



AK



# Sommaire

- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS





# Logistique

- Horaires
- Déjeuner & pauses
- Autres questions ?



# Rappels

AK

# Plan



- *Rappels*
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS

# Javascript fonctions



- Une fonction est définie par
  - le mot-clé `function`
  - un nom optionnel
  - une liste optionnelle de paramètres entre parenthèses
  - un corps entre accolades
- Une fonction sans nom est dite anonyme
- La propriété `name` d'une fonction donne son nom



# Javascript fonctions

- Une fonction, qu'elle soit nommée ou anonyme, peut être assignée à une variable

```
// Fonction nommée "foo"
function foo() { ... }
// Fonction anonyme
function () { ... }
// Variable pointant sur une fonction existante
var fooVar = foo;
// Variable pointant sur une nouvelle fonction nommée
var barVar = function bar() { ... }
// Variable pointant sur une nouvelle fonction anonyme
var bazVar = function () { ... }
// Noms des fonctions
foo.name === "foo"
fooVar.name === "foo"
barVar.name === "bar"
bazVar.name === ""
```



# Javascript fonctions



- Il est possible de passer une fonction en paramètre d'une autre fonction
  - Très utilisé pour les **callbacks**

```
function callServer(onSuccessFn) {  
    Ajax.get(url, function (data) {  
        onSuccessFn(data);  
    });  
}  
function displayResult(data) {  
    console.log(data);  
}  
callServer(displayResult);  
// Fonction anonyme  
callServer(function (data) {  
    console.log(data);  
});
```

# Javascript Scopes



- La limite dans laquelle un symbole (fonction ou variable) existe est appelée **scope**
- Visibilité des symboles au sein d'un scope
  - Fonctions nommées : utilisables partout dans le scope (**forward-reference** possible)
  - Variables déclarées avec `var` : le symbole existe partout dans le scope, mais sa valeur est `undefined` jusqu'à l'initialisation
  - Variables sans `var` : le symbole devient une propriété de `window` et est globalement accessible (!!)
- En Javascript, les scopes sont uniquement délimités par les corps des fonctions
  - Pas par les accolades des blocs `if / for / while` !

# Javascript Scopes



```
function scope() {  
  
    var answer1 = foo();  
  
    function foo() {  
        return 42;  
    }  
  
    var answer2 = foo();  
  
  
    if (true) {  
        var banana = "banana";  
    }  
  
    console.log(banana);  
  
}
```

# Javascript Scopes



```
> a
ReferenceError: a is not defined
> var a; a
undefined
> (function () { var b = a; var a = 1; return b; }())
undefined
> (function () { c = 42; }())
undefined
> c
42
> window.c
42
```



# Javascript Closures

- JavaScript implémente le concept des closures. Une fonction capture le scope dans lequel elle est déclarée.
- Ce concept est très utilisé avec AngularJS
- Attention, il s'agit bien des pointeurs qui sont capturés

```
var bar = 'hello';

function foo() { console.log(bar); }

foo();                      // "hello"

bar = 'world';

foo();                      // "world"
```

# Javascript This



- Au sein d'une fonction, `this` fait référence à un objet différent selon la façon dont elle est appelée
  - En tant que fonction autonome : `this = window`
  - En tant que méthode d'un objet : `this = l'objet`
  - En tant que constructeur ("new") : `this = l'objet créé`
- Il est possible de spécifier l'objet auquel `this` fait référence en utilisant la méthode `call()`

```
function foo() { console.log(this); }

foo();                                // this = "window"
var object = { bar: foo };
object.bar();                          // this = object
foo.call(object);                     // this = object
```





- **JavaScript Object Notation**
  - extension : `.json`
  - media type : `application/json`
- 2 éléments structurels
  - Objet (ensemble de paires nom / valeur) : `{"nom": "valeur"}`
  - Tableau (liste ordonnée de valeurs) : `["elem1", "elem2"]`



- Types basiques

- Nombre
- Booléen
- Chaîne
- null

# JSON



- JSON

```
{ "menu": {  
    "id": "file", "value": "File", "popup": {  
        "menuitem": [  
            { "value": "Open", "onclick": "OpenDoc()" },  
            { "value": "Close", "onclick": "CloseDoc()" } ]  
        }  
    }  
}
```

- XML

```
<menu id="file" value="File">  
    <popup>  
        <menuitem value="Open" onclick="OpenDoc()" />  
        <menuitem value="Close" onclick="CloseDoc()" />  
    </popup>  
</menu>
```

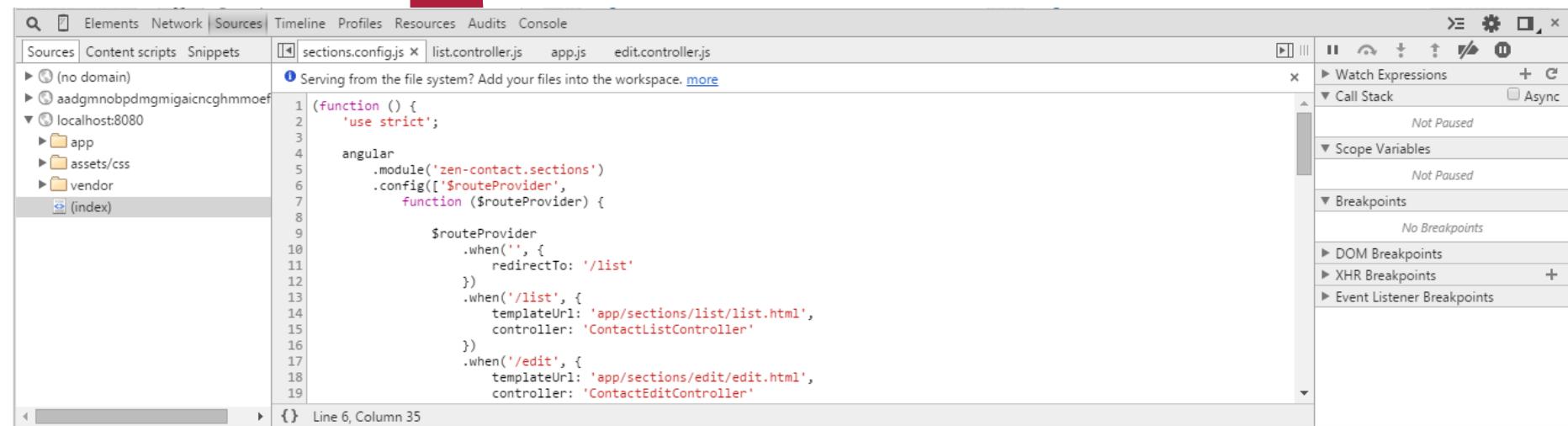
# La boîte à outils du développeur : F12



- La forme la plus basique de débogage consiste à écrire dans la console Javascript des navigateurs

```
console.log("Hello world");
```

- Tous les navigateurs récents proposent également des outils d'analyse et de débogage, accessibles via **F12**





# La boîte à outils du développeur : Batarang

- **Batarang** permet de déboguer les applications AngularJS
  - Plugin pour les outils de développement de Webkit
  - Permet de voir les scopes, les modèles et les services, et de mesurer les performances

The screenshot shows the Batarang developer tool interface. At the top, there is a navigation bar with tabs: Models (selected), Performance, Dependencies, Options, and Help. There is also an 'Enable' button with a checked checkbox.

The main area is titled 'Models'. A dropdown menu labeled 'Root' is set to '007'. Below this, a tree view displays the application's scope hierarchy and model data:

```
Scope (007) | scopes
  Scope (008) | scopes | models
    Scope (00A) | models
      todo:
        text: learn angular
        done: true
        $$hashKey: 009
  Scope (00C) | models
    todo:
      text: build an angular app
      done: false
      $$hashKey: 00B
```



# La boîte à outils du développeur : autres outils

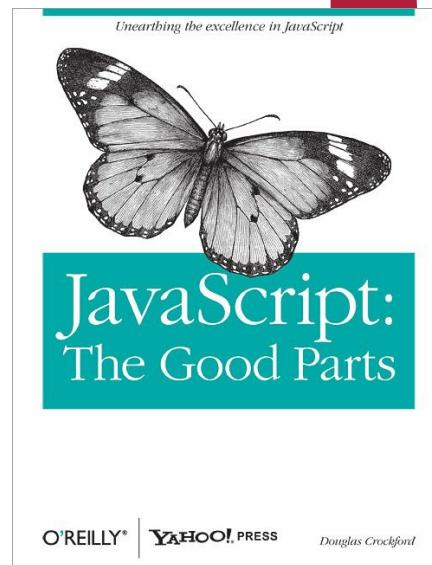


- Mais aussi...
  - **JSLint.com** : outil d'analyse statique développé par D. Crockford. Valide le javascript soumis et teste de nombreuses mauvaises pratiques.
  - **jsfiddle.net** : application web pour tester des fragments de code. L'application permet d'incorporer la plupart des librairies connues.(jquery, extjs...) pour tester directement leurs fonctionnalités.
  - **caniuse.com** : la référence pour connaître le degré de support des fonctionnalités HTML5 par les différents navigateurs.

# Douglas Crockford



- Douglas Crockford
  - Forte implication dans le développement du langage JavaScript et des outils associés (JSLint, JSMin)
  - Créeur du format JSON, une alternative légère au XML
  - Auteur du livre "JavaScript : The Good Parts" mettant en avant les bons côtés et les bonnes pratiques du langage





# Architectures orientées REST

- REST (REpresentational State Transfer) est un style d'architecture pour les systèmes hypermédia distribués, créé par Roy Fielding en 2000
- Architecture
  - Client-Serveur
  - Basée sur l'identification et la manipulation de ressources
  - Stateless : les requêtes au serveur doivent contenir toutes les informations nécessaires à leur traitement
  - Cachable
- L'architecture REST est indépendante du transport utilisé
  - HTTP, SOAP...

# Architectures orientées REST



- Dans un contexte web, les URI sont utilisées comme identifiants de ressources, et les actions HTTP comme commandes
  - Collection : <http://zenika.com/zencontact/contacts>
  - Objet : <http://zenika.com/zencontact/contacts/42>

Ressource	GET	PUT	POST	DELETE
Collection	Récupère la liste des éléments	Remplace la collection entière	Crée une nouvelle entrée dans la collection	Supprime toute la collection
Objet	Récupère les détails de la ressource	Met à jour la ressource	Crée une nouvelle ressource	Supprime la ressource



# AngularJS

AK

# Plan



- Rappels
- *AngularJS*
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS



# AngularJS



- Développé par Google
- Versions:
  - 1.2.x (stable) support à partir de IE8
  - 1.3.x / 1.4.x (stable) support à partir de IE9
  - 1.5.x (stable) support à partir de IE9
  - 1.6.x (Release Candidate)
  - 2.0 (stable) support à partir de IE9
- Site : <https://angularjs.org>
- Code : <https://github.com/angular/angular.js>
- API : <https://docs.angularjs.org/api>
- Blog : <http://blog.angularjs.org>



# AngularJS

- AngularJS est un framework "full stack" : il intègre l'ensemble des composants nécessaires à la réalisation d'applications riches
  - Routage par URL (deep-linking)
  - Contrôleurs Javascript
  - Templates HTML
  - Directives (composants personnalisés)
  - Gestion des ressources REST
  - IOC
  - Tests unitaires et de bout en bout
  - ...



# AngularJS : Points forts



- **Javascript pur**

Adhérence technique limitée, réactivité de l'interface

- **Deep-linking**

Les URLs des "pages" de l'application peuvent être bookmarkées

- **Testabilité**

Possibilité de jouer des tests unitaires ou de bout en bout

- **Binding automatique**

Binding automatique et bi-directionnel entre les placeholders des vues, et avec le modèle du contrôleur



# AngularJS : Cas d'utilisation

- Utilisable
  - **Localement** pour rendre dynamique un fragment de page **templating**, composants...
  - **Globalement** pour structurer une application full-Javascript MVC2 (routage, contrôleurs, templates, ressources REST...)
- Aussi simple ou complexe que nécessaire !



# AngularJS concepts : Contrôleur

- **Contrôleur**
  - Fonction Javascript contenant les données et la logique d'affichage relatives à un `scope`
  - Peut accéder à toutes les variables déclarées dans le contrôleur ou dans la vue gérée par le contrôleur
- **Chapitre 4**



# AngularJS concepts : Service

- **Service**
  - Comme son équivalent côté serveur, un service Angular représente un ensemble cohérent de fonctionnalités
  - Le module ng fournit un grand nombre de services `$document`, `$http`,  
`$location`, `$parse`, `$route`, `$window`...
  - Ce sont des singletons, injectés par IOC dans les contrôleurs
- **Chapitre 6**



# AngularJS concepts : Directive

- **Directive**
  - Une directive encapsule du code et/ou de la logique d'affichage qu'on associe à une balise ou un paramètre dans la page
  - Cela permet de créer des composants auto-contenus et réutilisables
    - Bibliothèque de widgets prêts à l'emploi
  - Handlers génériques pour certains événements
- **Chapitre 11**





# AngularJS concepts : Filtre

- **Filtre**
  - Un filtre est un composant permettant d'appliquer des transformations de données de manière déclarative directement dans la vue
    - Mettre en forme des dates, des montants
  - Filtrer ou ordonner une liste
- **Chapitre 10**

# AngularJS concepts : Module



- **Module**
  - Groupement logique de services, contrôleurs, directives et filtres
  - Configure également le mécanisme d'IOC permettant de les injecter dans les différents composants de l'application
  - On peut utiliser seulement le module par défaut, ou utiliser un module spécifique (toutes les applications non-triviales)
  - Un module peut dépendre d'autres modules
    - Ex: myApp → myServices → ngResource
- **Chapitre 6**

# AngularJS concepts : Ressource



- **Ressource**
  - Permet d'accéder à une ressource serveur exposée en REST
  - Fournit des opérations de type CRUD (Create, Read, Update, Delete) par défaut, et autorise la création d'actions personnalisées
- **Chapitre 8**

# AngularJS



- **Forces**

- Framework complet
- Peu intrusif
- Bonne documentation / Tutoriaux
- Architecture (Injection de dépendances, MVC)
- Templates (HTML)



- **Faiblesses**

- Routage
- I18N (partiellement géré par défaut)
- Directives et binding (difficile à appréhender lorsque l'on est habitué à la logique événementielle de jQuery)
- Performances (si page lourde et non optimisée)
- Navigateurs < IE8 (pour qui ce n'est pas un problème ?)

# AngularJS



- **Conclusion**

- LE framework par excellence pour les applications “single page” (porté par Google) et également un compagnon idéal pour l’intégration dans des applications “page par page” (très peu intrusif)



# Premiers pas

AK

# Plan



- Rappels
- AngularJS
- *Premiers pas*
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS



# Intégration



- Avant de pouvoir utiliser AngularJS, il faut l'importer et l'activer dans la page
- AngularJS est composé de plusieurs scripts, disponibles en version normale (développement) ou minifiée (production)
  - `angular.js`, `angular-resource.js`, `angular-mocks.js`...
  - Permet d'optimiser le chargement de l'application
- A déclarer de préférence à la fin de la page HTML

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



# Activation



- Une fois AngularJS importé, il faut l'activer dans la page
  - Directive `ng-app`
  - Généralement placée sur la balise `<html>`

```
<html ng-app>
  ...
</html>
```

- Il est également possible de charger un module personnalisé (plus d'informations dans le chapitre sur les modules)

```
<html ng-app="myModule">
  ...
</html>
```





# Templates et Bindings : Template

- Un **template** est un modèle de document
  - Page complète ou fragment
  - Complété par des bindings et/ou directives AngularJS
  - Peut rester valide au sens HTML
    - Utilisation de `data-*`, des classes CSS...
- Au sein d'un template, un binding permet d'indiquer la présence d'un élément dynamique
  - Lié à un modèle défini sur un **scope**
  - Plusieurs notations disponibles



# Templates et Bindings : Bindings

- Plusieurs notations possibles pour un binding

- Placeholder à accolades

```
Hello {{name}}
```

- Attribut de balise (notation data-\* valide HTML5)

```
Hello <span ng-bind="name"></span>
Hello <span data-ng-bind="name"></span>
```

- Classe CSS

```
Hello <span class="ng-bind:name"></span>
```



# Templates et Bindings : One-time binding

- Arrête le binding dès que la variable est définie (`!undefined`)

- Placeholder à accolades

```
Hello {{::name}}
```

- Attribut de balise (notation `data-*` valide HTML5)

```
Hello <span ng-bind="::name"></span>
Hello <span data-ng-bind="::name"></span>
```

- Pas de classe CSS

- Utile pour les applications avec beaucoup de bindings

```
<ul>
  <li ng-repeat="item in ::items">{{item.name}}</li>
</ul>
```

```
<div some-directive name="::myName" color="My color is {{::myColor}}"></div>
```





# Templates et Bindings : Cloaking

- Les bindings de type `{{foo}}` peuvent être aperçus par l'utilisateur pendant une fraction de seconde avant qu'AngularJS ne les remplace par leur valeur
- Solutions
  - Utiliser les autres formes de binding
  - Utiliser `ng-cloak` sur une balise parente pour la masquer temporairement

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

<div class="ng-cloak">
  Hello {{name}}
</div>
```

# Templates et Bindings : Autres types de bindings



- Autres types de bindings

- **ng-bind-html**

- Interprète le binding comme du code HTML et supprime les éléments dangereux (voir **\$sanitize**)

```
<span ng-bind-html="comment"></span>
```

- **ng-bind-template**

- Permet de fournir un template, dans les cas où il est impossible de placer des **<span>** (ex : dans **<title>**)

```
<title ng-bind-template="{{site}} - {{page}}></title>
```





# Templates et Bindings : Les modèles

- Pour définir un modèle sur le scope courant (pour l'instant, celui de l'application, c'est-à-dire le scope racine `$rootScope`)

- `ng-init`

Généralement placée sur la balise body

```
<body ng-init="fruit='banana'>
```

Attention cette directive n'est à utiliser que dans de très rares cas voir la documentation pour plus d'informations

- `ng-model`

Utilisable sur les balises input, select, textarea

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





# Gestion des listes

- La directive `ng-repeat` permet d'afficher des listes (appliquée sur l'élément à répéter)

```
<!-- names = ['you', 'me', 'them'] -->
<ul>
  <li ng-repeat="name in names"> {{name}} </li>
</ul>
```

- Fonctionne également avec des maps

```
<!-- people = {bob: 25, alice: 40} -->
<ul>
  <li ng-repeat="(name,age) in people">
    {{name}} is {{age}}
  </li>
</ul>
```



# Gestion des listes

- La boucle expose différentes variables
  - **\$index** (nombre) : index de l'élément courant (0..N-1)
  - **\$first** (booléen) : si c'est le premier élément
  - **\$middle** (booléen) : si c'est un élément intermédiaire (ni premier, ni dernier)
  - **\$last** (booléen) : si c'est le dernier élément
- Exemple : combinaison avec **ng-show / ng-hide**

```
<span ng-repeat="name in ['you', 'me', 'them']">
  {{name}}
  <span ng-hide="$last">, </span>
</span>
// you, me, them
```





# Filtres

- Un filtre permet d'altérer la valeur d'un binding
- AngularJS en fournit un certain nombre, et il est possible de développer ses propres filtres
  - `lowercase`, `uppercase`,
  - `number`, `date`, `currency`
  - `filter`, `limitTo`, `orderBy`
  - `json`
- Syntaxe

```
 {{ expression | filtre1 | filtre2:param1:param2 }}
```

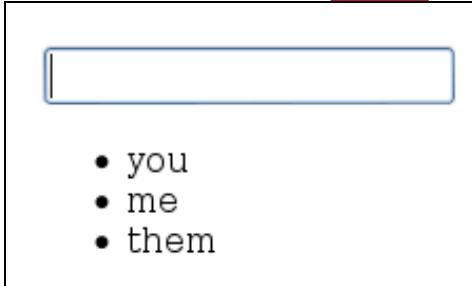


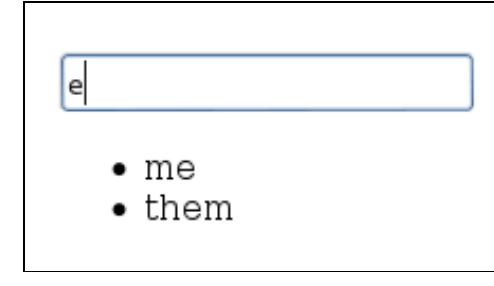
# Filtres



```
<ul ng-init="people=[{name: 'you'}, {name: 'me'}, {name: 'them'}]">
  <li ng-repeat="p in people | orderBy: 'name'">
    {{p.name | uppercase}}
  </li>
</ul>
```

```
<input type="text" ng-model="nameFilter"/>
<ul ng-init="names=['you', 'me', 'them']">
  <li ng-repeat="name in names | filter:nameFilter">
    {{name}}
  </li>
</ul>
```

- 
- you
  - me
  - them

- 
- e
  - me
  - them





# Quelques directives utiles (1/3)

- **ng-switch**
  - Ajout d'une structure DOM parmi plusieurs possibles en fonction d'une expression.
  - La directive s'appuie sur le pattern on/when/default.
- **ng-if**
  - Ajout ou suppression d'une partie du DOM en fonction d'une expression.
  - A la différence des directives **ng-show** / **ng-hide**, les éléments HTML sont recréés plutôt que simplement masqués.
- **ng-include**
  - Compilation et ajout d'un fragment HTML externe.



# Quelques directives utiles (2/3)

- **ng-show / ng-hide**
  - Affiche ou masque l'élément en fonction d'une expression
- **ng-href / ng-src**
  - Garantissent que les attributs href et src incorporant des placeholders {{foo}} seront bien calculés avant d'être utilisables
- **ng-style**
  - Applique un style CSS à l'élément, sous la forme d'une map de propriétés
    - :  

```
<div ng-style="{color: 'red', margin:0}"></div>
```



# Quelques directives utiles (3/3)

- **ng-class**
  - **ng-class-even / ng-class-odd** (dans un **ng-repeat**)
  - Modifient la classe CSS de l'élément

```
<!-- var expr01 = true; var expr02 = false; -->
<div ng-class="{classe1:expr01, classe2:expr02}" />
<!-- équivalent à class="classe1" -->

<!-- var montableau = ["classe1", "classe2"]; -->
<div ng-class="montableau"/>
<!-- équivalent à class="classe1 classe2" -->

<!-- var maChaine = "classe1 classe2"; -->
<div ng-class="maChaine"/>
<!-- équivalent à class="classe1 classe2" -->
```







# Modules

AK

# Plan



- Rappels
- AngularJS
- Premiers pas
- *Modules*
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS





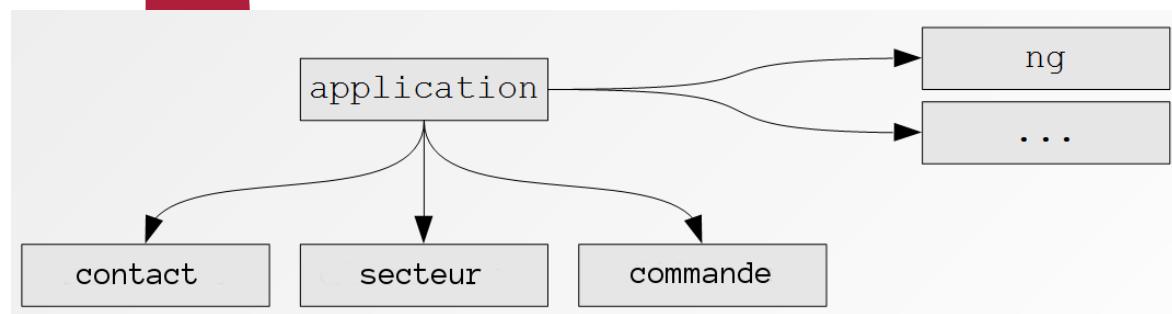
# Modules : Notion de module

- AngularJS propose la notion de **module**
- Un module permet d'encapsuler un ensemble de fonctionnalités
  - Code de configuration
  - Services, contrôleurs, directives et filtres
- Avantages
  - Architecture modulaire
  - Réutilisabilité
  - Testabilité (module de test, module de production)
- Le module **ng** fournit toutes les fonctionnalités vues jusqu'ici
  - Automatiquement importé



# Modules : Notion de module

- Equivalent d'un package en Java
- De manière générale, les modules peuvent être organisés
  - Par type (services, contrôleurs...) [Déconseillé]
  - Par fonctionnalité / domaine ("gestion des contacts", "admin")
- On peut par exemple utiliser le découpage suivant
  - Un module par fonctionnalité (contact, secteur, ...)
  - Un module applicatif qui importe les précédents, ainsi que les modules tierce-parties





# Modules : Déclarer un module

- La fonction `angular.module` permet de déclarer un module

```
angular.module('application', []);
```

- Exemple :

```
angular.module('module1', []);  
angular.module('module2', []);
```

```
angular.module('application',  
    ['module1', 'module2']);
```



# Modules : Utiliser un module

- La fonction `angular.module` permet d'appeler un module

```
angular.module('application');
```

- Exemple

```
// Création du module
angular.module('application', []);

// Utilisation du module
angular.module('application').controller( ... );
```

- La directive `ng-app` permet de spécifier le module applicatif

```
<html ng-app="application">
```



# Modules : Configuration

- Un module dispose de deux fonctions d'initialisation
  - `config()` : appelée au chargement du module
  - `run()` : appelée lorsque tous les modules ont été chargés

```
angular.module('application', []);  
  
angular.module('application')  
  .config(function () {  
    console.log('Chargement...');  
});  
  
angular.module('application')  
  .run(function () {  
    console.log('Initialisation...');  
});
```

- Utile pour configurer l'application au démarrage (ex : le routage des URLs vers les vues)



# Modules et Composants : Factories

- Un module instancie et expose des composants (services, contrôleurs, directives et filtres) grâce à des fonctions **factory**
  - `service(name, configFn)`
  - `controller(name, configFn)`
  - `directive(name, configFn)`
  - `component(name, configObj)`
  - `filter(name, configFn)`
- Exemple : déclaration d'un contrôleur

```
angular.module('app', []);  
  
angular.module('app')  
  .controller('PeopleCtrl', function ($scope) {  
    $scope.names = ['you', 'me', 'them'];  
    $scope.add = function () { ... };  
});
```





# Modules : Valeurs

- Pour finir, un module peut exposer des valeurs
  - Réutilisabilité, principe DRY
  - Injectables dans les composants
- Utilisation des fonctions
  - `value(key, value)`
  - `constant(key, value)`
- la valeur peut être de n'importe quel type (`Object`, `Array`, `String`, ...)

```
angular.module('app').value('Answer', 42);

angular.module('app')
  .controller('PeopleCtrl', function (Answer) {
    console.log('Answer : ', Answer);
});
```





# Contrôleurs et Scopes

AK

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- *Contrôleurs et Scopes*
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS





# Contrôleurs

- AngularJS promeut le pattern MVC, et permet de découpler les données de la façon dont elles sont affichées
  - Permet de tester les algorithmes en isolation
  - Permet de varier les représentations (web, mobile...)
- Ainsi, une application AngularJS est généralement composée
  - De **contrôleurs** en Javascript : Javascript est un langage impératif, idéal pour implémenter les traitements métier
  - De **templates** (vues) en HTML : HTML est un langage déclaratif, pratique pour exprimer une mise en forme. Pour des raisons de performance, AngularJS compile le template sous la forme d'un arbre DOM, et utilise ce dernier pour appliquer les transformations.





# Contrôleurs

- Un contrôleur AngularJS encapsule un modèle de données et les opérations qui agissent sur ce modèle
  - Comparable à un prototype Javascript ou un objet Java / C#
  - Développé en pur Javascript
- Modèles et fonctions doivent être associés au `$scope` fourni par AngularJS en paramètre du contrôleur

```
angular.module('application')
  .controller('PeopleController', function ($scope) {
    $scope.names = ['you', 'me', 'them'];

    $scope.add = function () {
      $scope.names.push('others');
    };
});
```





# Contrôleurs

- Pour afficher les données et appeler les fonctions du contrôleur, il faut l'associer à un template, qui peut être
  - Une page entière
  - Une vue, dans une application de type single-page
  - Une zone délimitée par une balise (ci-dessous, un `<div>`)
- Directive `ng-controller`

```
<div ng-controller="PeopleController">
  <ul>
    <li ng-repeat="name in names">{{name}}</li>
  </ul>
  <button ng-click="add()">add</button>
</div>
```





# Scopes : Principes

- Dans une application AngularJS, plusieurs scopes peuvent cohabiter
  - Le scope de l'application (scope racine ou `$rootScope`)
  - Les scopes des contrôleurs
  - Les scopes des directives
  - Des scopes dynamiques (ex : au sein d'un `ng-repeat`)
- Les scopes héritent de leurs scopes parents
  - Permet d'architecturer l'application sous forme de modules isolés mais partageant un référentiel commun
- **Batarang** (plugin pour Chrome) permet de visualiser l'imbrication et le contenu des différents scopes

# Scopes : Visualisation avec Batarang



## Models

```
Scope (002) | scopes | models
Scope (003) | scopes | models
Scope (005) | models
Scope (007) | models
Scope (009) | models
Scope (00B) | models
Scope (00D) | models
Scope (00F) | models
  contact:
    id: 5
    firstName: Johnny
    lastName: Storm
    address: Baxter building, New York
    phone: 555-TORCH
    $$hashKey: 00E
Scope (00H) | models
Scope (00J) | models
Scope (00K) | models
Scope (00L) | models
Scope (00M) | models
Scope (00N) | models
Scope (00O) | models
Scope (00P) | models
Scope (00Q) | models
Scope (00R) | models
```





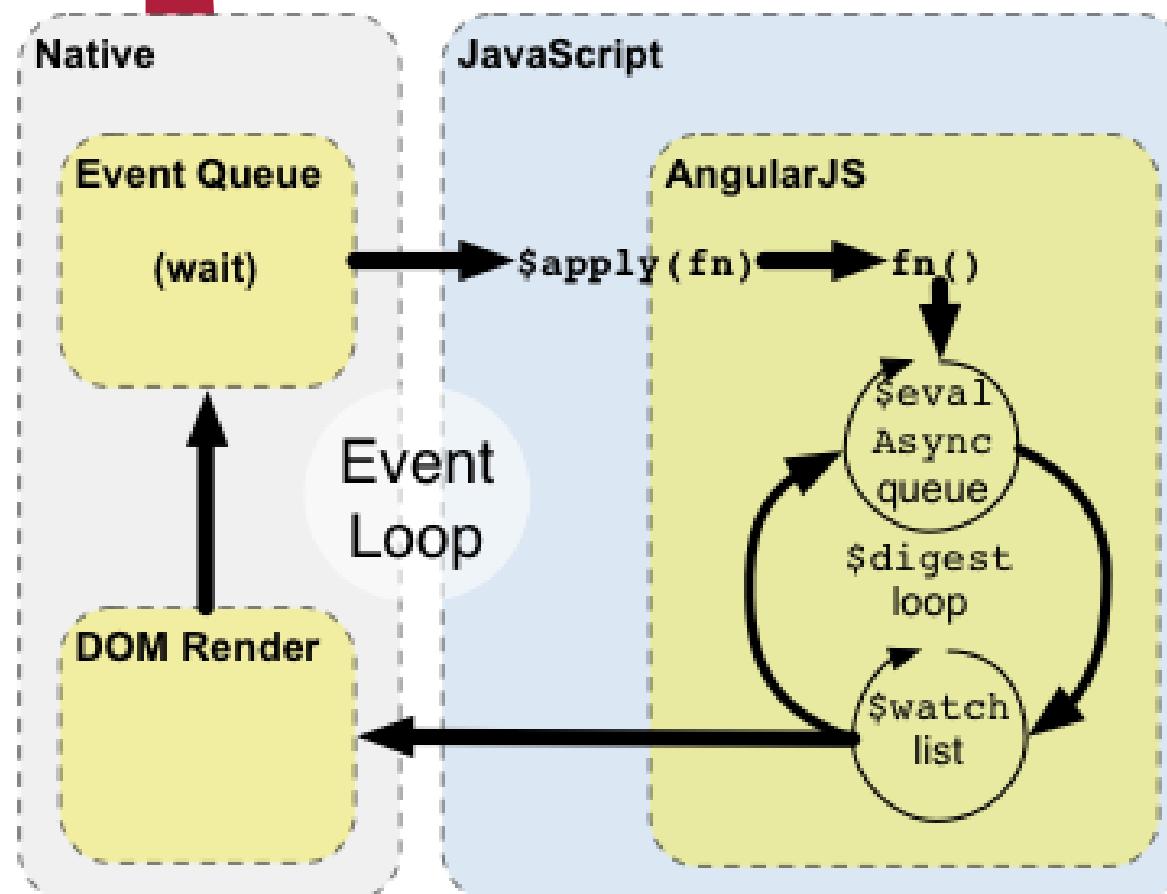
# Scopes : Debug

- Pour debugger (voir et modifier) le scope depuis son navigateur
  - La variable `$0` en console correspond à l'élément du DOM sélectionné
  - `angular.element(<noeud ou selecteur>)` comme `$()` pour jQuery décore le noeud avec les méthodes du framework
  - `angular.element($0).scope()` permet d'obtenir le scope d'un noeud du DOM
- Inspecter un élément du DOM, puis taper depuis la console :
  - Google Chrome : `$scope`
  - Firefox : `angular.element($0).scope()` ou `$($0).scope()`
- N'oubliez pas lors de modification de scope de prévenir Angular en invoquant le `$apply` sur le scope : `.$apply()`



# Scopes : Fonctionnement

- Toute la magie d'AngularJS vient de sa gestion des scopes





# Scopes : Fonctionnement

- Un événement natif est déclenché dans le navigateur
- La fonction callback associée est transférée dans l'univers AngularJS à l'aide de la fonction `$apply` du scope
- AngularJS "digère" la fonction
  - Il applique les modifications requises au modèle
  - Ces modifications sont détectées par la méthode `$watch`, et peuvent déclencher d'autres modifications en cascade
  - Ce cycle se répète jusqu'à ce que le modèle se stabilise
- Les nouvelles valeurs du modèle sont appliquées au template
- Enfin, le navigateur peut afficher le DOM mis à jour



# Scopes : \$apply

- Lorsque l'événement est généré par une balise gérée par AngularJS, il passe automatiquement par **\$apply**
  - Par exemple: ng-model sur un champ `<input>`
  - Utiliser **\$apply** uniquement depuis du Javascript pur ou des librairies tierces
- Pour appeler **\$apply** manuellement (depuis un framework tiers par exemple), il faut récupérer une référence sur le scope
  - Par injection depuis un composant AngularJS
  - Ou en utilisant les API AngularJS

```
var element = document.getElementById( ... );
var scope = angular.element(element).scope();
scope.$apply(function () { ... });
```





# Scopes : \$apply

```
angular.module('application')
  .controller('PeopleController', function ($scope) {
    $scope.names = ['you', 'me', 'them'];
 });
```

```
<div ng-controller="PeopleController">
  <ul>
    <li ng-repeat="name in names">{{name}}</li>
  </ul>

  <button onclick="
    var scope = angular.element(this).scope();
    scope.$apply(function () {
      scope.names.push('others');
    });
  >more</button>

  <!-- Equivalent AngularJS -->
  <button ng-click="names.push('others');">more</button>
</div>
```





# Scopes : \$watch

- La méthode `$watch` permet de surveiller un champ du `$scope`
- AngularJS utilise la technique du "**dirty-checking**" pour n'appeler l'observateur qu'en cas de réelle modification
  - 2 modes de comparaison : **by-reference** (égalité stricte des références) ou **by-value** (comparaison des valeurs des champs)

```
angular.module('application')
  .controller('PeopleController', function ($scope) {
    $scope.names = ['you', 'me', 'them'];

    $scope.$watch('names', function (newValue, oldValue) {
      console.log(oldValue + " → " + newValue);
    }, false); // false === comparaison "by-reference"

  });
}
```





# La syntaxe « controller as »

- Dans la directive `ng-controller` :

```
<div ng-controller="ParentController as parent">
  <span>{{parent.name}}</span>
  <div ng-controller="ChildController as child">
    <span>{{parent.name}} - {{child.name}}</span>
  </div>
</div>
```

- Sans controller as :

```
<div ng-controller="ParentController">
  <span>{{name}}</span>
  <div ng-controller="ChildController">
    <span>{{$parent.name}} - {{name}}</span>
  </div>
</div>
```



# La syntaxe « controller as »

- Permet de s'affranchir de l'utilisation du `$scope` (absent dans Angular 2)
- Utilisation du contrôleur comme une classe JavaScript
- Méthodes et propriétés sont associées à l'instance du contrôleur (object `this`) et non plus au `$scope`

```
angular.module('application')
  .controller('PeopleController', function () {
    var vm = this;

    vm.names = ['you', 'me', 'them'];

    vm.add = function () {
      vm.names.push('others');
    };
  });
}
```





# Routeur

AK

# Plan



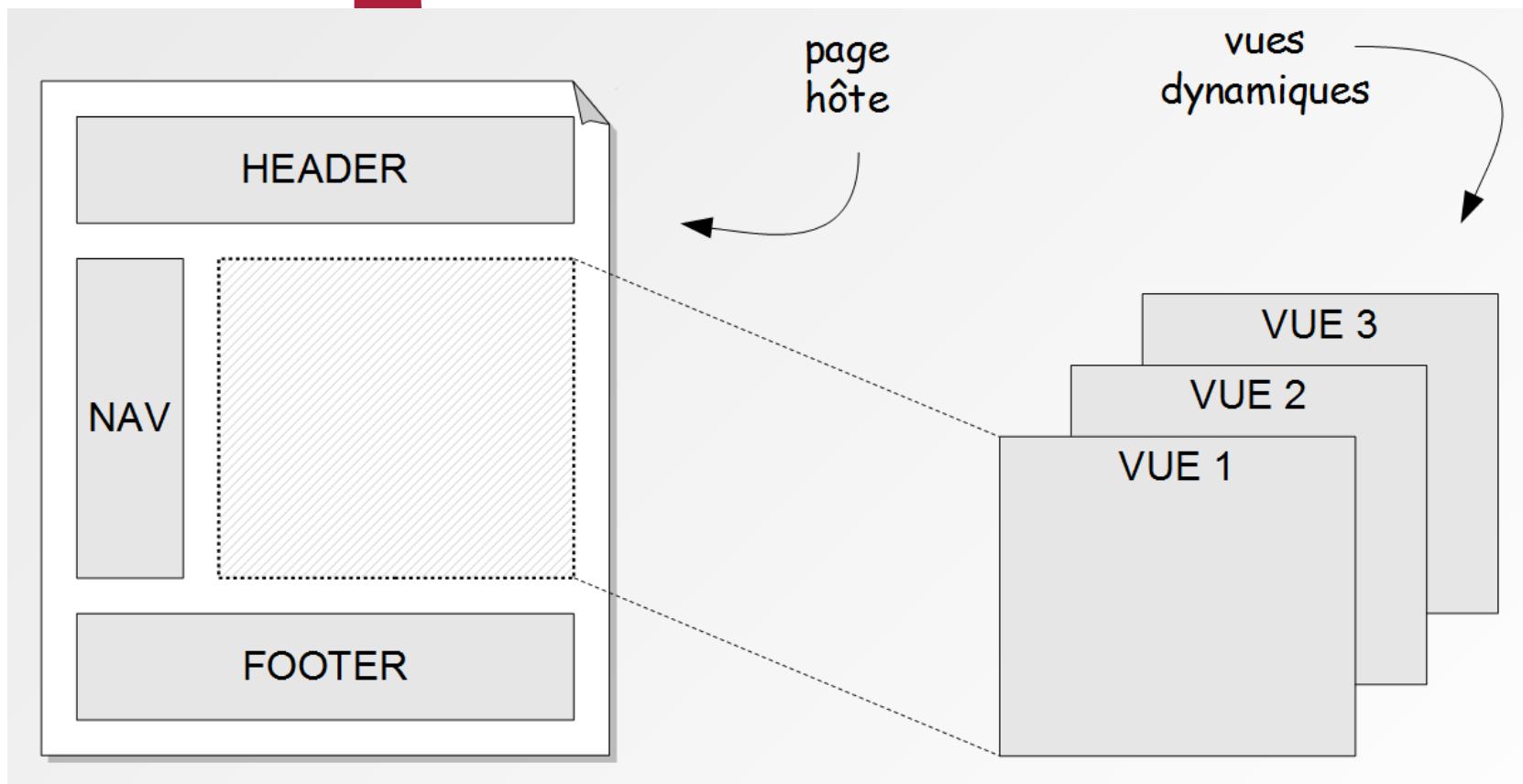
- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- *Routeur*
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS



# Routeur



- AngularJS permet de développer des applications multi-vues
  - Architecture MVC côté client en pur Javascript



# Routeur



- Dans la page hôte, la directive `ng-view` indique l'emplacement où les vues seront insérées

```
<body>
  <div>
    <h1>ZenContacts</h1>
    <h2>Taking care of your people, one at a time.</h2>
  </div>
  <div ng-view></div>
</body>
```



# Configuration du routage

- Le service `$routeProvider` permet de paramétriser la table de routage, qui associe des vues à des URLs particulières
  - `when(route, config)` : configure une route
  - `otherwise(config)` : si aucune route ne correspond
- Propriétés usuelles de configuration d'une route
  - `controller` : nom du contrôleur gérant la vue
  - `controllerAs` : nom de la référence du contrôleur
  - `templateUrl` : chemin vers la vue (template)
  - `redirectTo` : URL vers laquelle rediriger l'utilisateur



# Configuration du routage

- Il existe d'autres possibilités de configuration
  - En particulier, on peut associer des propriétés personnalisées aux routes (ex : pour donner un titre de page à chaque vue)

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

angular.module('app')
.config(function ($routeProvider) {
$routeProvider
    .when('/foo', {
        templateUrl: 'view/foo.html',
        controller: 'FooController',
        controllerAs: 'foo' })
    .when('/bar', {
        templateUrl: 'view/bar.html',
        controller: 'BarController',
        controllerAs: 'bar'
        title: 'Welcome to the bar!' });

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



# Configuration du routage

- Il est également possible de spécifier des patterns d'URL

```
$routeProvider
  .when('/contactBook/:bookId/contact/:id', {
    templateUrl: 'view/foo.html',
    controller: 'FooController',
    controllerAs: 'foo'
  });
}
```

- Le service `$routeParams` permet alors d'accéder aux valeurs des placeholders

```
$routeParams.bookId
$routeParams.id
```





# Configuration du routage

- La configuration de routage est exposée par le service `$route`
  - routes configurées : `$route.routes`
  - propriétés de la route courante : `$route.current` (y compris les propriétés personnalisées)

```
$route.current.title
```





# Configuration du routage

- Le service `$route` expose également différents événements
  - `$routeChangeStart`, `$routeChangeSuccess`, `$routeChangeError`, `$routeUpdate`
- Exemple :

```
function FooController($scope,$route) {  
  $scope.$on('$routeChangeSuccess', function () {  
    console.log("route changed");  
  });  
}
```

# Format des URLs



- Mode par default
  - `http://server:port/zendcontact/index.html#/list?a=b&c`
- AngularJS fait usage du caractère `#` ("Hash" en anglais) qui permet de modifier l'URL sans déclencher un rechargement de la page par le navigateur
- Il existe également un mode HTML5 qui se base sur l'API History HTML5 mais qui demande alors une gestion côté serveur des URL (URL rewriting)



# Gestion de la barre d'adresse

- Le service `$location` permet de lire et de modifier l'URL dans la barre d'adresse du navigateur
  - Abstraction de `window.location`
  - Synchronisation bidirectionnelle
- Accesseurs
  - [ro] `protocol()`, `host()`, `port()`
  - [rw] `path()`, `search()`, `hash()`
  - [rw] `url()` = path + search + hash
  - [ro] `absUrl()` = URL complète





# Services et Injection

AK

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- *Services et Injection*
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS





# service() vs factory()

- Il existe deux méthodes pour créer un service
  - Méthode `module.service(nom, configFn)`
  - Méthode `module.factory(nom, configFn)`
  - Permet d'exécuter du code avant d'instancier le service
- Différences
  - Avec `service()`, `this` fait référence à l'instance du service en cours de création.
  - Avec `factory()`, il faut instancier manuellement le service et retourner la nouvelle instance. `this` fait référence au module.



# service() vs factory()

- Avec `module.service()`

```
module.service('peopleSrv', function () {
  this.foo = 42;
  this.bar = function () { ... };
});
```

- Avec `module.factory()`

```
module.factory('peopleSrv', function () {
  var instance = {};
  instance.foo = 42;
  instance.bar = function () { ... };
  return instance;
});
```



# Injection de dépendances

- AngularJS propose un mécanisme d'**injection de dépendances**
- Principe : un composant ne va pas chercher lui-même ses dépendances, mais attend qu'on les lui fournisse
  - Aussi appelé **IOC** (Inversion Of Control)
- Avantages
  - Simplifie le code
  - Permet de varier les implémentations d'une dépendance (production vs mock)



# Injection de dépendances : Injection dans un module



- Dans le cas d'un module
  - Les dépendances envers les autres modules sont déclarées dans le tableau des dépendances
  - Les fonctions d'initialisation `run` et `config` acceptent également des dépendances en paramètre

```
// dépendance envers les modules annexes
angular.module('app', ['module1', 'module2']);

// injection du service $routeProvider
module.config(function ($routeProvider) {
    $routeProvider.when(...);
});
```

# Injection de dépendances : Injection dans un service, contrôleur...



- Dans le cas des services, contrôleurs, filtres et directives, les dépendances sont déclarées comme des paramètres de la fonction de configuration

```
module.service('peopleSrv', function () {  
    ...  
});  
  
// le $scope des contrôleurs est également une dépendance injectée !  
module.controller('PeopleCtrl', function ($scope, peopleSrv) {  
    ...  
});
```



# Injection de dépendances : Minification et offuscation



- AngularJS se base sur le nom du paramètre pour déterminer la dépendance à injecter
  - Nom défini lors de son instantiation par le module
- Problème : en cas de minification et/ou obfuscation du code, les variables, fonctions et paramètres sont renommés !

- Avant

```
angular.module('module1').controller('PeopleCtrl', function ($scope) { ... });
```

- Après

```
var$21.func$5('module1').func$33('PeopleCtrl', function (param$13) { ... });
```

- **ngStrictDi** : force la déclaration explicite des injections
  - Les injections ne présentant pas d'alias s'affichent en erreur dans la console

# Injection de dépendances : Minification et offuscation



- Solution : une notation alternative permet de préciser les dépendances sous forme de String
  - Les Strings ne sont pas minifiées ni obfusquées
- Utilisation d'un tableau
  - Les N premiers éléments sont les noms des dépendances
  - Le dernier élément est la fonction de configuration
  - L'ordre des noms doit correspondre à celui des paramètres

```
module.controller('MyCtrl', ['$foo', '$bar', function (foo, bar) {  
    foo(...);  
    bar(...);  
}]);
```



# Injection de dépendances : Minification et obfuscation



- Déclaration séparée

```
function MyCtrl(foo, bar) {  
    foo(...);  
    bar(...);  
};  
  
module.controller('MyCtrl', ['$inject', 'foo', 'bar', MyCtrl]);
```

- `$inject` : Déclaration et injection séparée

```
function MyCtrl(foo, bar) {  
    foo(...);  
    bar(...);  
};  
  
MyCtrl.$inject = ['$inject', 'foo', 'bar'];  
  
module.controller('MyCtrl', MyCtrl);
```







# Tests

AK

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- *Tests*
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS



# Concepts



- AngularJS a été conçu avec l'objectif d'être facilement testable
- C'est principalement l'injection de dépendances qui facilite les tests
- Il est possible de démarrer un composant dans un contexte «mocké» pour pouvoir le tester en isolation
- AngularJS propose *angular-mocks.js* qui fournit ces outils
- Dans la documentation Jasmine est utilisé comme framework de tests
- AngularJS peut également être testé avec d'autres frameworks
- Karma propose d'exécuter facilement les tests
  - Il a été développé par l'équipe d'AngularJS, il est donc mis en avant
  - Il n'est pour autant ni indispensable ni lié à AngularJS

# Jasmine



- DOM-less simple JavaScript testing framework
- <http://jasmine.github.io/>
- Propose une API JS plutôt naturelle pour décrire des tests
- Exemple :

```
describe("A suite", function () {
  it("contains spec with an expectation", function () {
    expect(true).toBe(true);
  });
});
```

# Jasmine



- Fournit la page SpecRunner.html

```
Jasmine 1.3.1 revision 1354556913                                         finished in 0.027s
● ● ● ● ●

Passing 5 specs                                                       No try/catch 
```

Player  
should be able to play a Song

when song has been paused  
should indicate that the song is currently paused  
should be possible to resume  
tells the current song if the user has made it a favorite

#resume  
should throw an exception if song is already playing



- Boite à outils fournie par AngularJS pour les tests unitaires
- Charger le fichier `angular-mocks.js` dans les tests
- Définit des fonctions globales : `dump`, `inject`, `module`, `TzDate`
- Définit des services supplémentaires : `$exceptionHandler`, `$httpBackend`, `$log`, `$timeout`
- `module('moduleName')`
  - Charge le module et toutes ses dépendances et les rend disponibles à la fonction `inject`
- `inject(function (...) {...})`
  - Exécute la fonction en mettant en œuvre l'injection de dépendances d'AngularJS avec le contexte du module chargé
  - `inject.strictDi()` : active le mode 'strict-di' dans les tests

# Jasmine & ngMock

- Avec les outils proposés par `ngMock`, il est possible de faire un test Jasmine d'un composant d'AngularJS

```
angular.module('myApplicationModule', [])
    .value('mode', 'app')
    .value('version', 'v1.0.1');

describe('MyApp', function () {
    beforeEach(module('myApplicationModule'));

    it('should provide a version',
        inject(function (mode, version) {
            expect(version).toEqual('v1.0.1');
            expect(mode).toEqual('app');
        })
    );
});
```





# ngMock \$httpBackend

- `$httpBackend` est le service fourni par `ngMock` qui permet de mocker des requêtes HTTP
- Il fonctionne dans le système d'injection de dépendances d'Angular et intercepte les requêtes lancées par `$http`
- `$httpBackend.expect`
  - Définit une requête « attendue » (stricte) → Assertion + Bouchon
  - alias : `$httpBackend.expectGET`, `$httpBackend.expectPUT`, `$httpBackend.expectPOST`, `$httpBackend.expectDELETE`
- `$httpBackend.when`
  - Définit une requête qui « peut » arriver (lâche) → Bouchon
  - alias : `$httpBackend.whenGET`, `$httpBackend.whenPUT`, `$httpBackend.whenPOST`, `$httpBackend.whenDELETE`



# ngMock \$httpBackend



```
describe('MyController', function () {
  var $httpBackend, $scope, myController;

  beforeEach(inject(
    function (_$httpBackend_, $rootScope, $controller) {
      $scope = $rootScope.$new();
      $httpBackend = _$httpBackend_;

      $httpBackend.when('GET', '/auth.py')
        .respond({userId: 'userX'}, {'A-Token': 'xxx'});

      myController = $controller('MyController', {
        '$scope': $scope
      });
    }
  ));
});
```





# Test d'un contrôleur

- Le contrôleur à tester

```
angular.module('myModule')
  .controller('myController', function ($http) {
    var vm = this;

    vm.foo = 'bar';
    $http.get('http://localhost/api/foo')
      .then(function (response) {
        vm.foo = response.data;
      });
  });
});
```

# Test d'un contrôleur



- Le test

```
describe('MyController Test', function () {
  var myController, $httpBackend;

  beforeEach(module('myModule'));

  beforeEach(inject(function (_$httpBackend_, $controller) {
    $httpBackend = _$httpBackend_;
    myController = $controller('myController');
  }));

  it('should switch foo from bar to new bar', function () {
    $httpBackend.expect('GET', 'http://localhost/api/foo').respond('new bar');
    expect(myController.foo).toBe('bar');
    $httpBackend.flush();
    expect(myController.foo).toBe('new bar');
  });
});
```



# Test d'une directive



```
describe('directives', function () {
  beforeEach(module('myApp.directives'));

  describe('app-version', function () {
    it('should print current version', function () {

      module(function ($provide) {
        $provide.value('version', 'TEST_VER');
      });

      inject(function ($compile, $rootScope) {
        var element = $compile('<span app-version></span>)($rootScope);
        expect(element.text()).toEqual('TEST_VER');
      });

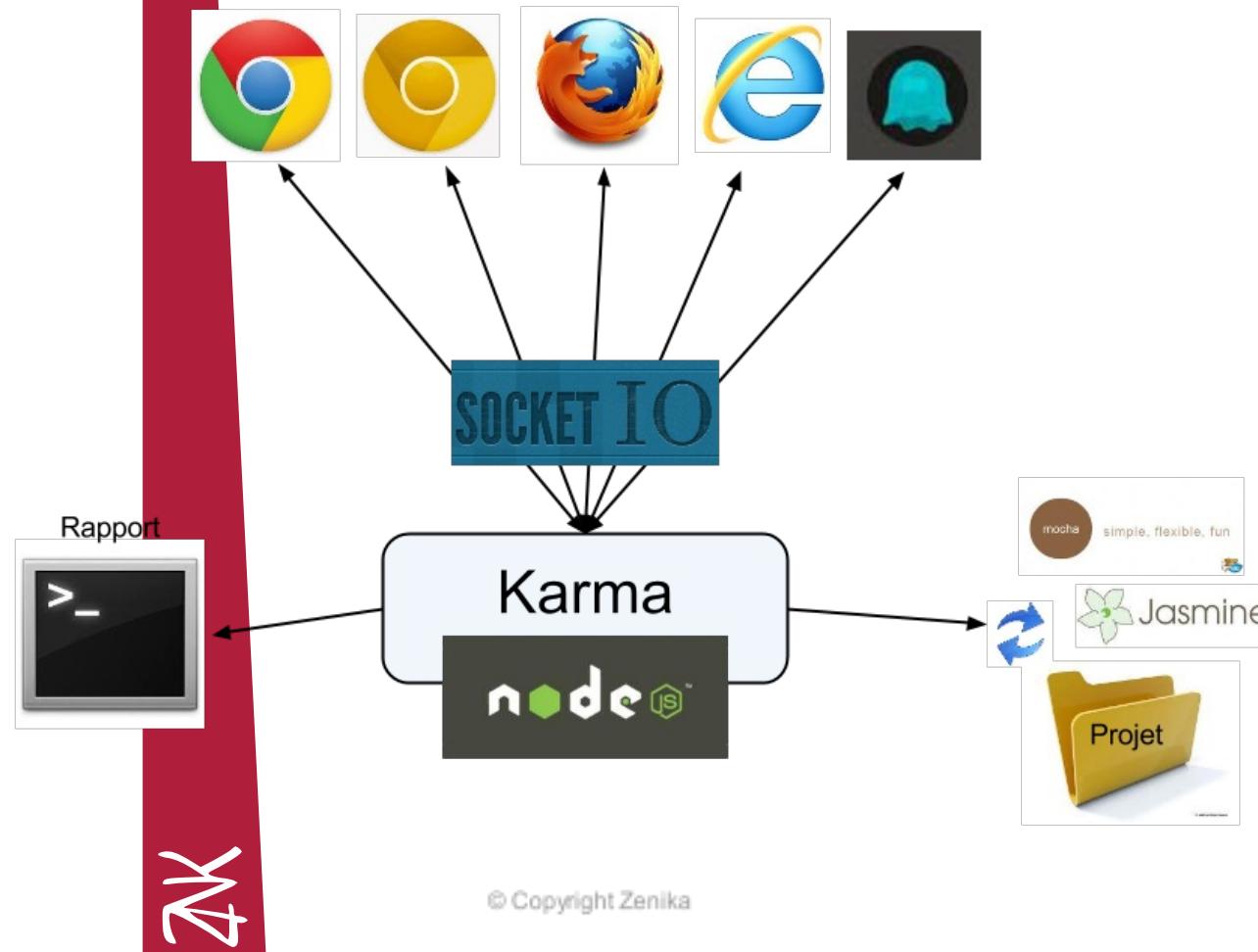
    });
  });
});
```



# Karma



- Karma est un outil qui permet d'automatiser l'exécution des tests





# Configuration de Karma

- Karma nécessite un fichier de configuration en JS, il contient :
  - Le framework de tests (compatibilité avec plusieurs fmk)
  - La liste des scripts à charger
  - Les navigateurs à piloter
  - Diverses configurations techniques





# Configuration de Karma

```
module.exports = function (config){  
  config.set({  
    basePath : '../',  
    files : [  
      'app/lib/angular/angular.js',  
      'app/lib/angular/angular-*.*js',  
      'test/lib/angular/angular-mocks.js',  
      'app/js/**/*.*js',  
      'test/unit/**/*.*js'  
    ],  
    autoWatch : true,  
    frameworks: ['jasmine'],  
    browsers : ['Chrome']  
  })  
}
```



# Tests E2E : Concepts

- Les tests end-2-end proposent une toute autre approche et sont complémentaires avec les tests unitaires
  - Simulent l'exécution complète de l'application
  - Plus lents et plus fastidieux à écrire ( → Page Object Pattern)
- Utilisent Protractor, également développé par Angular
  - S'appuie sur Selenium
  - Encapsulation de WebdriverJS
- Permettent le pilotage du browser
- Permettent d'accéder au DOM facilement
- <https://github.com/angular/protractor/>



# Tests E2E : API

- L'API est inspirée de Jasmine
  - `describe`, `beforeEach`, `afterEach`, `it...`
- Programmation des interactions séquentielles (le « control flow »)
  - Une mécanique assez complexe gère l'enchaînement des actions quand la précédente est terminée
  - Basé sur les promesses
  - En rupture avec l'asynchronisme naturel du JS
  - Nécessite de bien rester dans l'API



# Tests E2E : API

- Pilotage de la page
  - `browser.get(http://localhost:8080)`
  - `element(selector).sendKeys(value)`
  - `element(selector).click()`
- Assertions
  - `expect(future).<matcher>(expected)`
  - `future = element, repeater...`
  - `matcher = toBe, toContain, toBeLessThan...`



# Tests E2E : API

- Sélections
  - locators WebDriver
    - `element(by.id('foo'))`
    - `element(by.className('foo'))`
  - locators Protractor (spécifiques Angular)
    - `element(by.model('contact.name'))`
    - `element(by.binding('contact.name'))`



# Tests E2E : Exemple

```
describe('Phone list view', function () {
  beforeEach(function () {
    browser.get('http://localhost:8080/phones');
  });

  it('represents data correctly', function () {
    expect(element(by.tagName('h2')).getText()).toBe('Phone list');

    expect(element.all(by.css('img.thumb')).count()).toBe(6);

    element(by.id('filterInput')).sendKeys('nexus');

    var phoneRows = element.all(by.repeater('phone in phones'));
    phoneRows.each(function (element) {
      expect(element.getText()).toMatch(/nexus/);
    });
  });
});
```





# Tests E2E : Module ngMockE2E

- Externaliser dans `angular-mocke2e.js`
- Contient uniquement un `$httpBackend` « spécial E2E »
- Permet de mocker certaines requêtes (règles lâches) pour maîtriser les données retournées.
- Permet de transmettre au service `$http` « réel » d'autres requêtes (récupération de template par exemple)
- Pas de règles strictes (`expect(...)` nécessitant un `.flush()`)

```
var phones = [{name: 'phone1'}, {name: 'phone2'}] ;
// Return test datas
$httpBackend.whenGET('/phones').respond(phones);
// Delegate to the $http (perform a real request)
$httpBackend.whenGET(/^\templates\/\//).passThrough();
```







# I18N

AK

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- *I18N*
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS



# Installation



- AngularJS offre des fonctionnalités d'internationalisation
  - 155 langues ou variantes
  - Fichiers `i18n/angular-locale_*.js`
- C'est au développeur de sélectionner et d'importer les locales qu'il souhaite supporter
  - Déterminées côté serveur
  - Alternativement, les scripts peuvent être concaténés (manuellement) afin de limiter le nombre de requêtes

```
<script src="js/angular/angular.js" />
<script src="js/angular/i18n/angular-locale_fr.js" />
```

- Attention, un seul fichier `angular-locale_*.js` maximum dans le html
- Les filtres `number`, `date` et `currency` utilisent alors le bon format



# Nombres



- Le filtre **number** permet de localiser les nombres

```
  {{ <binding/expression> | number[:fractionSize] }}
```

- Paramètres

- [number] **fractionSize** : nombre de décimales  
Par défaut 3 ; des zéros sont ajoutés au besoin

```
  {{ 1234.5678 | number }}
```

Locale EN : 1,234.568

Locale FR : 1 234,568

Locale DE : 1.234,568

```
  {{ 1234.5678 | number:1 }}
```

Locale FR : 1 234,6

```
  {{ 1234.5678 | number:6 }}
```

Locale FR : 1 234,567800



# Dates et heures

- Le filtre **date** permet de localiser les dates et heures

```
  {{ <binding/expression> | date[:format] }}
```

- Paramètres

- [string] **format** : format de date/heure

Format prédéfini ("short", "long"... ) ou personnalisé

```
  {{ 1366668000000 | date:'short' }}  
Locale EN : 4/23/13 12:00 AM  
Locale FR : 23/04/13 00:00  
Locale DE : 23.04.13 00:00  
  {{ 1366668000000 | date:'fullDate' }}  
Locale FR : 23 avril 2013  
  {{ 1366668000000 | date:'dd/MM/yyyy' }}  
Locale FR : 23/04/2013
```



# Dates et heures

- Formats prédéfinis (dépendant de la locale)

```
// Locale EN
'medium'      : Apr 23, 2013 12:00:00 PM
'short'        : 4/23/13 12:00 PM
'fullDate'     : Tuesday, April 23, 2013
'longDate'     : April 23, 2013
'mediumDate'   : Apr 23, 2013
'shortDate'    : 4/23/13
'mediumTime'   : 12:00:00 PM
'shortTime'    : 12:00 PM
// Locale FR
'medium'      : 23 avr. 2013 12:00:00
'short'        : 23/04/13 12:00
'fullDate'     : mardi 23 avril 2013
'longDate'     : 23 avril 2013
'mediumDate'   : 23 avr. 2013
'shortDate'    : 23/04/13
'mediumTime'   : 12:00:00
'shortTime'    : 12:00
```



# Dates et heures

- Format personnalisé
  - Composé à partir de symboles représentant les composantes d'une date
  - Gestion manuelle des locales

'yyyy'	, 'yy'	, 'y'	: Année
'MMMM'	, 'MM'	, 'M'	: Mois
'dd'	, 'd'		: Jour (numéro)
'EEEE'	, 'EEE'		: Jour (nom)
'HH'	, 'H'		: Heure (sur 24h)
'hh'	, 'h'		: Heure (sur 12h)
'mm'	, 'm'		: Minute
'ss'	, 's'		: Seconde
'a'			: AM/PM
'Z'			: Décalage du fuseau horaire





# Montants monétaires

- Le filtre **currency** permet de localiser les montants monétaires

```
 {{ <binding/expression> | currency[:symbol] }}
```

- Paramètres

- [string] **symbol**: symbole monétaire (€, \$, USD...) Le formatage reste celui de la locale, seul le symbole change

```
 {{ 1234.5678 | currency }}  
Locale EN : $1,234.57  
Locale FR : 1 234,57 €  
Locale NO : kr 1 234,57  
 {{ 1234.5678 | currency:'€' }}  
Locale EN : €1,234.57  
Locale FR : 1 234,57 €  
Locale NO : € 1 234,57
```





# ngPluralize

- La directive `ngPluralize` permet d'adapter l'affichage de quantités en fonction de règles de catégorisation
  - Ex : "Aucun contact", "Un contact", "Plusieurs contacts"
- Une forme particulière de pluralisation peut être associée à
  - Une **quantité précise** d'éléments : 0, 1, 2, 27, 42...
  - Une **catégorie** (variable selon les locales)
- Catégories
  - En général, seulement "one" (1) et "other" (2+)
  - Japonais : "other" seulement
  - Irlandais : "one" (1), "two" (2), "few" (3..6), "many" (7..10), "other" (11+)





# ngPluralize

- Syntaxe

```
<ng-pluralize count="<binding/expression>" when="<règles>">
</ng-pluralize>
```

- Règles

- Exprimées sous la forme d'une map
- Correspondance sur des quantités exactes ou des catégories  
(les quantités exactes sont toujours prioritaires)

```
{
  '0'      : 'Aucun contact',
  '42'     : 'Juste le bon nombre de contacts',
  'one'    : 'Un contact',
  'other'  : 'Plusieurs contacts'
}
```





# ngPluralize

- Les chaînes pluralisées peuvent contenir
  - Des bindings AngularJS : {{binding}}
  - Un placeholder pour le nombre d'entités : {}

```
<ng-pluralize
  count="contacts.length"
  when=" {
    '0'      : 'Aucun contact',
    '1'      : 'Un contact : {{contacts[0]}}',
    'other'  : '{{}} contacts, dont
                {{contacts[0]}} et {{contacts[1]}}'
  }"
></ng-pluralize>
```



# Internationalisation des templates

- En-dehors des mécanismes vus dans ce chapitre, AngularJS n'offre pas de mécanisme global d'internationalisation
- Pistes
  - Paramétriser le serveur pour envoyer les ressources dans la bonne locale : templates, images, scripts...
  - Embarquer des dictionnaires dans l'application, interrogés à l'aide de directives AngularJS
- L'internationalisation reste l'un des points complexes des applications déportées côté client



# Formulaires

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- *Formulaires*
- Architectures REST
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS





# Formulaires : Principe général (1/4)

- Principe général de gestion des formulaires
  - Formulaire classique HTML
  - Désactiver le mécanisme de validation natif du navigateur
  - Associer des champs de saisie à des propriétés du scope grâce à la directive `ng-model`
  - Appeler une méthode du scope pour traiter le formulaire en Javascript



# Formulaires : Principe général (2/4)

- Désactivation du mécanisme de validation natif du navigateur
  - La validation sera effectuée par AngularJS
- Attribut `novalidate` sur le formulaire
  - Attribut standard HTML5

```
<form novalidate>  
  ...  
</form>
```



# Formulaires : Principe général (3/4)

- Association des champs de saisie à des propriétés du contrôleur grâce à la directive `ng-model`

```
<input type="text" ng-model="vm.contact.name" />
```

- Association du formulaire lui-même au contrôleur
  - Attribut `name`
  - Utile pour la validation

```
<form name="vm.contactForm" novalidate>  
  ...  
</form>
```





# Formulaires : Principe général (4/4)

- Appel d'une méthode du contrôleur pour traiter le formulaire
  - **ng-submit** sur le formulaire + bouton "submit"

```
<form ng-submit="vm.saveForm()" novalidate>
  <input type="submit"/>
</form>
```

- **ng-click** sur un bouton simple

```
<form novalidate>
  <button ng-click="vm.saveForm()">Save</button>
</form>
```



# Formulaires : Types de champs

- AngularJS gère les types de champs de saisie habituels
  - `input[text]`, `input[radio]`, `input[checkbox]`
  - `select`
  - `textarea`
- Ainsi que certains des nouveaux types introduits par HTML5
  - `input[email]`, `input[number]`, `input[url]`
- Il est possible de créer ses propres types de champs
  - Directive personnalisée



# Formulaires : Exemple

```
<form novalidate>
  Title :
  <input type="radio" ng-model="vm.contact.title" value="Mr"/> Mr
  <input type="radio" ng-model="vm.contact.title" value="Ms"/> Ms

  Name :
  <input type="text" ng-model="vm.contact.name"/>

  Email :
  <input type="email" ng-model="vm.contact.email"/>

  Website :
  <input type="url" ng-model="vm.contact.website"/>

  Notes :
  <textarea ng-model="vm.contact.notes"></textarea>

  <button ng-click="vm.saveContact()">Save</button>
</form>
```





# Formulaires : Fonctionnement

- Un champ peut posséder un ou plusieurs validateurs
  - Standards ou personnalisés
- Chaque champ dispose également d'une propriété `$error`
  - Map contenant l'état de chaque validateur du champ

```
<form name="vm.contactForm">
  <input type="text"
    name="address"
    ng-model="vm.address"
    required/>
</form>
```

Etat de tous les validateurs du champ ?

```
{{vm.contactForm.address.$error | json}}      // {"required":true}
```

Validateur "required" en erreur ?

```
{{vm.contactForm.address.$error.required}} // true
```





# Validation : Validation des champs

- Un champ peut être rendu obligatoire

- De manière permanente : `required`

```
<input type="text" ng-model="vm.name" required />
```

- De manière conditionnelle : `ng-required=<model>`

```
<input type="checkbox" ng-model="vm.nameRequired"/>
<input ng-required="vm.nameRequired" ng-model="vm.name">
```

- Etat du validateur : `<champ>.$error.required`

```
<input name="address" ng-model="vm.address" required>
<span ng-show="vm.form.address.$error.required">
    Erreur
</span>
```



# Validation : Validation des champs

- Validation de la longueur des chaînes
  - `ng-minlength` et `ng-maxlength`

```
<input type="text" name="address" ng-model="vm.address"  
       ng-minlength="3" ng-maxlength="10" />
```

- Etat du validateur :
  - `<champ>.$error.minlength`
  - `<champ>.$errormaxlength`

```
<input name="street" ng-model="vm.street" ng-maxlength="30">  
<span ng-show="vm.form.street.$errormaxlength">  
  30 lettres max  
</span>
```





# Validation : Validation des champs

- Contrôle du format de la saisie
  - Expression régulième : **ng-pattern**

```
<input type="text" name="phone" ng-model="vm.phone"  
ng-pattern="/^555-(\d){4}$/" />
```

- Etat du validateur : **<champ>.\$error.pattern**

```
<input name="phone" ng-pattern="/^555-(\d){4}$/>  
<span ng-show="vm.form.phone.$error.pattern">Ex: 555-1234</span>
```

# Validation : État du formulaire et des champs



- AngularJS expose 6 propriétés au niveau du formulaire et de chacun des champs de saisie
  - `$valid / $invalid` : Indique si l'élément passe le contrôle des validateurs.
  - `$pristine / $dirty` : Indique si l'utilisateur a altéré l'élément.
    - Un élément est considéré dirty dès qu'il subit une modification, même si la valeur initiale est restaurée ensuite.
  - `$untouched / $touched` : Indique si l'élément a été touché (focus).
- Les classes CSS correspondantes sont appliquées aux éléments
  - `ng-valid`, `ng-invalid`, `ng-pristine`, `ng-dirty`, `ng-untouched`, `ng-touched`



# Validation : État du formulaire et des champs

```
<form name="vm.contactForm" novalidate>
  <input type="text" name="contactName"
    ng-model="vm.contact.name" required />
  <div ng-show="vm.contactForm.$valid">
    Formulaire valide !
  </div>
  <div ng-show="vm.contactForm.contactName.$dirty">
    Champ "name" modifié !
  </div>
</form>
```

Formulaire valide !

Formulaire valide !

Champ "name" modifié !





# Validation : ngMessage

- Facilite la gestion des messages d'erreurs des champs de saisie.
- L'ordre de déclaration est l'ordre l'évaluation, l'évaluation s'arrête à la première erreur.
  - **multiple / ng-messages-multiple** : Affiche toutes les erreurs, pas uniquement la première.

- Dans un fichier séparé du 'core'

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

- Doit être déclaré comme une dépendance.

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



# Validation : ngMessage

- Directive attribut

```
<div ng-show="vm.form.field.$dirty" ng-messages="vm.form.field.$error">
  <span ng-message="required">Required</span>
  <span ng-message="minlength">Too short</span>
  <span ng-message="maxlength">Too long</span>
  <span ng-message="pattern">Respect pattern</span>
</div>
```

- Directive element

```
<ng-messages for="vm.form.field.$error">
  <ng-message when="required">Required</ng-message>
  <ng-message when="minlength">Too short</ng-message>
  <ng-message when="maxlength">Too long</ng-message>
  <ng-message when="pattern">Respect format 555-XXXXXX</ng-message>
</ng-messages>
```





# Architectures REST

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- *Architectures REST*
- Directives
- Filtres
- Authentification
- Au-delà d'AngularJS



# Architectures REST



- **REST** : REpresentational State Transfer
  - Type d'architecture pour systèmes hypermédia distribués
  - Inventé par Roy Fielding en 2000
- Architecture
  - Client / Serveur
  - Stateless : chaque requête au serveur contient toutes les informations nécessaires à son traitement
  - Basée sur la manipulation (création, modification, suppression) de ressources identifiées par leur URI
- Un système basé sur cette architecture est qualifié de **RESTful**

# Architectures REST



- Actions

Ressource	GET	PUT	POST	DELETE
Collection	Récupère la liste des éléments	Remplace la collection entière	Crée une nouvelle entrée dans la collection	Supprime toute la collection
Objet	Récupère les détails de la ressource	Met à jour la ressource	Crée une nouvelle ressource	Supprime la ressource

# Architectures REST



- Avantages
  - Découplage client/serveur
  - Scalabilité du serveur
  - Mise en cache des ressources
- Inconvénients
  - Trafic réseau supplémentaire
  - Gestion de la sécurité plus difficile (code côté client)



# Architectures REST

- AngularJS promeut les architectures REST
  - Applications riches entièrement gérées côté client
  - Manipulation de ressources REST
  - Utilisation possible du stockage local
- 2 mécanismes d'interaction avec le serveur
  - `$http` : requêtes AJAX simples
  - `$resource` : abstraction de ressources REST pour faciliter leur manipulation

# Le service \$http



- Le service `$http` est construit autour de la notion de **promesse**
  - Représente une valeur calculée de manière asynchrone
  - Implémentée par le service AngularJS `$q`  
(inspiré de l'API "Q" de Kris Kowal)
  - Comparable à la classe `Future<T>` en Java
- Avantages
  - Interface plus réactive
  - Possibilité de composer plusieurs opérations asynchrones (difficile avec les callbacks traditionnels)



# Le service \$http

- Le service \$http
  - prend en paramètre une map de configuration
  - renvoie une **promesse**

```
$http({ method: 'GET', url: '/foo' }).then(  
  
  function successCallback(response) {  
    // Appelée lorsque la réponse est disponible  
    // et valide (code HTTP 2xx)  
  },  
  
  function errorCallback(response) {  
    // Appelée en cas d'erreur technique,  
    // ou de réponse négative du serveur  
});
```



# Le service \$http

- Principaux paramètres de configuration de la requête
  - [string] **method** : 'GET', 'POST'...
  - [string] **url** : '/foo'
  - [object] **params** : { foo:'bar' }
  - [object] **headers** : { accept:'text/html' }
- Paramètres des fonctions callback **success** / **error**
  - [string/object] **data** : le corps du message
  - [number] **status** : code de retour HTTP
  - [fonction] **headers** : fonction d'accès aux headers
  - [object] **config** : la configuration utilisée pour la requête



# Le service \$http

- Paramètre des fonctions callback de la fonction **then**
  - Objet composé des propriétés suivantes
    - [string/object] **data** : le corps du message
    - [number] **status** : code de retour HTTP
    - [fonction] **headers** : fonction d'accès aux headers
    - [object] **config** : la configuration utilisée pour la requête
    - [string] **statusText** : statut texte HTTP de la réponse

# Le service \$http



- Exemple : récupération de contacts depuis un contrôleur

```
var vm = this;

vm.contacts = [];
vm.errorMsg = '';

vm.getContacts = function () {

    $http({ method: 'GET', url: '/contacts' }).then(
        function (response) {
            vm.contacts = response.data;
        },
        function (error) {
            vm.contacts = [];
            vm.errorMsg = 'Error ' + error.status;
        }
    );
}
```





# Le service \$http

- Le service \$http propose également une API simplifiée
  - `get(url, [config])`
  - `head(url, [config])`
  - `delete(url, [config])`
  - `jsonp(url, [config])`
  - `post(url, data, [config])`
  - `put(url, data, [config])`

```
$http.get('/contacts').then(function (response) {  
    vm.contacts = response.data;  
});
```



# Le service \$resource

- Syntaxe du service \$resource

```
$resource(url, [params], [actions]);
```

- Paramètres (détaillés plus loin)
  - [string] **url** : l'URL de la ressource REST (obligatoire)
  - [map] **params** : valeurs par défaut pour les bindings d'URL
  - [map] **actions** : actions personnalisées additionnelles



# Le service \$resource

- Le service `$resource` permet d'abstraire et de simplifier l'interaction avec des ressources exposées en REST
  - "Convention over configuration"
  - Basé sur le service `$http`
  - Objet de type "ActiveRecord"
- Installation
  - Inclure le script `angular-resource.js`
  - Déclarer une dépendance vers le module `ngResource`

```
angular.module('myModule', ['ngResource']);
```



# Le service \$resource : Déclaration d'une ressource



- Une ressource se déclare via la méthode `factory` d'un module

```
angular.module('ContactSvc', ['ngResource'])
  .factory('Contact', function ($resource) {
    return $resource('/rest/contacts');
});
```

- Elle peut alors être injectée dans d'autres entités AngularJS
  - Contrôleurs, services, modules, directives...

```
[module].controller('ContactListCtrl', function (Contact) {
  var vm = this;
  vm.contacts = Contact.query();
});
```



# Le service \$resource : Paramètres - URL



- L'URL peut contenir des paramètres de binding
  - Forme `:binding` (ex: `/rest/contacts/:id`)
  - Possibilité d'indiquer des valeurs par défaut (constantes ou extraites des propriétés de l'objet)
  - Les paramètres non associés à un binding sont passés sous forme de paramètres d'URL (`?foo=bar`)

```
return $resource(  
  '/rest/contacts/:group/:id',  
  {  
    group: 'friends', // constante  
    id: '@userId', // propriété "userId" de l'objet  
    foo: 'bar' // paramètre additionnel sans binding  
  }  
);
```





# Le service \$resource : Paramètres - Actions

- Une ressource expose différentes **actions**
- Une action représente une configuration particulière des paramètres d'appel du service \$http
  - `method`, `url`, `params`
- Une action peut être invoquée
  - A partir de la ressource, en lui passant une instance
  - Directement sur une instance particulière  
(l'action est alors prefixée par \$)
- Les actions retournent un résultat synchrone qui sera complété de manière asynchrone afin d'utiliser une notation pratique



# Le service \$resource : Paramètres - Actions

- Chaque ressource AngularJS expose 5 actions préconfigurées permettant la création, la mise à jour, la suppression et la recherche de ses instances
  - `get`, `save`, `remove`, `delete`, `query`
- Paramétrage des actions préconfigurées
  - `url` : l'URL de la ressource
  - `method` :
    - `get`, `query` → GET
    - `remove`, `delete` → DELETE
    - `save` → POST

# Le service \$resource : Paramètres - Actions



```
var vm = this;
vm.contacts = Contact.query(function success(contacts) {
  var firstContact = contacts[0];
  firstContact.$delete();
});
```

- De nouvelles actions peuvent être définies
  - Dernier paramètre de la factory \$resource
  - Possibilité de définir le type de requête HTTP et les paramètres par défaut

```
return $resource('/rest/contacts/:id', { id: '@userId' },
{
  export: { method: 'GET', params: { format: 'pdf' } },
  call: { method: 'GET', params: { where: 'home' } }
);
```





# Le service \$resource : Paramètres - Actions

- A l'appel d'une action (préconfigurée ou personnalisée), il est possible de passer 3 paramètres optionnels :
  - [map] **params** : paramètres d'appel, fusionnés avec les paramètres par défaut de l'action et ceux de la ressource
  - [fonction] **success** : callback de réussite
  - [fonction] **error** : callback d'erreur

```
var marvin = Contact.get({id: 42}); // params

marvin.$delete(function success() {
  // exécuté si la suppression retourne un code 2XX
  console.log("Contact deleted.");
});
```







# Directives

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- *Directives*
- Filtres
- Authentification
- Au-delà d'AngularJS



# Directives : Concept



- Une directive dans AngularJS permet d'ajouter du comportement au DOM
  - Permet d'enrichir le langage HTML
  - Associe du comportement
  - Evite d'avoir à utiliser des sélections comme dans jQuery
  - Peut aller du marqueur informatif au widget très riche
- Il y a 4 moyens de déclencher une directive :

```
<span my-directive="exp"></span>
<my-directive></my-directive>
<span class="my-directive: exp;"></span>
<!-- directive: my-directive exp -->
```

- URL à retenir :
  - <https://docs.angularjs.org/guide/directive>
  - [https://docs.angularjs.org/api/ng/service/\\$compile](https://docs.angularjs.org/api/ng/service/$compile)



# Directives : Directives d'AngularJS

- AngularJS propose par défaut de nombreuses directives
- Elles sont toutes écrites avec la même API que celle qui nous est proposée (le code source peut être consulté)
- C'est l'ensemble de ces directives augmenté du bloc `{}{}` qui forment le langage de template d'Angular
- Directives de base les plus connues :
  - `ngBind`, `ngShow`, `ngRepeat`, `ngClick`, `ngModel`...



# Directives : La directive ngBind

```
<span data-ng-bind="contact.address"></span>
```

```
// github.com/angular/angular.js
//   /blob/master/src/ng/directive/ngBind.js

var ngBindDirective =
  ngDirective(function (scope, element, attr) {
    element.addClass('ng-binding').data('$binding', attr.ngBind);
    scope.$watch(attr.ngBind, function ngBindWatchAction(value) {
      element.text(value == undefined ? '' : value);
    });
  });
});
```



# Directives : API : Les bases

- Se déclare dans un module via la fonction `directive`

```
angular.module('myModule')
  .directive('myDirective1', function () {
    return function (scope, elem, attrs) { ... }
  })
  .directive('myDirective2', function () {
    return { ... directive definition ... }
  })
```



# Directives : API : Les bases

- `<name>` : le nom de la directive sera repris dans son déclencheur. Attention : camelCase dans le JS, spinal-case dans l'HTML
- `<restrict>` : paramètre le type de déclencheur souhaité (Défaut : EA) :
  - balise (E)
  - attribut (A)
  - classe (C)
  - commentaire (M).
- `<template>`, `<templateUrl>` : L'activation de la directive ajoutera le contenu du template dans l'élément du DOM



# Directives : API : Cycle de vie

## AngularJS DOM Compilation

<my-directive>

compile

DOM Content

controller

pre-link

Compile DOM Content

post-link

© Copyright Zenika

# Directives : API : Cycle de vie



- Différentes méthodes de déclaration des étapes du cycle de vie :

```
angular.module('myModule').directive('myDirective1',
  function dirInjection() {
    return {
      controller : function ($scope, $elem, $attrs, otherInjectables) { ... }
      compile : function (elem, attrs) {
        return {
          pre : function (scope, elem, attrs, controller) { ... }
          post : function (scope, elem, attrs, controller) { ... }
        }
      }
    })
    .directive('myDirective2', function dirInjection() {
      return { link : function (scope, elem, attrs) { ... } }
    })
    .directive('myDirective3', function () {
      return function (scope, elem, attrs) { ... }
    });
  
```



# Directives : API : Cycle de vie



- La plupart du code des directives se positionne dans la fonction `link` (`postLink` plus précisément)
  - On y a accès au `scope`, à l'élément et ses attributs
  - Le template est alors déjà compilé et réagira au scope
- On utilise la fonction `compile` quand on veut pouvoir intervenir sur le template avant sa compilation
  - Utiliser un template différent suivant les cas



# Directives : API : Cycle de vie

- Le contrôleur est principalement utilisé pour la collaboration entre plusieurs directives
  - C'est le seul élément qui pourra être partagé entre deux directives en dehors du scope
  - On retrouve un contrôleur d'une autre directive en utilisant le paramètre **require** sur la directive

# Directives : API : Scopes



- Le lien entre la directive et le scope est fondamental et doit être paramétré avec soin
- Par défaut, la directive ne crée pas de scope et fonctionne dans le scope courant, c'est rarement le fonctionnement souhaité
- `scope : true`
  - Crée un scope enfant avec héritage
  - C'est celui qui est utilisé avec `ng-controller`
  - Attention à la directive qui modifie le scope parent par héritage



# Directives : API : Scopes

- `scope : {}`
  - Crée un scope enfant mais isolé
  - Permet à la directive de ne pas avoir d'impact sur le scope
  - Brise l'héritage des scopes si la directive n'est pas « finale »
  - Propose des notations simplifiées : `@`, `=`, `&`



# Directives : API : Scopes

- Notations raccourcies pour manipuler un scope isolé
  - '=' instaure un binding bi-directionnel avec le scope parent
  - '@' copie dans le scope enfant le résultat d'une expression évaluée dans le scope parent
  - '=' ajoute au scope enfant une fonction qui évalue une expression dans le scope parent

# Directives : API : Scopes



```
// HTML
<my-directive my-attr-1="vm.foo"
               my-attr-2="hello {{vm.foo}}"
               my-attr-3="vm.func(myAttr1)">
</my-directive>

// Controller
this.foo = 'bar'
this.func = console.log

// Directive
scope : {
  myAttr1 : '=' // scope.myAttr1 <=> this.foo
  myAttr2 : '@' // scope.myAttr2 = 'hello bar'
  myAttr3 : '&' // scope.myAttr3() => console.log('bar')
},
controller: 'FooController',
controllerAs: 'vm',
bindToController: true // (AngularJS 1.3)
// ou
bindToController: { myAttr1: '=' } // avec scope {} (AngularJS 1.4)
```



# Directives : ngModel



- **ngModel** est une directive très importante dans AngularJS
  - Unifie le fonctionnement du binding bi-directionnel des champs de saisie avec le model
  - Communique avec les directives associées aux champs de saisie pour réaliser le binding
  - Publie un contrôleur très important **ngModelController**
- Réaliser proprement une directive qui traite d'un champ de saisie implique de s'intégrer avec la directive **ngModel**
- Par défaut, **ngModel** collabore avec :
  - input text, checkbox, radio, number, email, url, select, textarea



# Directives : ngModelController

- Pour utiliser le `ngModelController`
  - Déclarer une directive avec « `require : '?ngModel'` »
  - Positionner comme quatrième paramètre de la fonction link l'argument `ngModelController`
- Utiliser le `ngModelController` permet de
  - Se brancher avec le système de validité de formulaire
  - Profiter du système de chaîne de parsers / formatters
- Fonctions à définir ou utiliser : `$render`, `$setPristine`, `$setValidity`, `$setViewValue`
- Propriétés : `$viewValue`, `$modelValue`, `$parsers`, `$formatters`, `$error`, `$pristine`, `$dirty`,  
`$valid`, `$invalid`



# Directives : ngModelController

```
angular.module('customControl', [])
  .directive('contenteditable', function () {
    return { restrict: 'A', require: '?ngModel',
      link: function (scope, element, attrs, ngModel) {
        if (!ngModel) return;
        ngModel.$render = function () {
          element.html(ngModel.$viewValue || '');
        };

        element.on('blur keyup change', function () {
          scope.$apply(read);
        });
        read();
      }

      function read() {
        var html = element.html();
        if( attrs.stripBr && html == '<br>' ) { html = ''; }
        ngModel.$setViewValue(html);
      }
    };
  });
});
```





# Directives : Boite à outils : Scope

- Pour développer une directive, il faut bien connaître l'API mais aussi savoir bien manipuler certains outils d'AngularJS
- La plupart des directives agissent sur le scope soit via un scope lié par héritage soit isolé
- **\$watch**
  - Pour que la directive puisse réagir à la modification du model, il faudra utiliser des **\$watch** sur le scope
  - Attention à limiter au maximum le **\$watch** par égalité
- **\$watchCollection**
  - Similaire à **\$watch** mais s'applique sur un tableau ou sur un objet (suit les références ses attributs au lieu de celle de la racine d'objet).



# Directives : Boite à outils : Scope

- **\$apply**
  - Si la directive écoute des événements hors d'Angular, il faut utiliser des **\$apply**
  - Attention, il ne peut pas y avoir deux **\$apply** dans la même pile d'exécution, cela soulève une erreur
  - Possibilité de remplacer **\$apply** par le service **\$timeout**



# Directives : Boîte à outils : \$parse & \$compile

- Le paramétrage d'une directive se fait par les attributs de l'élément, des outils existent pour les traiter efficacement
- Le service `$parse` permet de
  - Évaluer une expression dans un scope
  - Définir une valeur d'une expression dans un scope
  - Exécuter une expression dans un scope enrichi

```
var context = {  
    user : {  
        name : 'angular'  
    }  
};  
var expression = $parse('user.name');  
expression(context); // --> angular  
expression.assign(context, 'newValue'); // --> newValue
```





# Directives : Boîte à outils : \$parse & \$compile

- Il peut arriver également dans une directive d'avoir à compiler une partie d'HTML manuellement

```
var element = $compile(element.contents())(scope)
```



# Directives : Boîte à outils : angular.element

- Dans les directives, on dispose de l'élément sur lequel est appliquée la directive
- Cet élément est un objet enrichi par jqLite qui ressemble à un élément jQuery avec quelques particularités
  - Toutes les fonctionnalités de jQuery ne sont pas disponibles (consulter la liste dans la documentation)
  - `controller(<name>)` : Récupère le contrôleur courant
  - `injector()` : Récupère l'injecteur courant
  - `scope()` : Récupère le scope courant
- Ces outils peuvent s'avérer utiles pour retrouver le contexte Angular dans un composant « non Angular »
- <http://docs.angularjs.org/api/angular.element>





# Directives : La méthode component()

- Nouveauté AngularJS 1.5 : simplifier la création de directives (wrapper)
- Orientation « composant » : faciliter la migration vers Angular 2
- Avec directive :

```
[module].directive(name, function);
```

- Avec component :

```
[module].component(name, options);
```

```
[module].component(name, {  
    restrict: 'E', //E par défaut,  
    template: '<span>Hello, component!</span>', //ou templateUrl:  
    './component.html',  
    controller: 'ComponentController', //ou 'ComponentController as component'  
    controllerAs: 'component', //argument name par défaut,  
    isolate: true, //true par défaut  
    bindings: {}, //{} par défaut, remplace scope: {}  
    ...  
});
```







# Filtres

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- *Filtres*
- Authentification
- Au-delà d'AngularJS



# Filtres : Concept



- Un filtre dans AngularJS permet de mettre en forme une donnée
  - Conserve un modèle de données propre
  - Pilote la mise en forme des données depuis la vue

```
data | filtre1 | filtre2:param1 | filtre3:param1:param2
```

- Les filtres peuvent s'enchaîner et avoir des paramètres
- AngularJS propose un ensemble de filtres par défaut :
  - currency, date, filter, json, limitTo, lowercase, number, orderBy, uppercase

# Filtres : Filter



- Le filtre **filter** est très important et très modulable
- Il permet de filtrer un tableau de données

```
data | filter:'string'  
data | filter:{prop1:'value', prop2:'value'}
```

- **String** : Retourne les éléments qui contiennent la chaîne
- **Object** : Propriété par propriété, retourne les éléments qui contiennent chaque valeur
- **Function** : La fonction est appelée pour chaque élément de la liste. Si la fonction retourne **true**, l'élément est conservé.

```
data | filter:foo  
  
var vm = this;  
vm.foo = function (element) {  
    return element.indexOf('bar') > -1;  
}
```





# Filtres : Filtre personnalisé (1/2)

- Se déclare dans un module via la fonction `filter`

```
angular.module('myModule')
  .filter('myFilter', function () {
    return function (input, param1, param2) { ... }
  })
```

- S'utilise dans un template

```
<span>{{ data | myFilter:'value1':'value2' }}</span>
```

- S'utilise dans du JS à l'aide de l'injection du service `$filter` ou avec le suffixe `Filter` (`uppercaseFilter`)

```
var vm = this;
function ($filter) {
  vm.value = $filter('myFilter')(data, 'value1', 'value2');
}
function (uppercaseFilter) {
  vm.value = uppercaseFilter(data, 'value1', 'value2');
}
```





# Filtres : Filtre personnalisé (2/2)

- Permet d'utiliser les services d'AngularJS (à injecter)
- Attention à ne pas confondre avec la fonction du filtre `filter`
- Permet d'aller plus loin qu'avec le filtre `filter`
  - Transformation de données simples (pas seulement filtre de listes)
  - Possibilité d'utiliser des paramètres
- Le filtre est accessible dans les templates dès que le module est en dépendance





# Authentification

AK

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- *Authentification*
- Au-delà d'AngularJS





# Authentification sur une SPA

- Approches au niveau du formulaire d'identification
  - SSO : formulaire déporté sur un service différent
  - Page de login : navigation après validation du formulaire
  - Login intégré : formulaire intégré à l'application
- Approches au niveau de la transmission de la session
  - Cookie de session : nécessite du stateful côté serveur
  - Cookie avec credentials : nécessite un bon cryptage avec signature
  - Headers dans chaque requête
  - Passage d'un token en paramètre des requêtes

# Authentification sur une SPA



- Toutes les méthodes ont des avantages et des inconvénients mais certaines nécessités reviennent
- Quelque soit la sécurité mise en oeuvre côté client (JS), il faudra **toujours** implémenter une couche de sécurité côté serveur

# En-têtes HTTP



- Le service `$http` ajoute déjà automatiquement des en-têtes HTTP
- On peut y accéder au moment de la configuration de l'application via `$httpProvider.defaults.headers`
  - On pourra par exemple ajouter un entête sur les requêtes de type GET

```
$httpProvider.defaults.headers.get = { 'My-Header' : 'value' }
```

- On peut aussi y accéder au moment de l'exécution via `$http.defaults.headers`

```
$http.defaults.headers.common = { 'Content-Type' : 'application/xml' }
```

# Intercepteurs TTP



- Les intercepteurs filtrent les requêtes et les réponses de manière synchrone ou asynchrone à l'aide du service `$q`
- Il faut enregistrer les factory d'intercepteurs avec leurs dépendances dans l'array `$httpProvider.interceptors`

```
$httpProvider.interceptors.push(  
  function ($q, dependency1, dependency2) {  
    return {  
      request:  
        function (config) { return config || $q.when(config); },  
      requestError:  
        function (rejection) { return $q.reject(rejection); },  
      response:  
        function (response) { return response || $q.when(response); },  
      responseError:  
        function (rejection) { return $q.reject(rejection); } };  
});
```



# Intercepteurs

## HTTP



- Un exemple d'intercepteur qui stocke localement :
  - Les headers de la dernière réponse OK reçue
  - Le statut HTTP de la dernière réponse KO reçue

```
$httpProvider.interceptors.push(function ($q) {  
  return {  
    response: function (response) {  
      localStorage.setItem('latestHeaders', response.headers);  
      return response || $q.when(response);  
    },  
    responseError: function (rejection) {  
      localStorage.setItem('latestRejectionStatus',  
        rejection.status);  
      return $q.reject(rejection);  
    }  
  };  
});
```







# Au-delà d'AngularJS

# Plan



- Rappels
- AngularJS
- Premiers pas
- Modules
- Contrôleurs et Scopes
- Routeur
- Services et Injection
- Tests
- I18N
- Formulaires
- Architectures REST
- Directives
- Filtres
- Authentification
- *Au-delà d'AngularJS*



# Au-delà d'AngularJS



- AngularJS est un framework remarquable pour structurer une application riche en Javascript
  - Mais ce n'est qu'un composant de l'application totale
- D'autres outils, librairies et frameworks peuvent être utilisés en conjonction pour couvrir d'autres aspects du développement
  - Industrialisation
  - Optimisation des ressources
  - Librairies de composants



# Industrialisation : Grunt



- Grunt, "The JavaScript Task Runner"
- <http://gruntjs.com/>
- Basé sur NodeJS (<http://nodejs.org/>)
- Crée en 2011
- Le couteau suisse du développement web
  - compilation (coffeescript, LESS/SASS, templates...)
  - minification des scripts
  - optimisation des images
  - lancement des tests automatisés



# Industrialisation : Gulp



- Gulp, "The streaming build system"
- <http://gulpjs.com/>
- Basé sur NodeJS (<http://nodejs.org/>)
- Code over configuration
- Crée en 2013
- Le couteau suisse du développement web
  - compilation (coffeescript, LESS/SASS, templates...)
  - minification des scripts
  - optimisation des images
  - lancement des tests automatisés



# Industrialisation : Bower



- Bower : "A package manager for the web"
- <http://bower.io/>
- Basé sur NodeJS et Git
- Gestion des dépendances entre composants front-end
  - Le plus souvent des librairies JavaScript





# Industrialisation : Yeoman



YEOMAN

- <http://yeoman.io/>
- Yeoman est une collection d'outils permettant de gérer le cycle de vie d'une application JavaScript
  - Grunt, Bower, Yo (génération de squelette d'application ("**scaffolding**"))
- Démarrer rapidement une application
- La préparer pour une mise en production
- Gérer ses dépendances
- Prévisualiser l'application localement



# Industrialisation : Yeoman

- Exemple d'utilisation de bout en bout

```
# Initialisation de la structure de l'application  
# basée sur le template angular-seed  
mkdir ngapp  
cd ngapp  
yo angular  
  
# Lancement du serveur node.js local  
grunt server  
  
# Lance les tests avec Karma  
grunt test  
  
# Compilation, optimisation et packaging de l'application  
grunt build
```





# Optimisation des ressources

- Les ressources d'une application d'entreprise doivent être optimisées
  - Réduction de la consommation de bande passante
  - Accélération du chargement de l'application
- Chaque type de ressource dispose de techniques d'optimisation propres
  - JavaScript
  - Feuilles de style
  - Images



# Optimisation du code JavaScript

- Les ressources JavaScript peuvent être optimisées
- Vérification
  - JSLint : <http://jslint.com>
  - JSHint : <http://jshint.com>
- Optimisation et compression
  - UglifyJS : <http://github.com/mishoo/UglifyJS>
  - JSMin : <http://www.crockford.com/javascript/jsmin.html>



# Optimisation des styles CSS

- Les feuilles de style CSS peuvent également bénéficier de certaines optimisations
  - Fusion des feuilles de style
  - Optimisation et déduplication des propriétés CSS
- Optimisation
  - CSSLint : <http://csslint.net>
- Langages alternatifs générant du CSS (support des constantes, mixins, opérations sur les unités...)
  - LESS : <http://lesscss.org>
  - SASS : <http://sass-lang.com>



# Optimisation des images

- Il existe 2 types d'optimisations pour les images
  - Optimiser chaque image
  - Grouper toutes les images en une seule, pour les récupérer en une seule requête HTTP
- Yeoman embarque deux optimiseurs
  - OptiPNG : <http://optipng.sourceforge.net>
  - JPEGTran : <http://jpegclub.org>



# Angular-UI et autres modules

- AngularJS est au centre d'un écosystème en plein essor
  - Composants additionnels sur étagère
  - Intégration avec d'autres projets
- Projet ngModules (registre de composants) : <http://ngmodules.org>
  - Angular-UI : <http://angular-ui.github.io>
    - UI Bootstrap (intégration avec Twitter Bootstrap)
    - UI Router (remplacement de \$routeService)
  - Intégration Google Charts, Google Maps
  - Infiniscroll
  - Intégration MongoDB, Firebase

