère manuellement le service $http. On continue cependant à logguer en console, grâce à $log.

10.9. Décorateurs

Il reste une méthode exposée par l'object Module qui peut être très pratique : 'decorator'. Cette méthode a comme paramètres le nom du service à décorer et un callback avec comme paramètre $delegate qui représente le service à décorer. Il est ainsi possible d'ajouter des méthodes, de les surcharger…

Vous voulez changer tous les '$log.$warn' en '$log.$error' ?

```
angular.module('services').decorator('$log',
 ['$delegate', function($delegate) {
  $delegate.warn = $delegate.error;
  return $delegate;
}]);
```

Ou ajouter une fonction à un service fourni par une librairie tierce ? Par exemple, pour ajouter une méthode de déconnexion au service 'LoginService' :

```
    angular.module('services').decorator('LoginService',
 ['$delegate', '$http', function($delegate, $http) {
      $delegate.logout = function(){
        return $http.get('/logout');
      };
      return $delegate;
    }]);
```

10.10. Debugging

Angular est fantastique, mais il y a de bonnes chances qu'il vous donne des maux de tête malheureusement. Il arrive de se prendre un mur et de devoir debugger. Les

outils que vous utilisez habituellement en JavaScript sont très utiles (oui, même les console.log, ou plutôt les $log.log …). Mais il y a aussi Batarang qui peut vous aider.

L'extension Batarang pour Chrome est un bon moyen de débugger. Elle permet en effet d'ajouter au Chrome Developer Tools un nouvel onglet, nommé 'AngularJS', avec différentes possibilités :

- Models, qui permet de voir les différents scopes, et de les inspecter.
- Performance, ou sont affichés différents temps de votre application
- Dependencies, pour voir les dépendances des différents composants
- Options, qui permet de tracer sur la page les zones des scopes et bindings.

10.11. Modules indispensables

10.11.1. UI-Router

Si vous trouvez le routeur par défaut un peu limité, vous n'êtes pas le seul. Le projet UI-Router[8] est très souvent utilisé pour sa capacité à pouvoir imbriquer des vues dans d'autres vues, gérer des vues abstraites et des états.

10.11.2. UI-Bootstrap

Si vous aimez l'esthétique et l'efficacité de Twitter Boostrap, alors le projet UI-Boostrap[9] est pour vous ! Il permet d'utiliser facilement les composants du projet Bootstrap (TimePicker, Accordion, etc…) en les offrant sous formes de directives Angular. Très pratique !

10.11.3. Angular Translate

Si votre application doit être disponible en plusieurs langues, les fonctionnalités par défaut d'Angular en terme d'internationalisation ne vous suffiront probablement pas. Le projet qu'il vous faut, massivement adopté par la communauté est Angular Translate[10]

[8] http://angular-ui.github.io/ui-router/sample/#/
[9] http://angular-ui.github.io/bootstrap/
[10] http://pascalprecht.github.io/angular-translate/

Chapter 11. Le futur

Angular a un futur très brillant, avec une communauté très active, et une équipe de développeurs chez Google très impliquée !

La prochaine version majeure sera la version 2.0 pour fin 2014, ou plus probablement 2015. Elle entend tirer partie des nouveautés de EcmaScript 6. Une grande partie des concepts seront les mêmes, mais la version 2.0 ne sera pas compatible avec les versions actuelles.

Les modules ES6 seront ainsi intégrés au système de module d'Angular, et l'accent sera aussi mis sur le lazy-loading de ceux-ci. Angular 2.0 tirera aussi peut-être profit des annotations pour une meilleure injection de dépendances. Les classes sont également au programme.

Voilà à quoi cela pourrait ressembler (d'après une conférence donnée par l'équipe Angular) :

```
@NgDirective('[ng-src]')
class NgSrcDirective {
  @Inject
  constructor(element:Element) {
    this.element = element;
  }

  @AttrExpressionBinding('ng-src')
  srcChange(value) {
    this.element.src = value;
  }
}
```

La plus grande innovation sera peut-être l'intégration de Object.observe, qui, comme nous l'avons vu, accélérera considérablement la détection des changements dans le modèle et fera gagner beaucoup en performances. Une grande partie du travail portera sur la vitesse du framework.

Le support des applications offlines sera aussi développé, et le mobile sera mis en avant.

Les directives seront retravaillées pour être simplifiées. Elles devraient s'appuyer sur les nouveautés que sont le ShadowDOM, les DOM Mutation Observers et se rapprocher des Web Components.

L’isolation du code sera peut-être de la partie avec l’introduction des zones.

Bref, des perspectives excitantes !

L’effort est également porté pour sortir une version 1.0 de AngularDart, basé sur le langage Dart, développé par Google, et certaines innovations d’AngularJS viennent du projet AngularDart.

Chapter 12. Conclusion

Nous espérons que cette lecture vous aura donner envie de vous lancez ou vous aura éclairci les idées. Ce livre est la façon dont nous aurions aimé que l'on nous explique AngularJS, mais il est sans prétention : si vous trouvez des erreurs, ou avez des idées d'amélioration, nous serions ravis de les entendre : envoyez nous un mail à hello+books@ninja-squad.com[1].

Maintenant à vous de jouer. Et si vous cherchez comment passer à la vitesse supérieure pour vous ou votre équipe, nous donnons une super formation[2] ! ;)

Have fun !

[1] mailto:hello+books@ninja-squad.com
[2] http://ninja-squad.fr/training/angularjs