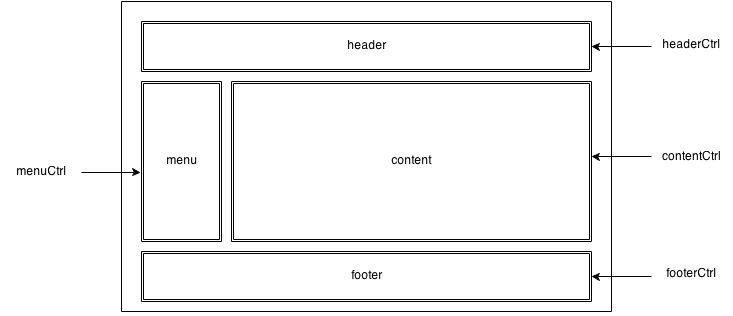
# Le modèle MVC (Modèle-Vue-Contrôleur)

## Définition

En deux mots, il s'agit donc d'un patron de conception qui vous permet d'avoir une stricte séparation entre la vue (ce que l'utilisateur voit), le modèle (les données du système) et le contrôleur (les actions possibles du système).

## Le MVC appliqué à AngularJS

Je ne m'étendrai pas trop sur le MVC, mais je souhaite toutefois vous parler d'une bonne pratique. Il ne s'agit pas de mettre l'ensemble de votre code au sein du même contrôleur. Cela deviendrait très vite anarchique. Il s'agit donc de développer plusieurs "petits" contrôleurs qui vont agir sur différents éléments de la page. Un exemple visuel sera plus parlant :



Comme vous pouvez le constater donc, chaque composant de la page possède son propre contrôleur. Cela possède l'avantage de bien isoler votre code.

Le lien entre la vue HTML et le code JavaScript se fait au moyen de la directive ng-controller. Ainsi, voici le squelette d'une page HTML :

<!doctype html>

<html>

<head>

<title></title>

</head>

<body>

<section ng-controller="headerCtrl">

<h1>HEADER</h1>

</section>

<section ng-controller="menuCtrl">

<h1>MENU</h1>

</section>

<section ng-controller="contentCtrl">

<h1>CONTENT</h1>

</section>

<section ng-controller="footerCtrl">

<h1>FOOTER</h1>

</section>

</body>

</html>

Il ne reste plus qu'à déclarer ces contrôleurs dans la partie JavaScript :

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

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

//...

});

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

//...

});

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

//...

});

app.controller("contentCtrl", function($scope){

//...

});

Nous verrons par la suite que cette façon de faire possède quelques faiblesses. En effet, les parties header, footer et menu ne sont pas censées changer. Par contre, la partie principale (la partie content) sera susceptible de changer. Il pourra par exemple être intéressant de gérer les routes de notre application. De ce fait, nous peuplerons cette partie "content" au moyen de templates. La déclaration des contrôleurs changera donc quelque peu. Évidemment, nous reviendrons sur ces points par la suite, c'est promis !

# La notion de data-binding

## Définition

La plupart des frameworks qui permettent de créer des applications web utilisent cette notion de data-binding. C'est le cas par exemple de Knockout.js (http://knockoutjs.com/), mais c'est également le cas pour AngularJS. Il s'agit d'un moyen de lier la partie vue à la partie logique. En d'autres termes, grâce à cela, les éléments de votre code HTML seront liés à votre contrôleur JavaScript. Formidable, n'est-ce pas? Non, vous n'êtes pas convaincu de l'utilité ? Eh bien lisez la suite alors!

## Utilité du data-binding

Admettons que je souhaite afficher si une personne est majeure en fonction de son âge. Je vous ai préparé un JSFiddle afin d'illustrer le fonctionnement: http://jsfiddle.net/6nwL3r1t/6/ .

<div ng-app="myApp">

<div ng-controller="exemple1Ctrl">

<input ng-model="age"/>

<span>Vous êtes <b ng-bind="majeurOrMineurText()"></b></span>

</div>

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

myApp.controller("exemple1Ctrl", function($scope){

$scope.age = 0;

$scope.majeurOrMineurText = function(){

return ($scope.age >= 18) ? "majeur" : "mineur";

};

});

Ne vous souciez pas de la syntaxe qui peut paraître terrifiante quand on la voit pour la première fois. Intéressez-vous aux deux bindings présents dans cet exemple :

* Le premier au niveau du champ input : vous constatez la présence de l'attribut ng-model dans la balise HTML. En fait, le binding Angular se fait au moyen de ce que l'on appelle des directives. Vous les reconnaissez car elles sont précédées de ng. Bref, dans notre cas, nous disons tout simplement de lier la valeur du champ input à la valeur de la variable age présent dans le code JavaScript. Ne vous souciez pas de $scope, nous y reviendrons par la suite.
* Le deuxième au niveau de la balise <b>. Cette fois, nous utilisons la directive ng-bind à laquelle nous lions une fonction qui retourne la chaîne de caractère "mineur ou majeur" en fonction de l'âge.

Si vous vous amusez à changer la valeur dans le champ input pour y mettre 19 par exemple, vous constaterez que le texte dans la balise <b> changera.

Le data-binding est un concept extrêmement puissant qui va réellement révolutionner votre façon de développer.

Essayons de développer la même fonctionnalité sans utiliser de binding et en s'autorisant à utiliser simplement jQuery. La manière très naturelle de faire serait donc la suivante : <http://jsfiddle.net/atgh5zur/1/>

<div>

<input id="age"/>

<span>Vous êtes <b id="majeurOrMineur"></b></span>

</div>

console.log("test");

$("#age").on("change", function(e){

var age = e.currentTarget.value;

(age > 18) ? $("#majeurOrMineur").text("majeur") : $("#majeurOrMineur").text("mineur");

});

Le problème de cette méthode réside dans le fait que nous devons modifier le DOM (Document Object Model), ce qui est très mauvais lorsque l'on se pose la question de la maintenabilité du code. En effet, imaginez que demain votre code HTML change, que vous décidiez de ne plus utiliser d'id. Votre code ne fonctionnera plus car il faudra également revoir le JavaScript.

Par ailleurs, posez-vous toujours la question du test. Dans mon premier exemple, la spécification de la fonction majeurOrMineur était très simple et j'étais parfaitement en mesure d'écrire un test unitaire adéquat. Dans ce deuxième exemple, nous jouons avec des événements et des modifications de notre vue. Notre logique métier se retrouve enfermée dans un callback, ce qui complexifie énormément le test.

## Un autre exemple pour vous convaincre

AngularJS est un framework qui appartient à la catégorie des frameworks MVVM (Modèle-Vue-VueModèle). En d'autre termes, ces frameworks sont créés sur la base du binding. Fini les modifications du DOM. Tout se passera désormais grâce aux bindings.

Il est très courant d'afficher ou de cacher des éléments sur notre page HTML en réaction aux actions de l'utilisateur.

Prenez exemple sur ce nouveau JSFiddle : <http://jsfiddle.net/7sh8a9oz/>

<div ng-app="app">

<div ng-controller="myController">

<label>Afficher le paragraphe</label>

<input type="checkbox" ng-model="showContent"/>

<p ng-show="showContent">Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.</p>

</div>

</div>

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

app.controller("myController", function($scope){

$scope.showContent = true;

});

Dans ce cas, nous avons créé 2 nouveaux bindings :

* Le premier est lié à la case à cocher. Nous avons lié sa valeur à la variable showContent. Lorsque la case est cochée, la variable contiendra la valeur true et false sinon !
* Le deuxième binding se fait au moyen de la directive ng-show. Cette directive prend un booléen en paramètre et affichera le contenu si ce booléen est vrai, sinon il sera caché.

# Conclusion

Prenez vraiment conscience que le data-binding est un concept clé de ce framework. Croyez-moi, vous verrez sa puissance au fur et à mesure de son utilisation. Le plus difficile, au regard de mon expérience, c'est de changer sa façon de penser. Ne plus agir directement avec le DOM est quelque chose de compliqué à assimiler, surtout au début ! Le data-binding est un peu ce que sont les pointeurs pour le langage C. Ils sont partout et il est important de les comprendre avant de poursuivre.

AngularJS pratique ce que l'on appelle le "two-way data binding". En d'autres termes, il s'agit d'un data-binding bidirectionnel. Ainsi, grâce à cette technique, lorsque vous ferez des actions sur votre page HTML, les variables dans votre JavaScript se mettront à jour et vice-versa : lorsque vous changerez vos variables dans votre JavaScript, les répercussions sur la vue seront immédiates.

# L'injection de dépendances

## Définition

Au même titre que le data-binding, l'injection de dépendances est un concept clé lorsque vous développez avec AngularJS. Je vous rassure, Angular n'est pas le seul framework à utiliser ce système qui est extrêmement important aujourd'hui. Symfony2 exploite également l'injection de dépendances.

Dans la théorie, l'injection de dépendances permet à des modules de ne pas se soucier de l'instanciation des modules dont ils dépendent. Il suffit d'appeler les dépendances et Angular se charge de les instancier et de les injecter pour nous.

## L'exemple de $scop

Si vous reprenez les JSFiddles du précédent chapitre, vous constaterez que nos contrôleurs possédaient tous une dépendance : $scope . C'est notamment grâce à ce module qu'il était possible pour nous de faire du data-binding. Seulement, à aucun moment je n'ai procédé à l'instanciation du module. D'ailleurs, je n'ai aucune idée du comment je dois m'y prendre pour instancier ce module. Cette instanciation est faite par le système d'injection de dépendances géré par Angular. Mes contrôleurs ont juste à demander ce qu'ils veulent.

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

Remarquez que ces modules sont tous précédés par $ . C'est une convention qui nous permet de savoir qu'il s'agit effectivement de modules natifs à Angular. Nous verrons par la suite qu'il est bien évidemment possible de développer nos propres modules. Il sera donc déconseillé de suivre la même convention.

## À quoi ça sert

Il existe de multiples intérêts à pratiquer l'injection de dépendances.

* **La simplicité**. Avouez que c'est très pratique pour un développeur. Vous n'avez plus à vous soucier du comment instancier les modules que vous utilisez. Cela suit le principe du "least knowledge". Lorsque vous développez quelque chose, vous n'avez pas envie de vous soucier des autres composants, vous voulez juste les utiliser !
* **La fiabilité**. Lorsque votre module est chargé, vous avez la certitude que toutes ses dépendances sont chargées et que vous avez la possibilité de les utiliser.
* **La réutilisabilité**. Nous le verrons lorsque nous aborderons la partie des services, mais il s'agit d'un point très important. Lorsque vous développez des services permettant par exemple de faire des conversions de dates, il y a fort à parier que vous souhaiteriez pouvoir réutiliser ce module dans d'autres projets. L'injection de dépendances permet donc d'inciter les développeurs à créer de petits modules unitaires et à les assembler par la suite pour créer des systèmes plus conséquents.
* **Les tests**. C'est un point extrêmement important. Vous commencez à me connaître et vous savez que j'accorde une très grande importance aux tests. Si le module que vous souhaitez tester possède 10 dépendances, il est assez embêtant d'avoir à instancier les 10 modules afin de pouvoir juste tester notre module. À la place, nous allons dire au système d'utiliser des mocks (des bouchons) qui vont se comporter comme nos dépendances.

## Pour finir

L'injection de dépendances constitue un patron de conception assez complexe. Je vous incite à vous documenter sur ce concept. Voici un lien intéressant : http://www.martinfowler.com/articles/injection.html

Je veux juste que vous sachiez qu'Angular exploite largement ce concept. Il n'est pas nécessaire de comprendre précisément le fonctionnement du service d'injection de dépendances, mais sachez que cela existe et que les commodités offertes par le framework ne sont pas magiques.

# $scope et $watch

## $scope

Appuyons-nous sur le code suivant :

<div ng-app="app">

<div ng-controller="exempleCtrl">

HELLO {{name}}!

</div>

</div>

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

app.controller("exempleCtrl", function($scope) {

$scope.name = "World"

});

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

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

Voyez vraiment ce $scope comme le contexte courant dans lequel vous pouvez agir. Chaque contrôleur possédera son propre objet $scope et donc son propre contexte.

Même si cela n'est pas naturel, je souhaite vous en parler. Il est parfois nécessaire d'accéder à d'autres contextes que le sien. En effet, lorsque vous êtes dans votre contrôleur, il est possible que vous ayez besoin d'accéder au contexte d'un autre contrôleur. Si tel est le cas, Angular possède l'objet $rootScope qui permet d'accéder à l'ensemble des contextes présents dans votre page.

## $watch

Il est souvent intéressant d'observer les changements de notre modèle.

Prenons l'exemple suivant. Admettons que nous souhaitions développer un module de réservation. Lorsque ma facture est supérieure à 100€, je souhaite appliquer une promotion de 10%. Voici donc un exemple de solution :

app.controller("BillCtrl", function($scope){

$scope.articles = [{"name": "Téléphone sans-fil", "quantity": 1, "price": "29.99"}, {"name": "Chargeur iPhone5", "quantity": 1, "price": "29.99"}];

$scope.total = function(){

var total = 0;

for(var i = 0; i < $scope.articles.length; i++){

total += $scope.articles[i].price \* $scope.articles[i].quantity;

}

return total;

};

function calculateDiscount(newValue, oldValue, scope){

$scope.discount = (newValue > 100) ? newValue \* 0.10 : 0;

};

$scope.finalTotal = function(){

return $scope.total() - $scope.discount;

};

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

});

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

La spécification de la fonction est la suivante :

$watch(watchFn, watchAction, deepWatch)

Analysons les paramètres :

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

Voyez le résultat de l'utilisation de $watch en action ici : <http://jsfiddle.net/hwzfm6nL/>

L'utilisation de $watch est très puissante et je vous encourage vivement à l'utiliser, mais attention aux performances qui peuvent parfois se dégrader. Dans la partie 4 de ce cours, je vous propose l'utilisation d'une extension Chrome qui vous permettra de debugger efficacement votre code Angular. Les questions de performance seront au rendez-vous.

Pour plus de détails, je vous invite à consulter la documentation officielle très bien faite à ce niveau-là : <https://code.angularjs.org/1.2.26/docs/api/ng/type/$rootScope.Scope>

# Les directives

## Définition

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

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

Angular fournit un certain nombre de directives. Rappelons-en quelques-unes :

* ng-Controller : directive permettant d'attacher un contrôleur à la vue
* ng-Repeat : directive permettant de répéter un template pour chaque élément d'une collection.
* ng-Model : directive permettant de lier les input, textarea ou select à une propriété du contexte actuel.

ngApp est une autre directive dont nous n'avons pas parlé mais qui est extrêmement importante. Cette directive permet tout simplement d'initialiser votre application. Placez-la au niveau de votre balise body ou html, et passez-lui le nom de votre application.

Dans l'exemple suivant, je crée un module myApp. Tous les modules (contrôleurs, services, directives) seront OBLIGATOIREMENT rattachés à ce module.

<html>

<head>

</head>

<body ng-app="myApp">

</body>

</html>

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

## Créer une directive

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

### Le nommage

Cela peut sembler dérisoire, mais ça ne l'est pas. Vous avez sans doute remarqué que les directives natives à Angular étaient TOUTES précédées de ng. Ce ng constitue l'espace de nom Angular ; par conséquent, évitez de l'utiliser lorsque vous créez vos propres directives.

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

Lorsque vous souhaiterez appeler votre directive dans votre vue, utilisez la syntaxe suivante : namespaceDirectiveName

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

### Spécification

Voyons un pseudo-code avec TOUTES les options disponibles afin de créer une directive :

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

myModule.directive('namespaceDirectiveName', function factory(injectables) {

var directiveDefinitionObject = {

restrict: string,

priority: number,

template: string,

templateUrl: string,

replace: bool,

transclude: bool,

scope: bool or object,

controller: function controllerConstructor($scope,

$element,

$attrs,

$transclude),

require: string,

link: function postLink(scope, iElement, iAttrs) { ... },

compile: function compile(tElement, tAttrs, transclude) {

return {

pre: function preLink(scope, iElement, iAttrs, controller) { ... },

post: function postLink(scope, iElement, iAttrs, controller) { ... }

}

}

};

return directiveDefinitionObject;

});

### Les fonctions compile et link

Il s'agit des fonctions qui vont être utilisées dans le processus de manipulation du DOM, ajout de listeners...

Afin de comprendre le fonctionnement de ces deux fonctions, étudions de plus près le processus d'initialisation utilisé par Angular. Ce processus se fait en 3 étapes :

1. Chargement : Angular charge et recherche la directive ng-app.
2. Compilation : au cours de cette phase, Angular parcourt le DOM afin d'identifier l'ensemble des directives. Il s'occupe notamment des manipulations du DOM qui ne requièrent pas l'utilisation du contexte.
3. Mise en place des liens : il s'agit de la dernière phase. C'est notamment au cours de celle-ci qu'Angular se charge de lier un contexte à la directive.

Vous avez donc accès au contexte seulement dans les fonctions link et non dans les fonctions compile.

Les fonctions compile ne sont appelées qu'une seule fois alors que les fonctions link sont appelées plusieurs fois, une fois pour chaque instance de la directive.

Dois-je mettre mon code dans la fonction compile ou link ?

Pour répondre à cette question, posez-vous les questions suivantes :

* Ai-je besoin d'utiliser le contexte ? Si oui, utilisez la fonction link. Sinon, utilisez la fonction compile.
* Faites-vous des manipulations du DOM ? Si oui et que vous n'avez pas besoin du contexte, utilisez compile, sinon, utilisez link.

Faire le bon choix peut s'avérer important au niveau de la performance, car rappelez-vous que la fonction compile est appelée une seule fois alors que la fonction link est appelée n fois.

## Un exemple concret

Je conçois que cette partie soit complexe mais elle est centrale lorsque l'on développe avec AngularJS. Je vous propose donc d'illustrer cette partie théorique à l'aide d'un exemple concret. Supposons que nous souhaitions développer une directive qui va nous permettre de créer un menu vertical. Voici le comportement que nous voulons :



menu fermé

Lorsqu'on clique sur le titre, le menu devra s'ouvrir comme cela :



menu ouvert

<div ng-app="app">

<div ng-controller="SomeController">

<expander class="expander" expander-title="title">

{{text}}

</expander>

</div>

</div>

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

app.controller("SomeController", function($scope){

$scope.title = "Titre";

$scope.text = "Contenu";

});

app.directive("expander", function(){

return {

restrict: 'EA',

replace: true,

transclude: true,

scope: {title: '=expanderTitle'},

template: '<div>' +

'<div class="title" ng-click="toggle()">{{title}}</div>' +

'<div class="body" ng-show="showMe" ng-transclude></div>' +

'</div>',

link: function(scope, element, attrs){

scope.showMe = false;

scope.toggle = function toggle(){

scope.showMe = !scope.showMe;

};

}

};

});

Vous pouvez vérifier le résultat ici : http://jsfiddle.net/qks8p28g/

Apportons quelques commentaires sur ce code :

Remarquez que nous avons enrichi notre DOM de la balise expander. Cela est possible puisque nous avons défini l'option restrict avec les paramètres 'EA' (élément et attribut). Autrement dit, il nous est possible de déclarer notre directive directement au sein d'un nouvel element ou bien au sein d'un attribut que nous aurions pu mettre dans un élément div par exemple.

Les options replace, transclude et template fonctionnent ensemble. Cela indique à Angular que nous souhaitons remplacer la déclaration de notre directive expander par notre template et que le contenu de notre balise expander (à savoir {{text}}) se trouvera à l'endroit où nous avons déclaré notre ng-transclude.

La fonction link permet tout simplement de créer notre modèle qui comprend notamment la fonction permettant d'ouvrir et de fermer notre onglet.

Notez enfin que nous choisissons d'avoir un contexte isolé puisque nous redéfinissons la propriété scope. Nous souhaitons tout de même conserver l'attribut title issue de notre contexte parent (à savoir le contexte du contrôleur).