

# **FORMATION ANGULARJS**

Initiation au framework Javascript AngularJS



# FORMATEUR

Jocelyn N'TAKPE

Toulousain, 28 ans

Architecte JavaEE/JS, Responsable archi BU Banque&Finance

Formateur AngularJS

# TOUR DE TABLE

Vos noms et prénoms

Votre parcours

Votre agence et site

Vos expériences en développement et JavaScript

Vos attentes

# JAVASCRIPT

Ce n'est pas du Java !!! (ni du Java en script  
)

- Langage objet mais sans classes
- Faiblement typé
- Fonctionnel

# ECMAScript

Actuellement EcmaScript 5.1

- mode strict
- JSON

Bientôt EcmaScript 6

- classes
- modules
- promises
- arrow functions

# MODE STRICT

Toujours utiliser le mode strict

En début de fichier ou en début de fonction.

Permet de signaler des erreurs courantes de immédiatement  
(fail-fast) :

- variable non déclarée
- unicité des propriétés dans les littéraux
- unicité des noms d'arguments
- etc ...

# SEMICOLON

JavaScript insère tout seul les semicolons  
manquants (ou pas)

```
function hum() {  
  var name, age  
  name = 'joss'  
  age = 28  
  return  
    {  
      name: name,  
      age: age  
    };  
}
```





# PORTÉE

En JavaScript la portée est au niveau de la fonction

Contrairement à la syntaxe C, la portée n'est pas limitée aux accolades.

Ce comportement sera évitable en ES6 avec le mot clé *let*.

Toute variable ou fonction non déclarée dans une fonction sera stockée dans le scope global

# TYPES

Il existe des objets et types primitifs en JS

Primitifs:

- boolean
- number
- string
- undefined
- null

Pas de type 'char' et un seul type numérique (décimal sur 8 octets  
= pas précis)

# AUTOBOXING

Les primitifs sont convertis à la volée si c'est nécessaire

```
var n=5;  
n.toString();
```

```
var s='hello';  
s.length;
```

Evitez de créer (comme en Java) des objets  
Boolean, Number et String

```
var a = new String('a');  
a == 'a'; //true  
a === 'a'; //false
```

Préférez la comparaison stricte (===)

# OBJETS

Dans les langages objets classiques, les objets sont des instances de classes.

En JavaScript, un objet est une map donc un ensemble key/value

Il n'y a pas de notion de visibilité.

Pour accéder à une propriété :

```
obj[ 'mykey' ];
```

ou

```
obj.mykey;
```

# LITERAL

Pour créer un objet en JavaScript :

```
var emptyObj = {};  
var user = {  
  name: 'joss',  
  age: 28,  
  adress: {  
    street: 'Jean Jaurès',  
    city: 'Toulouse'  
  }  
};
```

Pour modifier une propriété :

```
user.name = 'Jocelyn';  
user['age'] = 29;
```

Pour supprimer une propriété :

```
delete user.name;
```

# PROTOTYPE

Manière de faire de l'héritage en JS (très puissant mais compliqué).

Pour récupérer une valeur :

- on cherche pas l'objet courant
- puis dans son prototype
- puis dans le prototype du prototype
- ...jusqu'à Object.prototype

Pas toujours vrai pour setter une valeur

Pour créer un objet avec un prototype :

```
var obj = Object.prototype(base);  
console.log(this.baseFct()); //execute function called baseFct
```

# ARRAY

Pour créer un tableau :

```
var tab = ['a', 1, true];  
tab[0]; // 'a'  
tab['2']; // true
```

Les tableaux sont des objets JS

La longueur d'un tableau JS est modifiable :

```
tab.length; //3  
tab[20] = 'wtf';  
tab.length; //21  
tab.length = 2;  
tab // ['a', 1]  
tab.length = 0;  
tab // []
```

# MÉTHODES DES TABLEAUX

- `mytab.push(elem1, elem2, ...)` : ajoute des éléments à la fin du tableau
- `mytab.pop()` : enlève et renvoie le dernier élément du tableau
- `mytab.shift()` : enlève et renvoie le premier élément du tableau
- `mytab.unshift(elem1, elem2, ...)` : ajoute des éléments au début du tableau
- `mytab.splice(index, howMany, elem1, elem2, ...)` : supprimer et/ou ajoute des éléments
- `mytab.concat(tab1, tab2, ...)` : renvoie un nouveau tableau correspondant au tableau courant concaténé avec ceux passés en paramètres
- `mytab.slice(begin [,end])` : renvoie un nouveau tableau correspondant à une sous-partie du tableau



# MÉTHODES DES TABLEAUX - SUITE

- `mytab.indexOf(searchElem [, fromIndex])` : renvoie le premier élément trouvé
- `mytab.lastIndexOf(searchElem [, fromIndex])` : renvoie le dernier élément trouvé
- `mytab.forEach(callback [, thisArg])` : applique la fonction passée en paramètre sous tous les éléments du tableau
- `mytab.filter(callback [, thisArg])` : filtre en utilisant la fonction passée en paramètre
- ...

# FUNCTION

Les fonctions sont des objets JS

On peut :

- mettre une fonction dans une variable
- mettre une fonction dans une propriété d'objet
- mettre une fonction dans un tableau
- passer une fonction en paramètre d'une autre fonction
- renvoyer une fonction

Il n'y a de contrôle sur le type et le nombre de paramètre lors de l'appel d'une fonction

# OUTILS

- Votre IDE préféré
- nodejs package manager (npm)
- Git
- Grunt
- Gulp
- Yeoman
- Bower
- JsLint
- BrowserSync

**INTELLIJ**

**(HTTPS://WWW.J  
ETBRAINS.COM/I  
DEA/)**

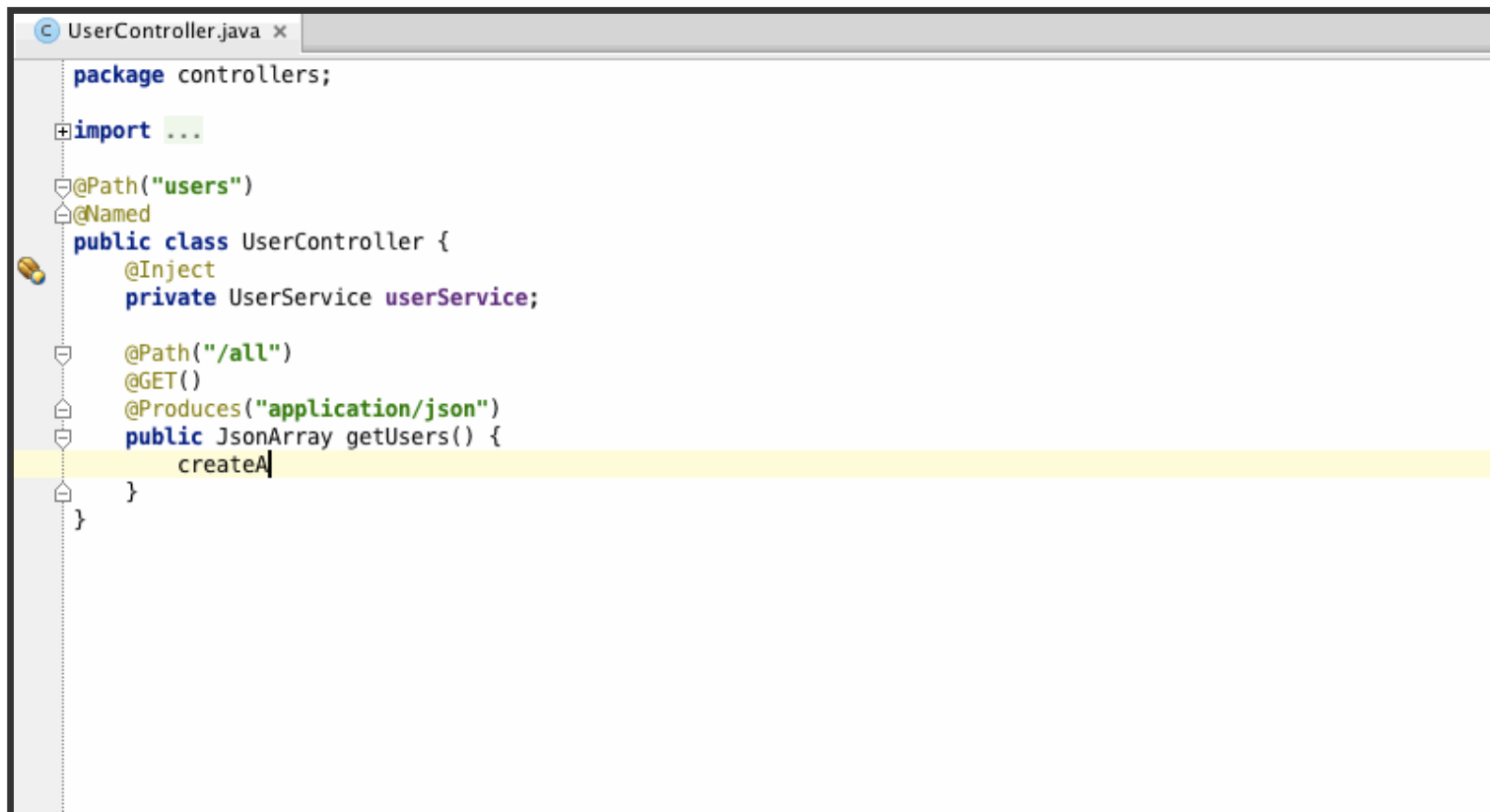
**WEBSTORM**

**(HTTPS://WWW.J**

# INTELIJ

Développé par JetBrains depuis 2001

Versions community (open-source) et payante (ultimate)



```
UserController.java x
package controllers;

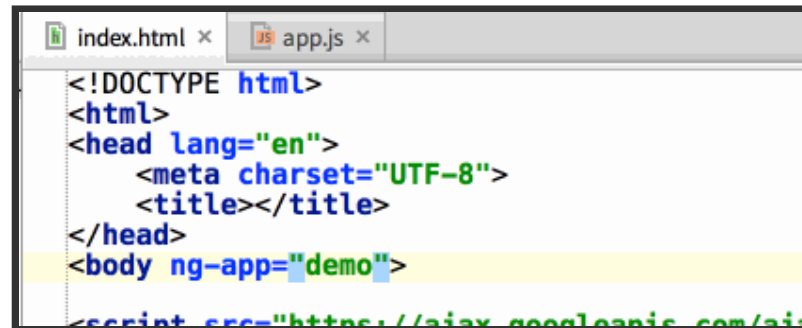
import ...

@Path("users")
@Named
public class UserController {
    @Inject
    private UserService userService;

    @Path("/all")
    @GET()
    @Produces("application/json")
    public JSONArray getUsers() {
        createA
    }
}
```

# SUPPORT ANGULARJS

Large support de l'écosystème JavaScript dont AngularJS



```
index.html x app.js x
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title></title>
</head>
<body ng-app="demo">
<script src="https://ajax.googleapis.com/ajax"
```



```
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title></title>
</head>
<body ng-app="demo" ng-controller="AppCtrl as app">
|
<script src="https://ajax.googleapis.com/ajax/libs/a
<script src="app.js"></script>
```

# GRUNT ([HTTP://GRUNTJS .COM/](http://gruntjs.com/))

*The JavaScript task runner*

# GRUNT

*Grunt* est un lanceur de tâches. Il permet comme Maven l'automatisation des tâches répétitives comme :

- la compilation
- la minification des ressources statiques
- la concaténation des ressources statiques
- l'exécution des tests unitaires

*Grunt* utilise 2 fichiers de configurations :

- **package.json** : définissant les plugins et metadata (il s'agit bien entendu du fichier de configuration de NPM)
- **gruntfile.js** : déclarant la configuration des tâches.



# GRUNT

## PACKAGE.JSON

```
{  
  "name": "Nom_du_project",  
  "version": "1.0.0",  
  "devDependencies": {  
    "grunt": "latest",  
    "grunt-bower-task": "latest",  
    "grunt-contrib-watch": "latest",  
    "grunt-contrib-cssmin": "latest",  
    "grunt-contrib-uglify": "latest"  
  }  
}
```

# GRUNT

## GRUNTFILE.JS

```
module.exports = function (grunt) {  
  grunt.initConfig({  
    pkg: grunt.file.readJSON('package.json'),  
    concat: {  
      libs: {  
        src: [  
          'js/jquery/dist/jquery.min.js',  
          'js/angular/angular.min.js'  
        ],  
        dest: 'target/js-libs.min.js'  
      }  
    }  
  });  
  grunt.loadNpmTasks('grunt-contrib-concat');  
  grunt.registerTask('default', ['concat:libs']);  
}
```

# **GULP** **(HTTP://GULPJS.** **COM/)**

*The streaming build system*

# GULP

*Gulp* est un lanceur de tâches concurrent de *Grunt*. Contrairement à *Grunt*, *Gulp* utilise les streams NodeJS réduisant ainsi les écritures et lecture disque. Les fichiers de configuration *Gulp* sont également moins verbeux.

*Gulp* comme *Grunt* utilise 2 fichiers de configurations :

- **package.json** : définissant les plugins et metadata (il s'agit bien entendu du fichier de configuration de NPM)
- **gulpfile.js** : déclarant la configuration des tâches.

# GULP

## PACKAGE.JSON

```
{  
  "name": "Nom_du_projet",  
  "version": "1.0.0",  
  "private": true,  
  "devDependencies": {  
    "gulp": "latest",  
    "gulp-minify-css": "latest",  
    "gulp-uglify": "latest",  
    "gulp-flatten": "latest"  
  }  
}
```

# GULP

## GULPFILE.JS

```
var gulp = require('gulp'),
    flatten = require('gulp-flatten'),
    dirs = {
      bower: 'src/main/webapp/static/bower_components',
      dest: 'src/main/webapp/dist',
    };

gulp.task('copy:fonts', function () {
  gulp.src(dirs.bower + '/*.{ttf,woff,eof}')
    .pipe(flatten())
    .pipe(gulp.dest(dirs.dest + '/fonts'));
});

gulp.task('default', function () {
  runSequence('copy:fonts');
});
```

# **BOWER**

# **(HTTP://BOWER.I**

# **0/)**

*A package manager for the web*

# BOWER

*Bower* est un gestionnaire de dépendances client.

*Bower* est configuré à l'aide de 2 fichiers :

- **.bowerrc** permet de renseigner le répertoire de destination, le proxy, etc...
- **bower.json** permet de gérer les dépendances



# BOWER

## .BOWERRC

```
{
  "directory": "src/main/webapp/static/bower_components",
  "analytics": false,
  "timeout": 120000,
  "registry": {
    "search": [
      "http://localhost:8000",
      "https://bower.herokuapp.com"
    ]
  }
}
```

# BOWER

## BOWER.JSON

```
{
  "name": "Nom_du_projet",
  "version": "1.0.0",
  "dependencies": {
    "jquery": "latest",
    "bootstrap": "latest",
    "angular": "latest",
    "angular-messages": "latest",
    "angular-route": "latest",
    "angular-resource": "latest",
    "angular-bootstrap": "latest",
  },
  "resolutions": {
    "jquery": "2.1.1",
    "angular": "1.3.7"
  }
}
```

# **YEOMAN** **(HTTP://YEOMAN.** **IO/)**

*The web's scaffolding tool for modern webapps*

# YEOMAN

*Yeoman* est un outil permettant de générer un squelette de webapp en utilisant les conventions (comme les artifacts Maven).

*Yeoman* utilise Bower et Grunt lors de la génération du squelette de l'application.

Un ensemble de générateurs de webapps sont disponibles (<http://yeoman.io/generators/>).

# YEOMAN

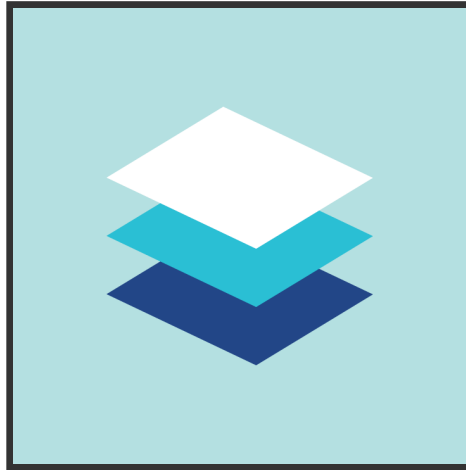
## GÉNÉRATION D'UN SQUELETTE D'APPLICATION

```
npm install -g generator-gulp-angular  
yo gulp-angular Mon_Projet
```

# UI FRAMEWORKS



Bootstrap,  
développé par Twitter, projet open-  
source le plus populaire



Material design,  
spécification de Google notamment  
utilisé sur lollipop



Foundation,  
dans l'ombre de Bootstrap

# CSS PREPROCS



**SASS,**

extension du CSS, implémentation en  
Ruby et C++



**LESS,**

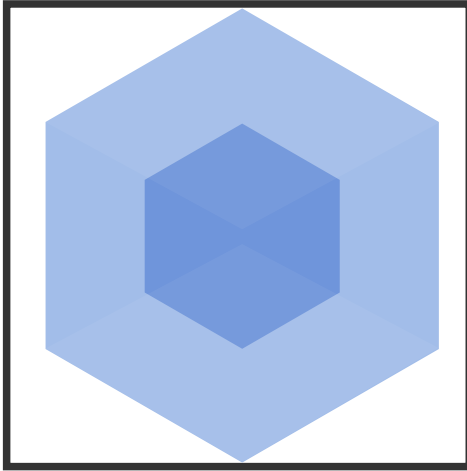
extension du CSS concurrente de  
SASS utilisé par Bootstrap,  
implémentation en NodeJS



**Stylus,**

même fonctionnalités que SASS en  
NodeJS et sans accolades

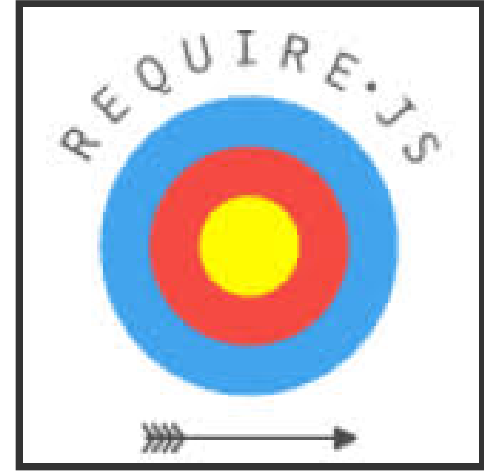
# COMMON JS



Webpack,  
modules NodeJS pour le navigateur,  
très rapide



Browserify,  
ancêtre de Webpack



RequireJS,  
première implémentation, verbeuse



# WEB FMK



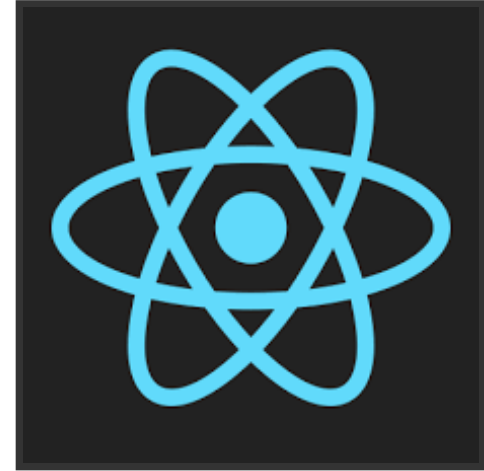
BackboneJS,

le plus ancien laissant beaucoup de  
liberté mais demandant beaucoup de  
code



EmberJS,

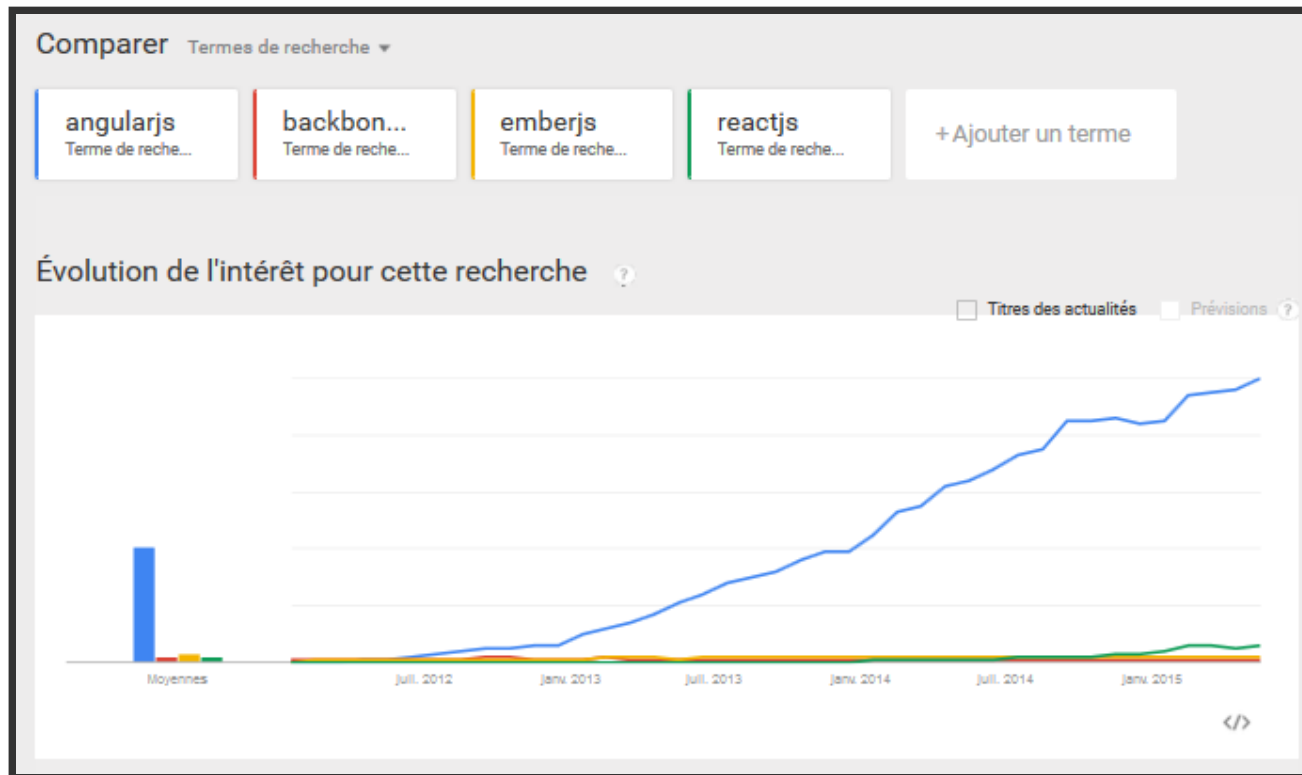
qui a conquis la communauté Ruby



ReactJS,

le plus récent, développé par  
Facebook

# POPULARITÉ



# ANGULARJS

Open source (MIT)

Créé en 2009 par Google et la communauté

Framework (≠ librairie)

Firefox, Chrome, IE  $\geq 9$  (depuis 1.3), Safari, Opera, IOS, Android  
browser

Courbe d'apprentissage pas toujours linéaire (<img/feelings-angular.png>)

# VISION ANGULARJS DU WEB

- Peu de 'boilerplate'
- Découplage entre le DOM et la logique
- Utilisation de composants
- Découplage client/serveur
- Automatisation des tests

# PRINCIPES

- Binding bi-directionnel
- Pattern MVW
- Modularité
- Injection de dépendances
- Routage
- Validation

# SYNTAXE

## Déclarative (HTML) pour la construction de l'interface

```
<table ng-controller="StatsController">
  <tr ng-repeat="stat in stats">
    <td>{{stat.class}}</td>
  </tr>
</table>
```

## Impérative (JS) pour la logique de l'application

```
myApp.controller('StatsController', function ($scope, application, StatsService) {
  StatsService.findAll(application.id)
  .success(function (data) { $scope.stats = data; })
  .error(function(data, status) { //Some error handling code });
});
```

# MVW

- Model : purs objets JavaScript (POJO) contenant les données
- View : templates HTML accédant aux données exposées
- Controller : fonction JS exposant le modèle aux vues via l'objet \$scope
- Directive : concept spécifique Angular permettant d'ajouter du comportement aux applications

# EXPRESSIONS

Angular interprète les templates, ce qui permet d'écrire du HTML contenant des expressions dynamiques (!= eval()).

```
<h1>{{ 1+1 }}</h1> // 2  
<h1>{{ "Hello " + "world !!!" }}</h1> // Hello world
```

Les expressions pardonnent les null ou undefined (une chaîne de caractères vide sera affichée).



# RÈGLES

- Les variables des expressions sont évaluées depuis l'objet scope et non l'objet window.
- Pour respecter la séparation du code entre les vues et les contrôleurs, il est impératif de garder les expressions simples. Il est préférable de faire appel à une fonction exposée par un contrôleur.
- Il est possible d'utiliser des filtres dans les expressions

# BINDING

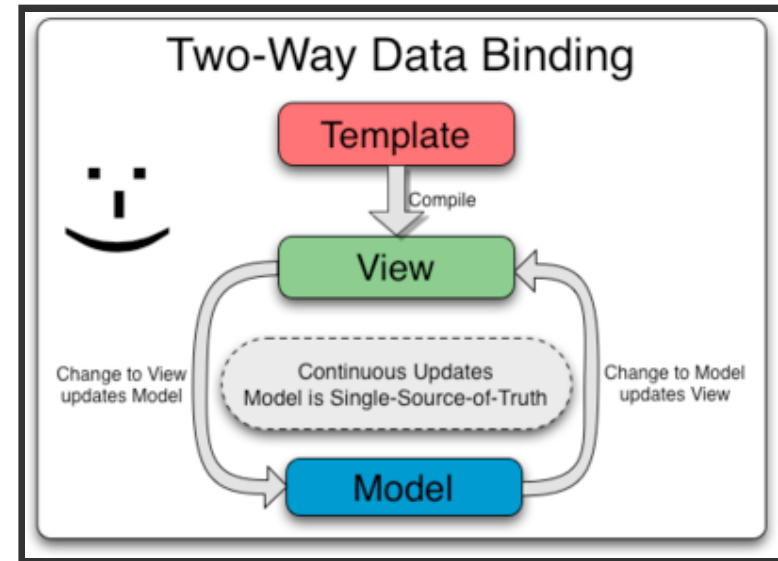
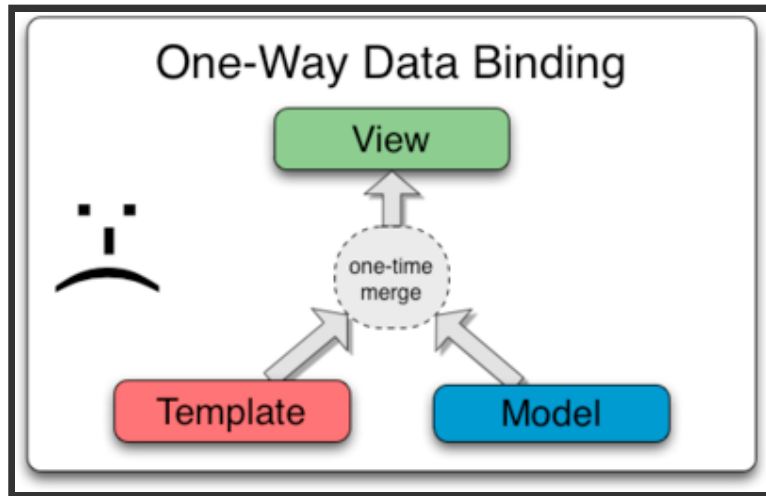
Le double data-binding d'Angular permet de maintenir le modèle en accord avec la vue et vice-versa.

```
<input type="text" ng-model="hello"/>  
<h1>{{ hello }} world !!!</h1>
```

En préférant la 'dot notation' avec l'expression évaluée dans la classe

```
<input type="text" ng-model="hello.myclass"/>  
<h1 class="{{ hello.myclass }}">Hello world !!!</h1>
```

# DOUBLE DATA BINDING



# CONTROLLER

Les contrôleurs sont chargés de manipuler le modèle et de faire lien avec la vue (page HTML).

Il est possible de définir plusieurs contrôleurs sur un template.

AngularJS gère le cycle de vie des contrôleurs.

# SCOPE

```
<input type="text" ng-model="hello"/>  
<h1>{{ hello }} world !!!</h1>
```

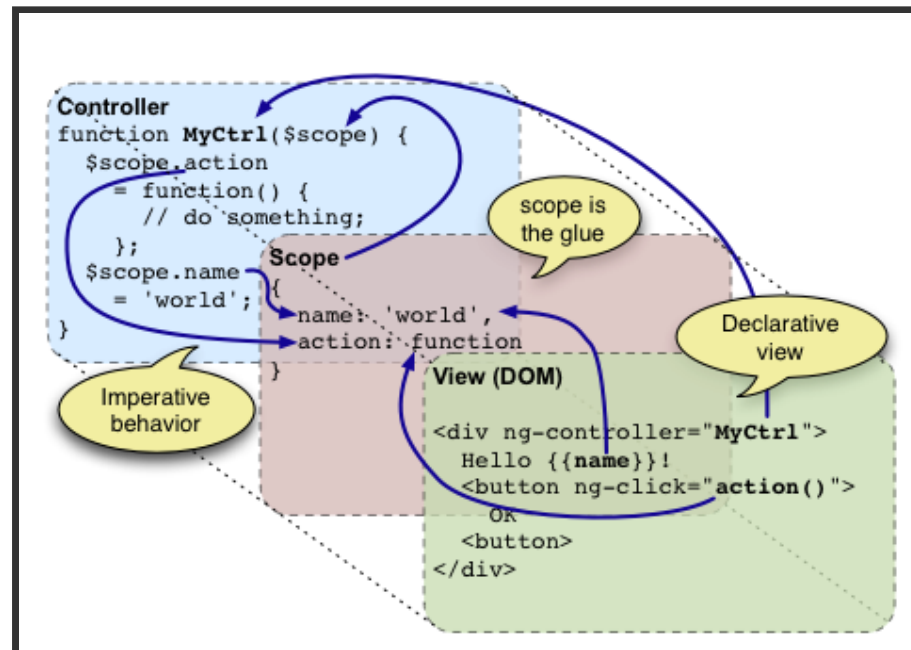
Comment est évaluée l'expression {{ hello }} ?

AngularJS va chercher un attribut nommé 'hello' dans l'objet scope, si celui-ci est trouvé alors la valeur de l'attribut sera affichée sinon une chaîne de caractères vide.

Le scope est ainsi utilisé pour initialiser des valeurs ou récupérer les valeurs modifiées par un utilisateur (cf double data binding).

# SCOPE

- Le scope est lien entre le contrôleur et la vue.
- Chaque contrôleur créé aura son propre scope.



# TP TODO LIST

Exemple controller :

```
git checkout ex-controller
```

Début TP :

```
git checkout start-tp-controller
```

Étapes :

- Init application et contrôleur
- Init variables : tâches restantes
- Init de la liste de tâches et affichage
- Switch tâche finie/restante
- Suppression d'une tâche
- Ajout d'une tâche
- Sélection de toutes les tâches
- Comptage des tâches restantes (*plus tard*)

# HÉRITAGE DES SCOPES

- Lors de démarrage de l'application un objet `$rootScope` est créé, il sera le parent de tous les scopes.
- AngularJS comme JavaScript utilise l'héritage prototypal (<https://github.com/angular/angular.js/wiki/Understanding-Scopes>) pour les scopes.

Ayez toujours un `'.'` dans vos `ngModel`.



# CONTROLLER AS SYNTAX

- Plus de proche de la logique objet JavaScript
- Fournit la notation avec un "." directement
- Permet de nommer explicitement dans les vues le contrôleur contenant une variable ou fonction

Attention à l'utilisation de this

## HTML

```
<div ng-controller="CustomerCtrl as customer">
  {{ customer.name }}
</div>
```

## JS

```
function CustomerCtrl() {
  var vm = this;
  vm.name = {};
}
```

# FILTRES

Les filtres permettent de transformer les valeurs d'affichage, de filtrer ou réordonner les éléments d'une collection.

La syntaxe UNIX est utilisée :

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

On peut chaîner les filtres :

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

Passer des paramètres :

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

Ou encore accéder au filtre en JS :

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

# FILTRES NATIFS

- **number** : permet de préciser le nombre de chiffres après la virgule

```
{{ 31.26 | number:1 }} // 31.3  
{{ 31.26 | number:3 }} // 31.260
```

- **currency**: permet de préciser la monnaie

```
{{ 31.26 | currency:'$' }} // $31.26
```

- **date** : permet de formater la date en passant le pattern

```
{{ today | date:'dd/MM/yyyy' }} // 21/05/2015
```

- **lowercase/uppercase** : convertit en minuscules ou majuscules

```
{{ "Jocelyn" | uppercase }} // JOCELYN  
{{ "Jocelyn" | lowercase }} // jocelyn
```

- **json** : affiche l'objet au format JSON

```
{{ user | json }} // { name: 'Jocelyn', company: 'Sopra Steria' }
```

# FILTRAGE DE TABLEAUX

- `limitTo` : permet de filtrer un sous-ensemble

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

- `orderBy` : permet de trier les éléments

Par exemple :

```
var pierre = {name: 'Pierre', gender: 'male'};  
var paul = {name: 'Paul', gender: 'male'};  
var marie = {name: 'Marie', gender: 'female'};  
var julie = {name: 'Julie', gender: 'female'};  
$scope.persons = [pierre, paul, marie, julie];
```

Donne :

```
{{ persons | orderBy: 'name'}} // Julie, Marie, Paul, Pierre  
{{ persons | orderBy: ['gender', '-name']}}// Marie, Julie, Pierre, Paul
```

# FILTRAGE DYNAMIQUE

- Agit sur un tableau pour en retourner un sous-ensemble
- Retient tout élément du tableau ayant une propriété contenant la chaîne de caractères recherchée

```
<input type="text" ng-model="mysearch"/>  
<tr ng-repeat="person in persons | filter:mysearch">
```

Ou sur un propriété spécifique :

```
<input type="text" ng-model="mysearch.name"/>  
<tr ng-repeat="person in persons | filter:mysearch">
```

# FILTRES PERSONNALISÉS

- Déclaration au niveau du module en utilisant le mot clé 'filter'
- Une fonction de filtrage renvoie une autre fonction qui prend en paramètre l'élément à filtrer

angular

```
.module('filterapp', [])  
.controller('FilterCtrl', function ($scope) {  
    $scope.name = "";  
})  
.filter('reverse', function () {  
    return function (text) {  
        return text.split("").reverse().join("");  
    };  
});
```

# TP TODO FILTRAGE

Début TP :

```
git checkout start-filter-tp
```

- Afficher les tâches en majuscules
- Compter les tâches restantes
- Filtrer via les boutons les tâches actives, complètes ou ne pas appliquer de filtre
- Facultatif : Utiliser la syntaxe Controller as

# MODULES

- AngularJS utilise une logique modulaire pour déclarer les différents composants de l'application
- Le chargement des modules peut être fait dans n'importe quel ordre
- Les librairies externes sont souvent récupérées sous forme de module
- Permet d'avoir des fichiers séparés par responsabilité (un fichier / service, controller, module)

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

Même si un module n'a pas de dépendance, il ne faut pas omettre les crochets



# DÉCLARATION DE MODULES

```
var accountModule = angular.module('myapp-account', []);  
accountModule.controller('accountCtrl', function () {});  
accountModule.service('accountService', function () {});  
var myapp = angular.module('myapp', ['myapp-account']);
```

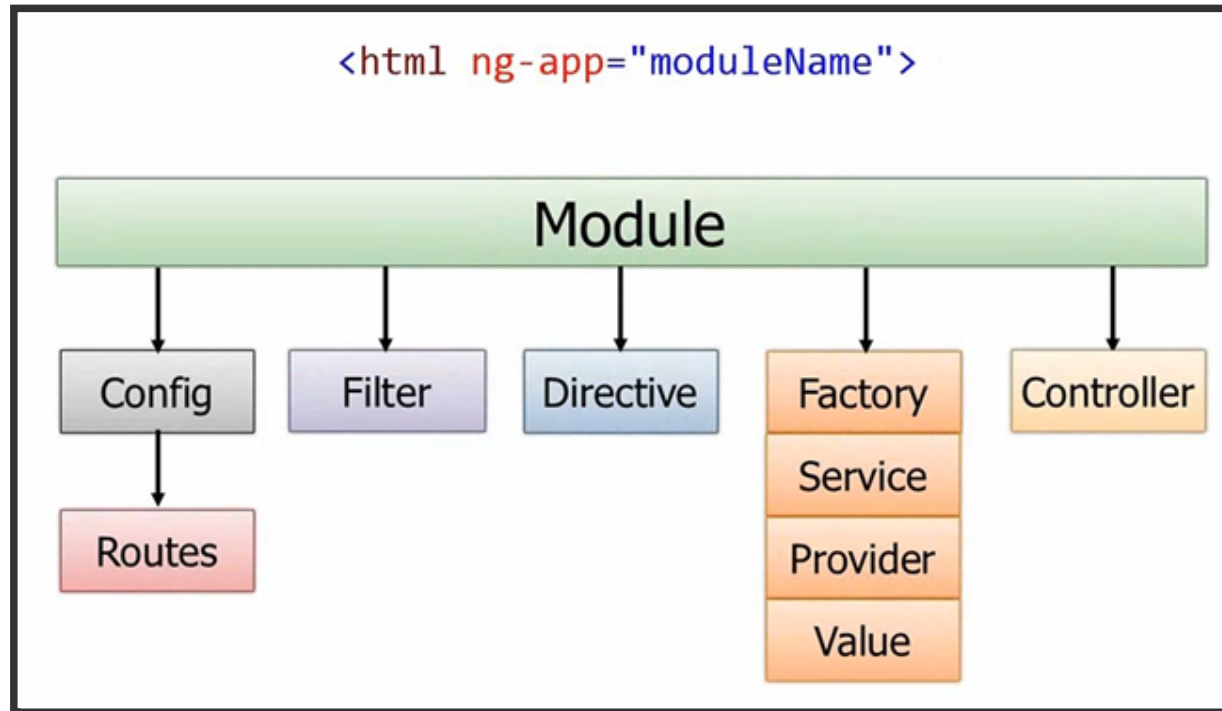
# MODULES FACULTATIFS

- *ngCookies*, pour la gestion des cookies
- *ngTouch*, pour les événement liés au mobile
- *ngRoute*, pour le routage
- *ngResource*, pour interagir avec des ressources REST
- *ngMessages*, pour l'affichage de messages notamment pour les champs de formulaires
- *ngAnimate*, pour les animations
- *ngSanitize*, pour vérifier le HTML

# AUTRES MODULES ESSENTIELS

- *AngularUI Bootstrap*, adaptation du module JS de Bootstrap en directives Angular
- *Angular Material*, implémentation de la spécification Material Design de Google en Angular.
- *AngularUI Router*, module de gestion de routes beaucoup plus puissant que le module officiel (ngRoute)
- *Angular Translate*, permet une gestion plus poussée de l'i18n.

# MOTS CLÉS



# ORGANISATION

- css/
- img/
- js/
  - app.js
  - controllers.js
  - directives.js
  - filters.js
  - services.js
- lib/
- partials/

Basique

- controllers/
  - LoginController.js
  - RegistrationController.js
  - ProductDetailController.js
  - SearchResultsController.js
- directives.js
- filters.js
- models/
  - CartModel.js
  - ProductModel.js
  - SearchResultsModel.js
  - UserModel.js
- services/
  - CartService.js
  - UserService.js
  - ProductService.js

Technique

- cart/
  - CartModel.js
  - CartService.js
- common/
  - directives.js
  - filters.js
- product/
  - search/
    - SearchResultsController.js
    - SearchResultsModel.js
  - ProductDetailController.js
  - ProductModel.js
  - ProductService.js
- user/
  - LoginController.js
  - RegistrationController.js
  - UserModel.js
  - UserService.js

Fonctionnel

# ROUTES

AngularJS permet de créer des single page applications

Même si l'application ne contient qu'une page, il est possible de recharger une partie du contenu en utilisant le mécanisme de routage d'Angular.

Angular propose un module de routage nommé ngRoute.  
Néanmoins, en entreprise on lui préfère le module uiRouter.

# STATE VS ROUTE

Un état de votre application

Hiérarchie imbriquable

Un état correspond à un nom

Navigation entre états par noms  
ou urls

Plusieurs views dans un  
template

Une URL de votre application

Hiérarchie plate

Un état correspond à une url

Navigation par url

Une seule view dans un  
template

# SIMILITUDES

Etat:

```
$stateProvider.state('contact.detail', {  
  url: '/contacts/:id',  
  templateUrl: 'contacts.html',  
  controller: 'ContactsCtrl as contacts',  
  resolve: {...}  
});
```

Route:

```
$routeProvider.when('/contacts/:id', {  
  templateUrl: 'contacts.html',  
  controller: 'ContactsCtrl as contacts',  
  resolve: {...}  
});
```



# DÉMARRAGE

Ajouter le script :

```
<script src="angular-ui-router.js"/>
```

Charger le module en tant que dépendance :

```
angular.module('myApp', ['ui.router']);
```

Déclarer un état :

```
angular.module('myApp').config(function($stateProvider) {  
    $stateProvider.state('home', {  
        url: '/',  
        templateUrl: 'home.html'  
    });  
});
```

# ACTIVER UN ÉTAT

Naviguer vers l'URL correspondant à l'état

ou

```
$state.go('mystate');
```

ou

```
<a ui-sref="mystate">Lien</a>
```

# RESOLVE

Permet de résoudre des dépendances et de les injecter dans un contrôleur. Si resolve est une promesse, elle sera résolue avant l'appel du contrôleur.

```
$stateProvider.state('contact', {  
  resolve: {  
    username: function($http) {  
      return $http.get('/current/user');  
    },  
    simple: function() {  
      return 'coucou';  
    }  
  },  
  controller: function ($scope, username, simple) {  
    $scope.currentUser = username;  
    $scope.msg = simple;  
  }  
});
```

# HIÉRARCHIE

Permet d'imbriquer les états les uns dans les autres.

```
$stateProvider
  .state('home', {...})
  .state('contacts', {...})
  .state('contacts.detail', {...})
  .state('contacts.detail.edit', {...});
```

Alternative :

```
$stateProvider
  .state('home', {...})
  .state('contacts', {...})
  .state('details', {
    parent: 'contacts'
  });
```

Le point dénote sur relation de parent à enfant

# VUES MULTIPLES

Plusieurs vues peuvent être activées sur un même template.

```
<!-- index.html -->
<body>
  <div ui-view="filters"></div>
  <div ui-view="tabledata"></div>
  <div ui-view="graph"></div>
</body>
```

—

```
$stateProvider
  .state('report',{
    views: {
      'filters': {
        templateUrl: 'report-filters.html',
        controller: function($scope){ ... }
      },
      'tabledata': {
        templateUrl: 'report-table.html',
        controller: function($scope){ ... }
      },
      'graph': {
        templateUrl: 'report-graph.html',
        controller: function($scope){ ... }
      }
    }
  })
```

# PRÉPARATION TP - MODULES ET ROUTES

Récupérer l'application fra :

```
git clone https://www.github.com/jntakpe/fra
cd fra
git checkout ex-routes
npm install
npm install -g gulp
bower install
gulp serve
```

Optionnel

```
mvn package
java -jar target/fra.jar --spring.profiles.active=dev
```

Suivre l'exemple de création du module 'fra-home' et de l'état 'main.home'.

# TP MODULES ET ROUTES

- Créer un état pour l'écran 'documentation' ('/doc')
- Créer un état pour l'écran 'liste des requests' ('/requests')
- Créer un état pour l'écran 'liste des endpoints' ('/endpoints')
- Sur la page d'accueil 'home' mettre à jour les liens
- Sur le header et l'écran 'home' mettre à jour les liens
- Appliquer la classe 'active' sur le 'li' du header de la page courante
- Pour chaque état créer les contrôleurs respectifs
- Créer un état pour afficher l'écran d'édition des endpoints ('/endpoints/:id')

# SERVICES

- Permet de stocker la logique métier de l'application
- De nombreux services sont fournis par le framework
- Les services Angular sont des singletons
- Les services fournis par Angular commencent toujours par \$
- Les services sont récupérées via l'injection de dépendance Angular



# INJECTION DE DÉPENDANCES - FONCTIONNEMENT

Par exemple pour utiliser le service \$http, on peut écrire :

```
myModule.controller('MyCtrl', function ($scope, $http) {});
```

Ca marche aussi en inversant l'ordre des paramètres.

Ne fonctionne pas en changeant l'ordre des paramètres

```
myModule.controller('MyCtrl', function ($scope, httpService) {});
```

Pour créer un contrôleur Angular :

- va reconnaître les paramètres de la fonction (en appelant toString() sur le contrôleur)
- puis récupérer les objets auprès de *providers* (ici \$httpProvider)
- ensuite appeler le contrôleur avec les objets récupérés

# INJECTION DE DÉPENDANCES - MINIFICATION

Après minification ce code :

```
angular.module('mod').controller('MyCtrl', function ($scope, $http) {});
```

deviendra :

```
angular.module('mod').controller('MyCtrl', function (a, b) {});
```

AngularJS recherchera des providers aProvider et bProvider qui n'existeront certainement pas.

Pour palier ce problème, on peut utiliser la forme :

```
angular.module('mod')  
  .controller('MyCtrl', ['$scope', '$http', function ($scope, $http) {}]);
```

# SERVICES DISPONIBLES - \$HTTP

Permet de communiquer avec un serveur en abstrayant la plomberie AJAX.

Utilise l'API des promises \$q.

L'API permet de déclarer un callback d'erreur et de succès.

```
$http.get('monServeur', {params: {key1: value1, key2: value2}})
    .success(function(data, status, header, config) {})
    .error(function(data, status, header, config) {});
```

```
$http.post('monServeur', {login: jOSS, password: nokidding})
    .success(function(data, status, header, config) {})
    .error(function(data, status, header, config) {});
```

# SERVICES DISPONIBLES - \$RESOURCE

Service possédant un niveau d'abstraction supérieur à \$http, permettant de communiquer de manière transparente avec des ressources REST en fournissant des méthodes query, save, get, remove, delete.

\$resource provient du module externe  
ngResource qui doit être importé

```
var userRes = $resource('/users/:userId', {userId: '@id'});  
userRes.query(); // récupère la liste des utilisateurs  
var firstUser = userRes.get({userId: 1}); // récupère l'utilisateur 1  
firstUser.name = 'toto'; // change le nom de l'utilisateur  
firstUser.$save(); // enregistrement de l'utilisateur
```

# SERVICES DISPONIBLES - \$HTTPBACKEND

Service de bas niveau utilisé par \$http et \$resource pour faire les requêtes. Ce service est très rarement utilisé directement par les développeurs excepté pour les tests.

```
it('should get 2 users when list is called', function () {  
  
  $httpBackend.expectGET('http://localhost:8080/myapp/users').respond([joss, sophie]);  
  
  scope.list();  
  $httpBackend.flush();  
  expect(scope.users.length).toBe(2);  
});
```

# SERVICES DISPONIBLES - \$LOCATION

Permet de manipuler l'url de la page.

Une url AngularJS est constituée de 6 parties, par exemple pour 'http://localhost:8080/docs/training?subject=angular#service (http://localhost:8080/docs/training?subject=angular#service)' :

- protocol = http
- host = localhost
- port = 8080
- path = /docs/training
- search = subject=angular
- hash = service

Par exemple, on peut modifier le path en écrivant:

```
$location.path('/docs/exercise')
```

# SERVICES DISPONNIBLES - \$TIMEOUT ET \$INTERVAL

\$timeout permet de différer l'exécution d'une action en respectant le cycle de vie des objets AngularJS.

```
var delayed = $timeout(function(){//mon action}, 1000);
```

\$interval permet d'exécution toutes les X secondes une action en respectant le cycle de vie des objets AngularJS.

```
var interval = $interval(function(){//mon action}, 1000, 0);
```

# AUTRES SERVICES DISPONIBLES

- `$log` : permet de wrapper le `console.log()`. Il est possible de déclarer différent niveaux de logs : `debug`, `info`, `warn`, `error`, `log`
- `$cookie` / `$cookieStore` : permet de lire et d'écrire les cookies du navigateur
- `$route` / `$routeParams` : permet de gérer le routage (voir plus tard partie dédiée)



# TYPES DE SERVICES

- *provider*, méthode de création de services
- *factory*, enregistrement d'une factory sans passer par un provider
- *service*, enregistrement d'un service via son constructeur
- *constant*, pour enregistrer une constante
- *value*, objet accessible par les services mais pas par les providers

Généralement seuls *factory* et *service* sont utilisés

# FACTORY

La méthode de création de service la plus fréquemment utilisée.

Une factory doit renvoyer le service à enregistrer, on utilise en général le 'revealing module pattern' :

```
angular.module('mod').factory('RWService', function ($http) {  
  function readUsers() {  
    //Some code  
  }  
  function writeUsers() {  
    //Some code  
  }  
  return {  
    read: readUsers,  
    write: writeUsers  
  }  
});
```

```
angular.module('mod').controller('MyCtrl', function (RWService) {  
  RWService.read();  
  RWService.write();  
});
```

# SERVICE

Pour la création de service au sens AngularJS du terme, le constructeur sera appelé :

```
angular.module('mod').service('RWService', function ($http) {  
    this.read = function readUsers() {  
        //Some code  
    };  
  
    this.write = function writeUsers() {  
        //Some code  
    };  
});  
  
angular.module('mod').controller('MyCtrl', function (RWService) {  
    RWService.read();  
    RWService.write();  
});
```

# CONFIGURATION DES SERVICES

Un service peut posséder un bloc config et run.

Le bloc config est exécuter pendant la phase d'enregistrement du service et peut agir sur le provider correspondant.

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

Le bloc run permet d'agir sur le service une fois le provider enregistré et configuré.

```
angular.module('mod').run(function($http) {  
    $http.defaults.headers.common.Authorization = 'Basic YmVlcDpib29w'  
});
```

# PROMISE

Les promesses sont utilisées par AngularJS pour gérer les traitements asynchrones, elles permettent la composition contrairement aux callback et ainsi éviter le Callback Hell.

Une promesse peut avoir deux résultats :

- fulfilled en cas d'appel avec succès
  - rejected en cas d'erreur
- avec deux états :

- pending en attente de résultat
- settled lorsque le résultat a été renvoyé

On dit que les promesses sont des objets 'thenable'

# PROMISE \$HTTP

Le service \$http renvoie une promise avec en supplément deux méthodes success et error.

```
$http.get('/some/uri').then(  
    function(data)
```

```
{  
    //promise resolved  
}
```

```
, function
```

```
(error)
```

```
{  
    //promise rejected  
}
```

# CRÉER SES PROMESSES

```
angular.module('app').factory('promiseService', function ($q, $timeout) {  
  return {  
    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;  
    }  
  });  
});
```

# PRÉPARATION TP - SERVICES

```
git reset --hard HEAD  
git checkout tp-ex-service  
mvn package  
java -jar target/fra.jar --spring.profiles.active=dev
```

Suivre l'exemple d'utilisation des services sur la vue requests



# TP SERVICES

Sur la vue endpoints :

- Récupérer les données depuis le backend en utilisant un service
- Afficher les données dans la table HTML
- Créer les liens vers la vue 'endpoints.edit'
- Implémenter la méthode de suppression d'un endpoint
- Facultatif : afficher le JSON 'content' dans une popup en utilisant ui.bootstrap

# DIRECTIVES

Les directives sont un concept clé d'Angular, vous en avez déjà utilisé plusieurs sans le savoir.

Une directive est un comportement que vous voulez ajouter ou un composant réutilisable de l'application. C'est un ensemble de méthodes qui régissent un comportement qui peut être injecté dans le HTML permettant une programmation déclarative.

Les directives comme le projet Polymer de Google se rapprochent des web components.

Il s'agit de la partie la plus puissante du framework mais aussi de la plus complexe.

# **DIRECTIVES DE FRAMEWORK**

# DIRECTIVES DE FRAMEWORK - NGAPP

Cette directive permet de déclarer une application AngularJS.

Il ne peut y avoir qu'un ngApp par application. Les ng-app ne peuvent pas s'imbriquer.

Plusieurs syntaxes sont possibles :

- ng-app
- ngApp
- data-ng-app

Il est possible d'avoir plusieurs application sur la même page, dans ce cas il faudra manuellement bootstraper les autres applications en utilisant : `angular.bootstrap()`

# DIRECTIVES DU FRAMEWORK - NGSTRICTDI

Depuis la version 1.3, il est possible de démarrer l'application en mode injection de dépendance stricte c'est-à-dire avec la syntaxe :

```
app.controller('MyCtrl', ['$scope', '$http', function($scope, $http) {  
    // handle minification  
}]);
```

Si un développeur ne respecte pas cette syntaxe, le message d'erreur suivant sera affiché :

MyCtrl is not using explicit annotation and cannot be invoked in  
strict mode

Il est possible d'utiliser strictDI avec les plugins ng-min ou ng-annotate basés sur \$inject.

# DIRECTIVES FRAMEWORK - NGINIT

Cette directive permet d'initialiser une variable dans le scope courant sans utiliser le contrôleur.

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

Cette pratique n'est pas recommandée, il est préférable d'initialiser les variables dans le contrôleur.

# DIRECTIVES FRAMEWORK - NG CONTROLLER

Cette directive permet d'initialiser un contrôleur :

```
<body>  
  <section id="todoapp" ng-controller="TodoCtrl"></section>  
</body>
```

Lorsque l'on utilise un routeur, cette directive sera peu ou pas utilisée.

# **DIRECTIVES DE TEMPLATE**

Ces directives sont moins connues car leur utilisation est transparente.



# DIRECTIVE DE TEMPLATE - NGMODEL

Indique à AngularJS que toute modification de ce champ doit être répercutée sur le modèle.

Le travail est fait par un contrôleur lié à la directive (ngModelController). Ce contrôleur gérera la validation du champ et expose les méthodes :

- \$pristine : permet de savoir si le champ n'a pas été touché
- \$dirty : inverse de \$pristine
- \$valid : savoir si le champ est valide
- \$invalid : savoir si le champ est invalide
- \$error : map contenant les erreurs
- \$untouched : si le champ n'a pas perdu le focus
- \$touched : si le champ a perdu le focus

# DIRECTIVE DE TEMPLATE - FORM

La directive form instancie un contrôleur FormController. Ce contrôleur est un parent des contrôleur ngModelController et il est possible d'accéder aux champs en utilisant le nom du champ suivi du nom du contrôleur.

Ce contrôleur expose les méthodes :

- \$pristine : si l'utilisateur n'a pas touché le formulaire
- \$dirty : si l'utilisateur a touché le formulaire
- \$valid : si l'ensemble des champs du formulaire est valide
- \$invalid : si un des champs du formulaire est invalide
- \$error: représente le dictionnaire des erreurs
- \$submitted : indique si le formulaire a été envoyé

# DIRECTIVE TEMPLATE - EXAMPLE VALIDATION

```
<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>
```

# DIRECTIVE DE TEMPLATE

## - INPUT

La directive input possède un ensemble d'attributs permettant de contrôler le format des champs :

- minLength : valider la longueur min du champ
- maxLength : valider la longueur max du champ
- onChange : appeler une fonction lorsque le champ est modifié
- ngList : convertit un chaîne de caractères en tableau
- ngTrueValue : valeur de la checkbox si true
- ngFalseValue : valeur de la checkbox si false
- ngValue : valeur du radio bouton si coché

# DIRECTIVES DE TEMPLATE - CSS

En plus des attributs spécifiques Angular, les attributs min, max et type email et url sont aussi gérés.

AngularJS ajoute une classe CSS pour décrire les différents état du système.

Par exemple pour un input de type number :

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

```
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; }
```

# DIRECTIVE DE TEMPLATE - NGMESSAGES

Permet d'afficher des messages en fonction de l'état d'un champ sans avoir à passer par plusieurs ng-if.

```
<div ng-if="myForm.myFieldName.$dirty"
ng-messages="myForm.myFieldName.$error">
  <div ng-message="required">The field name is required</div>
  <div ng-message="minlength">The field name is too short</div>
  <div ng-message="maxlength">The field name is too long</div>
</div>
```

Les messages seront affichés par ordre de priorité. Il est également possible d'afficher plusieurs messages en ajoutant l'attribut 'multiple'.

```
<div ng-if="myForm.myFieldName.$dirty"
ng-messages="myForm.myFieldName.$error" multiple>
```

# DIRECTIVES DE TEMPLATE - IF

ngIf permet de créer un élément du DOM en fonction de la valeur d'un paramètre.

Contraire à ngHide ou ngShow, l'élément sera supprimé ou créé sur le DOM plutôt que juste caché.

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

# DIRECTIVES DE TEMPLATE - REPEAT

Permet de parcourir un tableau

```
<li ng-repeat="todo in todos">
```

Chaque élément aura son propre scope exposant les propriétés :

- `$index` : position dans l'itérateur
- `$first` : vrai s'il s'agit du premier élément
- `$last` : vrai s'il s'agit du dernier
- `$middle` : s'il s'agit ni du premier ni dernier élément
- `$even/$odd` : vrai si l'élément est pair/impair



# DIRECTIVES DE TEMPLATE - TRACKBY

ngRepeat ne permet pas d'avoir d'éléments dupliqués dans un tableau.

AngularJS calcule un hash qu'il insère dans la propriété \$\$hashkey. Par exemple, dans le cas d'un tableau de noms, il est possible qu'il y ait un conflit. Dans ce cas, Angular lancera une erreur ngRepeat:dupes.

Pour éviter cette erreur, il est nécessaire de préciser la manière avec laquelle Angular doit suivre les éléments :

```
<div ng-repeat="name in names track by $index">{{ name }}</div>
```

# DIRECTIVE DE TEMPLATE - SWITCH

Permet de remplacer un élément selon la valeur d'une expression :

```
<div ng-switch="user.role">
  <span ng-switch-when="ADMIN">Administrateur</span>
  <span ng-switch-when="USER">Utilisateur</span>
  <span ng-switch-default>Anonyme</span>
</div>
```

# DIRECTIVE DE TEMPLATE - INCLUDE

Permet d'inclure un template après l'avoir compilé dans un élément du DOM

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

# TP - FORMULAIRES

```
git reset --hard HEAD
git checkout start-tp-form
bower install
```

Sur la page 'main.endpoints.edit' :

- Initialiser les champs de la page en cas de modification de endpoint (sauf paramètres)
- En cas de création de endpoint, initialiser la radio 'method' à la valeur GET
- Mettre en oeuvre la validation des champs et afficher les messages d'erreurs respectifs
- Gérer la création et la modification d'un endpoint.
- Une fois le endpoint créé ou modifier, rediriger vers la liste des endpoints.
- Le bouton 'retour' doit renvoyer vers la page des endpoints

# DIRECTIVES DE BINDING

- ngBind : permet d'éviter le flickering (chgt rapide `{{}}` vers template)
- ngBindHtml : permet d'éviter le flickering en bindant directement un template
- ngBindTemplate : pareil que ngBind en supportant de multiple valeurs
- ngNonBindable : pour que les `{{}}` ne soient pas interprétées

# DIRECTIVES DE STYLE - CLOAK

Comme ngBind permet d'éviter le flickering. S'applique sur une balise html qui sera uniquement affichée lorsque que le template sera compilé.

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

# DIRECTIVES DE STYLE - HIDE/SHOW

ngHide et ngShow affiche ou cache un élément en ajoutant un style display:none.

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

# DIRECTIVES DE STYLE - CLASS

Permet de modifier dynamiquement les classes CSS.

```
.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>
```

Il existe également deux directives ngClassEven et ngClassOdd pour changer de classe dans un repeat.



# DIRECTIVES DE STYLE - DISABLED

Comme son nom l'indique permet disable un élément en fonction d'une valeur.

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

# DIRECTIVES DE STYLE - READONLY

Comme ng-disable sauf que l'élément passe en readonly en fonction d'une valeur.

```
<input type="checkbox" ng-model="isReadOnly"/><br/>  
<input type="text" ng-readonly="isReadOnly" />
```

# DIRECTIVES DE STYLE - STYLE

Permet d'ajouter dynamiquement un style sur un élément sous la forme 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>
```

# CRÉER SES DIRECTIVES

Il existe 4 façons d'utiliser les directives :

- comme un élément :
- comme un attribut :
- comme une classe CSS :
- comme un commentaire :

On utilise généralement la forme élément pour nos propres composants et la forme attribut pour ajouter un comportement à un composant existant.

# CRÉER SES DIRECTIVES - TEMPLATE

Template permet de remplacer la directive par le contenu de la propriété template :

```
<myDir>This text will be removed</myDir>
```

```
app.directive('myDir', function() {  
    return {  
        template: '<h1>Some title</h1>'  
    };  
});
```

# CRÉER SES DIRECTIVES - TEMPLATEURL

Il est également possible de passer :

```
<myDir>This text will be removed</myDir>
```

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

# CRÉER SES DIRECTIVES - RESTRICT

Par défaut la directive est restreinte à l'élément.

Il est possible de modifier ce comportement en utilisant :

- A pour attribut
- E pour élément
- C pour classe
- M pour élément

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

```
app.directive('myDir', function() {  
  return {  
    restrict: 'A',  
    templateUrl: 'dirs/mydir.html'  
  };  
});
```

# CRÉER SES DIRECTIVES - TRANSCLUDE

Permet de garder le contenu d'un élément

```
<myDir>This text wont be removed</myDir>
```

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

donnera

```
<h1><div>This text will not be removed</div> Some title</h1>
```



# CRÉER SES DIRECTIVES - LINK

Permet de manipuler le DOM à la façon de jQuery.

```
app.directive('myDir', function() {  
  return {  
    link: function(scope, element, attributes) {  
      scope.turnRed = function () {  
        scope.color = { color: 'red' };  
      }  
    },  
    template: '<h1 ng-click="turnRed({{color}})" ng-class="">Some title  
</h1>'  
  };  
});
```

# CRÉER SES DIRECTIVES - SCOPE

Précédemment les directives héritait du scope englobant.  
Pour faire des composants réutilisables, il est possible de les  
isoler du scope.

```
app.directive('myDir', function() {  
    return {  
        scope: {},  
        template: '<h1>{{color}} Wont work</h1>'  
    };  
});
```

# CRÉER SES DIRECTIVES - @ SCOPE

Il est possible de passer des valeurs au scope isolé par chaîne de caractères.

```
<myDir color="Red" />

app.directive('myDir', function() {
  return {
    scope: {
      myColor: '@color'
    },
    template: '<h1>{{myColor}} Will work</h1>'
  };
});
```

# CRÉER SES DIRECTIVES - = SCOPE

Il est possible de passer des valeurs au scope isolé par référence.

```
<myDir color="obj" />
```

```
app.directive('myDir', function() {  
  return {  
    scope: {  
      obj: '=color'  
    },  
    template: '<h1>{{obj.color}} Will work</h1>'  
  };  
});
```

# CRÉER SES DIRECTIVES - FUNCTION SCOPE

Il est possible de passer des fonctions au scope isolé par référence.

```
<myDir turn-color="turnRed()" />

$scope.turnRed = function() {alert('Red');};

app.directive('myDir', function() {
  return {
    scope: {
      turnColor: '&'
    },
    template: '<h1 ng-click="turnColor()">Will work</h1>'
  };
});
```

# CRÉER SES DIRECTIVES - CONTROLLER

Il est possible d'extraire le comportement de la fonction link dans un contrôleur plus facile à tester unitairement.

```
app.directive('myDir', function() {
    return {
        controller: myCtrl,
        template: '<h1 ng-click="turnRed({{color}})" ng-class="">Some title
</h1>'
    };
});

function myCtrl(scope, element, attributes) {
    scope.turnRed = function () {
        scope.color = { color: 'red' };
    }
}
```