AngularJS

Objectifs

- Comprendre la philo, les **forces** et les **faiblesses** d'*AngularJS*
- Pouvoir consommer, assembler et créer des **modules**
- Savoir envoyer, stocker et interroger des **données**
- Partager nos **connaissances**

Agenda

Jour 1	Jour 2	Jour 3
Intro - Install	Services	Directives
/	/	/
Templates / Fitres	Routage	Q/A - Debrief

Plan

- Présentation, Concepts
- Modularisation
- Controleurs / Vues, Templates, Filtres
- Formulaires
- Services
- Scopes
- Routage
- Directives
- Tests, Jasmine, Karma, Protractor

Légende

- **commands**: `npm i github-todos`
- **directive**: `ng-show`
- **method**: `.factory()`
- **property**: `templateUrl`
- **service**: `$http`

HTML4 : dividite aigüe

```
<div>
  Mon espace personel
  <ul>
    <li><a href="/about">A propos</a></li>
    <li><a href="/contact">Ecrivez moi!</a></li>
  </ul>
</div>
<div>
  <div>
    <h1>AngularJS</h1>
    <img src="ng-logo.jpg" alt="logo">
  </div>
  <div>
    <h3>Recherche</h3>
    <input name="q">
  </div>
</div>
<div>
  Page créée par Lilian
</div>
```

HTML5 : Sémantique !

```
<header>
  Mon espace personel
  <nav><ul>
    <li><a href="/about">A propos</a></li>
    <li><a href="/contact">Ecrivez moi!</a></li>
  </ul></nav>
</header>
<main>
  <article>
    <h1 draggable>AngularJS</h1>
    <figure>
        <img src="ng-logo.jpg" srcset="logo-medium.jpg 1000w, logo-large.jp
    </figure>
  </article>
  <aside>
    <h3>Recherche</h3>
    <input name="q" placeholder="mots clés" required>
  </aside>
</main>
<footer itemscope itemtype="http://schema.org/Person">
  Page créée par <span itemprop="givenName">Lilian</span>
</footer>
```

Pages web dynamiques

Avant (2006)

- jQuery, MooTools & co
- On manipule directement le DOM

Après (> 2010)

- Frameworks : Backbone, Ember…
- Juin 2012 : *AngularJS* 1.0.0
  - Two-way data-binding
  - On ne manipule plus le DOM
  - Modularité (composants "plug 'n' play")
  - Testabilité

Outils et écosystème

Les outils classiques du développeur web :

- Un **éditeur** de code efficace (SublimeText, Atom, vim…)
- Un **linteur** (ESLint, JSHint, editorconfig)
- Des navigateurs modernes et leurs **consoles** de débuggage (Firefox, Chrome)
- Un **terminal** efficace (Windows ?)
- npm (depuis node.js ou io.js)
- Un système de **packaging** front (bower, browserify, component…)

Outils et écosystème

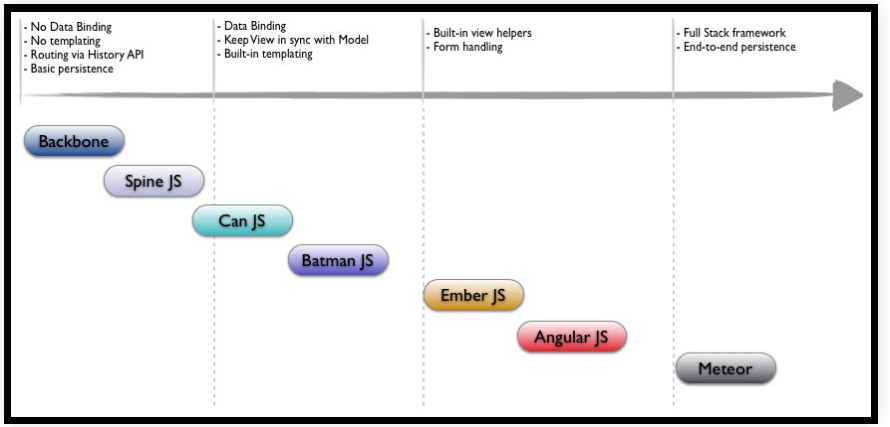
Les outils du développeurs *AngularJS* :

- **Yeoman**: générateurs specifiques ?
- **Grunt / Gulp**: transformeurs dédiés comme *ng-annotate*
- **Karma**: lanceur de tests unitaires
- **Protractor**: lanceur de tests fonctionnels
- **Batarang**: extension Chrome (pas top)
- **ngInspector**: extension navigateur

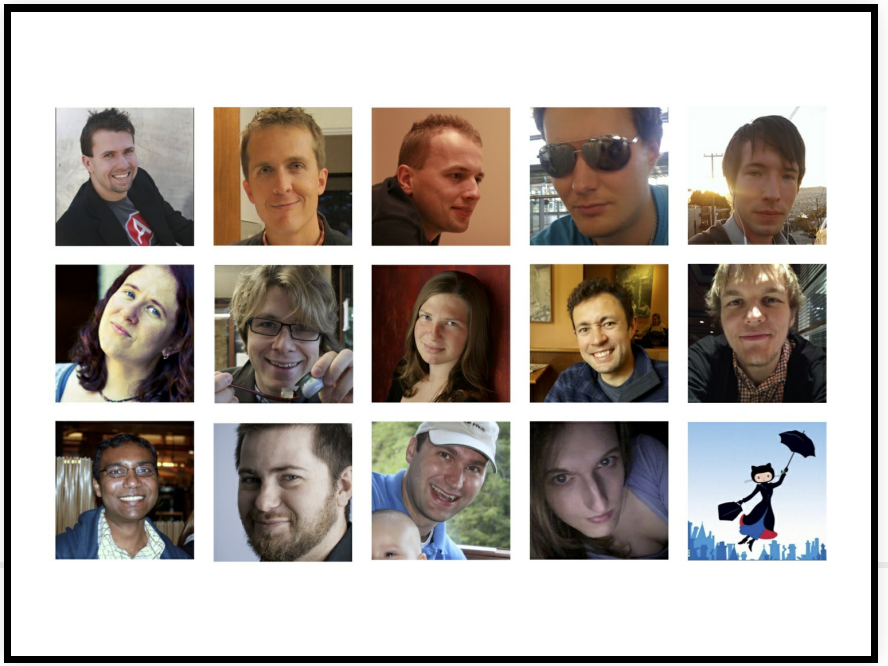
Alternatives à AngularJS

- **Backbone**
  - Léger, rapide
  - Aucune structure prédéfinie, peu de fonctionnalités
- **Ember**
  - Plus lourd
  - "Convention over configuration"
  - Intégration REST
- **React**
  - Virtual DOM : très efficace
  - Limité à la vue
  - Pas de structure prédéfinie (ex. Flux)

Couverture des frameworks



L'équipe



- Miško Hevery
- Igor Minár
- Vojta Jína
- Mary Poppins
- La communauté !

- Aura & légitimité...
- ...mais aussi tendance à tuer ses projets

Versions

Vers une stabilisation avec des cycles :

- Annoncé dès 2009 par **Google**
- 1.0 - 13 Juin 2012 - Temporal Domination
- 1.1 - 31 Aout 2012 - Increase Gravatas
- 1.2 - 08 Nov 2013 - Timely delivery
- 1.3 - 22 Oct 2014 - Superluminal Nudge (ng-europe)

Le futur (plus ou moins) proche

1.4.0-rc.0 smooth-unwinding (2015-04-10)

- **Router**- nouveau routeur pour *AngularJS* 1 and 2
- **I18N** - meilleur support de l'internationalisation
- **Forms** - une usage simplifié du parsage/formatage/validation des formulaires
- **HTTP** - ajouts à `$http`, comme la serialisation, le parsage JSON, DSL de test
- **Parser** - performances améliorées pour le service `$parse`
- **Documentation** - nouveau look utilisant Material Design

La fameuse 2.0...

- Performance
- Intégration WebComponents
- Priorité au mobile

Changements profonds :

- En **AtScript** : apport des annotations (optionnel)
- Système d'injection de dépendances (kill the magic)
- Component Directive / Decorator Directive
- Templating avec data-binding
- Nouveau système de routing
- RIP `$scope`, `ng-controller`

AngularDart

Dart

- Version compilée en JS ou tournant sous Dartium
- Labo experimental
- Source d'inspiration

Les Web Components

Des standards W3C pour le développement Web modulaire :

- **HTML imports** (importer du contenu dans sa page via `<link>`)
- **Templates** (le tag `<template>`, plus efficace que `<script type="text/html">`)
- **Shadow DOM** (portions isolées de DOM)
- **Tags personnalisés** (`<google-map lat="42" long="69"></google-map>`)

Cf. directives AngularJS...

Le secret d'AngularJS

NE MANIPULEZ QUE LES DONNÉES !

On ne touche (sauf rarissime exception) jamais au DOM, mais seulement aux données (le modèle) et on laisse *AngularJS* faire les mises à jour de vues

Compatibilité navigateurs

- *AngularJS* est compatible avec les navigateurs modernes et IE >= 9
- *AngularJS* 1.2 supporte encore IE 8, mais pas de corrections spécifiques

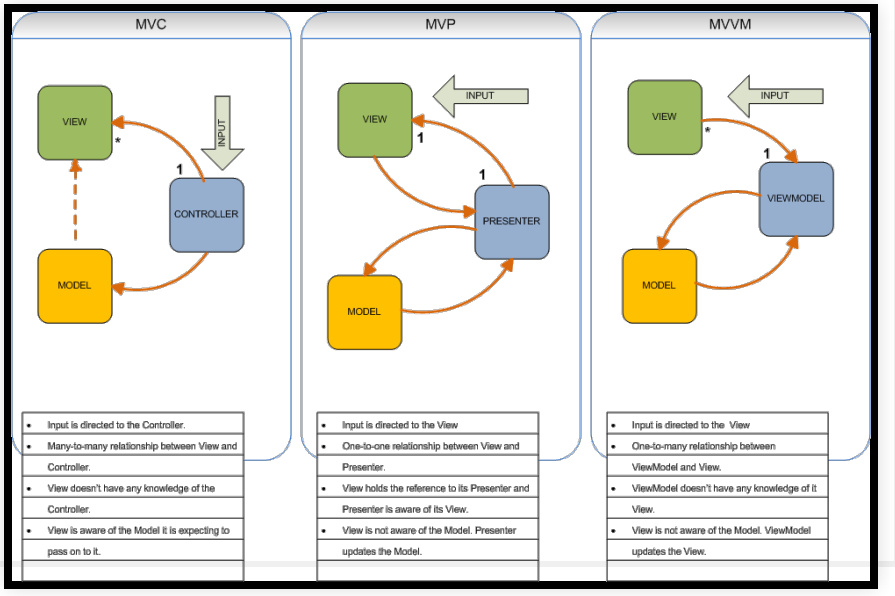
Helpers

- Des helpers globaux sous la forme `angular.helper()` lissent les différences entre les navigateurs sur l'API JavaScript
- lodash pour manipuler les objets et collections

Quelques Helpers

- `angular.element(string)` alias de jQuery si disponible, sinon "jqLite"
- `angular.copy(obj)` cloner un objet
- `angular.extend(dst, src1, src2…)` copier les propriétés vers l'objet cible
- `angular.isArray`, `isDate`, `isFunction`, `isNumber`, `isDefined`... pour tester des valeurs
- `angular.forEach` boucler aussi sur des objets
- `angular.toJSON` ignore les propriétés commençant pas `$$` (norme interne à *AngularJS*)

MVC, MVP, MVVM



MVC (Model, View, Controller)

- Le contrôleur reçoit les entrées et génère des vues en fonction de ces entrées et des données du modèle
- Adapté à l'ancien modèle serveur : une requête = une nouvelle page, pas d'interactions dans les deux sens

MVVM (Model, View, ViewModel)

- La vue devient le point d'entrée, elle passe la main à un ViewModel qui se charge de calculer les nouvelles données pour que la vue se mette à jour

Single Page Application

Application web mono-page :

- Une seule page transmise par le serveur
- Des mises à jour du contenu au lieu de charger l'integralité
- Interactions serveur réduites au minimum : API d'accès aux données

Single Page Application

Le code front reste en vie longtemps, sa qualité doit être supérieure !

- Modularisation
- Design patterns
- Interactions client/serveur standardisées

Les promesses

- Callbacks : on passe à l'action asyncrhone la fonction qu'elle devra exécuter une fois que le résultat sera disponible

Les promesses inversent la responsabilité :

- Promesse : on récupère un objet représentant le futur résultat

API

- `.then(callback)` exécuté quand la promesse est tenue
- `.catch(callback)` exécuté quand la promesse est cassée

Les promesses

- Chaînage : `.then` et `.catch` retournent une promesse, représentant le résultat du callback
- Erreurs : tant qu'on n'appelle pas `.catch`, le statut "cassé" de la promesse est propagé de `.then` en `.then`

Pourquoi ?

- Organisation du code asynchrone plus simple
- Permet de manipuler le résultat (stocker, passer en paramètre...)
- Meilleure gestion des erreurs

Les promesses

```
var newStats = Promise.all([getPlayers(), getGames()])
.then(buildStats)

var oldStats = getStats()

Promise.all([oldStats, newStats])
.catch(showError)
.then(showDiff)
```

Implémentations

- Bluebird, Q, Implémentations natives...
- Spécs: Promise/A+

Les promesses dans AngularJS

- Service `$q`
- Constructeur standard : `$q(function (resolve, reject) { … })`
- API standard :
  - `$q.all`
  - `promise.then`
  - `promise.catch`

N'est pas une implémentation complète de la spec, mais suffisante.

Programmation impérative vs. déclarative

- Impératif : on décrit les opérations à effectuer pour changer d'état (le mode de programmation le plus habituel)
- Déclaratif : on décrit un état, les transitions sont à la charge du système (ex. HTML, SQL)

AngularJS ~= déclaratif

```
<a href="#" ng-click="show=!show">Hide / Show</a>
<div ng-show="show">Some cool content</div>
```

Attacher des données à la vue

```
<body ng-app>
  <h1>Bonjour {{ info.name }}</h1>
  <input ng-model="info.name">
</body>
```

Comment ?

- Dirty checking
- `$digest()` est appelé quand une donnée est potentiellement changée
- Il "digère" les données et en ressort (sic) la vue à jour

Injection de dépendances

- Description des composants d'un côté
- Chargement au besoin via un mécanisme simplifié

Dans AngularJS

```
angular.module("…").composant("nom", …);
```

```
// Implicite : par nom d'argument
function foo (nom) { … }

// Explicite via un tableau
["nom", function bar (param) { … }

// Explicte via $inject
composant.$inject = ["nom"];
```

REST (REpresentational State Transfer)

- Architecture client-serveur sans état
- Ressources identifiées de manière unique

HTTP

- Les actions sont représentées par un verbe HTTP (GET, POST, PUT…)
- Les ressources sont identifiées par leur *URL* unique
- Headers de cache

Ressources HTTP

- Le format de communication est généralement *XML* ou *JSON*
- Des entêtes ou des propriétés de la ressource définissent les actions disponibles (Hypermedia)

RESTful (et pragmatisme)

- Les API sont rarement "RESTful" (HATEOAS, Hypermedia)…
- … mais c'est rarement nécessaire, surtout pour une API non publique

API habituel

Verbe + URL	Action
GET /items	Récupération de la collection
GET /items/{id}	Récupération d'un élément
POST /items	Création d'un élément
DELETE /items/{id}	Suppression d'un élément
PUT /items/{id}	Remplacement et/ou mise à jour d'un élément
Actions moins classiques...	
POST /items/{id}	Création d'un élément
PATCH /items/{id}	Mise à jour exclusive d'un élément
PUT /items	Écrasement de la collection
DELETE /items	Suppression de la collection

Pipes

Les systèmes Unix disposent d'un système de *pipeline* pour chaîner des processus : on envoie la sortie d'un processus en entrée d'un autre processus

```
cat lignes.txt | sort | uniq | grep "42"
```

Cette mécanique se retrouve dans les filtres *AngularJS* qui reprennent la syntaxe `|` :

```
{{ lignes | orderBy | deDuplicate | filter:"42" }}
```

Routage

Le principe général du routage est de prendre en entrée une URL, et de produire en sortie une app

Dans le cas d'*AngularJS*, une app est composée d'un **template** et d'un **contrôleur**, on définira donc une route par :

- son URL
- son template
- son contrôleur
- d'autres propriétés éventuelles (services à injecter, contraintes sur les données, etc...)

Module AngularJS

Un module *AngularJS* représente un package réutilisable, ou en tous cas un ensemble de composants regroupés

L'app principale est un module *AngularJS* définissant un système de routage et/ou exposant des contrôleurs qui pourront être référencés dans la vue

Déclarer un module

```
angular.module("univers", [
  // modules tiers dont dépend ce module
  "vie"
])
```

Composants d'un module AngularJS

Un composant est une partie d'un module :

- **value, constant**: une valeur directe
- **provider, service, factory, decorator**: un objet complexe offrant un service
- **controller**: une fonction gérant la vue associée
- **directive**: un composant de la vue
- **filter**: une fonction de transformation utilisable directement dans la vue

Un composant est décrit par un *nom unique* au sein du module, qui sera utilisé pour l'injection de dépendances.

Déclarer un composant

```
// Le composant "laReponse" dans le module "univers"
angular.module("univers")
.value("laReponse", 42)
```

Getter vs Setter

Attention, le comportement de `.module()` est altéré par son **arité**.

```
// Cette fois ci on passe un tableau vide en deuxième argument
// -> setter
angular.module("univers", [])
.value("laReponse", 42)
```

Façade

Les fonctions `angular.module('foo').componentType()` sont en réalité des raccourcis :

- animation -> `$animateProvider.register()`
- controller -> `$controllerProvider.register()`
- directive -> `$compileProvider.directive()`
- filter -> `$filterProvider.register()`
- provider, factory, service, value, constant et decorator -> `$provide.*()`

Cycle de vie d'un module

Un module ou une app *AngularJS* passe par deux phases :

- **Configuration** : enregistrement des composants déclarés (on n'a accès qu'à certains composants à ce stade)
- **Exécution** : exécution à proprement parler (la vue est alors traitée, le routing résolu...)

```
angular.module("…")

// Phase de config
.config(function (serviceProvider, constante) {
  // …
})

.run(function (service, valeur) {
  // …
})
```

Dépendances dans AngularJS

Injection de dépendances : modules et composants

```
// Un premier module…
angular.module("module1", [])
// …déclare un composant
.value("valeur1", 42)
// → injection d'une dépendance interne au module
.factory("service", function (valeur1) {})

// Un deuxième module…
angular.module("module2", [
  "module1" // ← chargement d'un module tiers
  // tous ses composants sont alors disponibles comme dépendances
])
.value("valeur2", "hello world")
// → injection des dépendances
.factory("service", function (valeur1, valeur2) {})
```

De manière automatique, le service `$injector` se charge du branchement pour nous. On peut être amener à le manipuler directement lors de l'écriture des tests.

Modules & Namespaces

Le problème

- Pas de "namespace" : conflits

```
angular.module("module1", [])
.value("valeur", 42)

angular.module("module2", ["module1"])
.value("valeur", "hello world")
.factory("service", function (valeur) {
  // Qui est "valeur" ???
})
```

Modules & Namespaces

La solution

Préfixer les noms de ses composants

```
angular.module("module1", [])
.value("mod1Valeur", 42)

angular.module("module2", ["module1"])
.value("mod2Valeur", "hello world")
.factory("service", function (mod1Valeur) {
  // Référence explicite
})
```

Mais on perd en simplicité et en lisibilité

DI & Minification

Le problème

- Les noms des composants s'invitent comme noms de variable
- Les minificateurs vont casser l'inférence!

```
angular.module("mod", [])
.value("ncTmBhPrefixOverflowValeur", "hello world")
.factory("service", function (ncTmBhPrefixOverflowValeur) {
  console.log(ncTmBhPrefixOverflowValeur)
});

// Minification…
a.b("mod", [])
.c("ncTmBhPrefixOverflowValeur", "hello world")
.d("service", function (e) { // ← "e" ? Composant inconnu !
  f.g(e)
});
```

Minification

La solution

Dépendances explicites

```
// On déclare les dépendances explicitement
.factory("service", [
  "ncTmBhPrefixOverflowValeur",
  function (valeur) { // Attention à l'ordre, ça doit matcher !
    console.log(valeur)
  }
]);

// Minification…
.d("service", [
  "ncTmBhPrefixOverflowValeur",
  function (e) { // ← OSEF des noms, l'ordre est préservé
    f.g(e)
  }
]);
```

- Une troisième possibilité est d'annoter le composant avec une propriété `$inject`
- Cette opération peut être réalisée automatiquement durant le build par `ng-annotate`
- Sinon la directive `ng-strict` peut jouer le rôle de garde de fou

Chargement d'un module tiers

Il suffit :

- de charger le code du module
- de dépendre du module dans la déclaration de notre app
- tous les composants disponibles sont injectables dans notre app !

```
<script src="/app/vendor/module-tiers.js"></script>
<script>
  angular.module("app", ["module-tiers"])
  .factory("service", function (composantTiers) {
    // …
  });
</script>
```

Dépendance vs injection

Attention à ne pas confondre ces 2 mécanismes :

- les dépendances s'opèrent entre **modules**

```
angular.module('app', [/* ex: ngRoute, ngMessage etc... */]);
```

- les injections se réalisent entre **composants**

```
module.service('AlbumSvc', function (/* ex: $http, $q */) {});
```

Architecture d'une app AngularJS

- Des modules tiers
- Des composants internes (services d'accès aux données, directives)
- Des templates HTML
- Du CSS, des images...

Comment ranger tout ça ? On va généralement regrouper les composants par "famille" au sein d'un module la représentante

2 Organisations

- **Modularisation technique** : les éléments sont regroupés par type (service, directives, templates, styles, images...)
- **Modularisation fonctionnelle** : les éléments sont regroupés par fonctionnalité (user, phonebook, forum...)

La modularisation technique évite de se poser des questions (posez-vous la bonne question :))

Modularisation fonctionnelle

- un dossier par module fonctionnel
- un dossier pour les fonctions partagés entre modules (services de données communes)
- parfois un dossier "vendors" pour isoler les librairies tierces

```
app/

  module1/
    controllers/
      controller1.js
    directives/
      directive1.js
    filters/
      filtre1.js
    services/
      service1.js
    views/
      template1.html
    styles/
    images/
    index.js

  module2/
    controllers/
    filters/
    views/

  shared/
    controllers/
    directives/
    filters/
    services/
    views/
    styles/
    images/
    index.js
```

Modularisation technique

- un dossier par type de fichiers

```
app/
  js/
    controllers/
      controller1.js
      controller2.js
    directives/
      directive1.js
    filters/
      filtre1.js
      filtre2.js
    services/
      service1.js
    app.js

  views/
    template1.html
    template2.html

  styles/
    main.css

  images/
    logo.png
```

Déclaration d'une app : ng-app

L'attribut `ng-app` placé sur `<body>` ou `<html>` permet d'initialiser l'app (`$rootElement`)

La valeur de l'attribut est le nom de de l'app (module) à démarrer

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

Cette directive ne peut être utilisée **qu'une seule fois** par document. Dans les rares cas où plusieurs apps doivent être instanciées en même temps, il faut le faire manuellement avec `angular.bootstrap()`

Contrôleur

Un **contrôleur** gère les interactions entre la vue et le modèle.

C'est l'endroit dans lequel la logique de présentation peut être déclarée pour que l'affichage reste en phase avec le modèle de données

Un contrôleur est attaché à un module via la méthode `.controller()`. Celle-ci accepte 2 arguments : le nom du contrôleur, qui servira de référence dans toute l'app et une fonction contenant sa logique.

```
angular.module("app", [])
.controller("MyFirstController", function () {
    // body of the controller
})
```

La bonne pratique veut que l'on nomme le contrôleur avec ces 2 règles :

- PascalCase
- Terminé par "Controller" (attention 2 "l" en anglais)

ng-controller

Un contrôleur est en charge de faire le lien entre la vue et le modèle. En ajoutant dans le code HTML, sur une balise voulue, l'attribut `ng-controller`, on déclare qu'un contrôleur est en charge de cette portion du DOM. Cet attribut attend un paramètre qui est le nom du contrôleur.

```
<body ng-app="app">
  <div ng-controller="MyFirstController"></div>
</body>
```

Il est possible d'utiliser le même contrôleur à plusieurs endroits dans le DOM. Dans ce cas, une instance différente sera utilisée pour chaque déclaration `ng-controller`

Encapsulation des contrôleurs

Si plusieurs déclarations de contrôleurs sont sur un même sous-arbre, les contrôleurs se **partagent** la responsabilité de la zone du DOM qu'ils ont en commun.

```
<div ng-controller="FirstController">
  L'affichage dans ce div est "contrôlé" par FirstController
  <p ng-controller="SecondController">
    L'affichage dans ce p est "contrôlé" par FirstController et SecondContr
  </p>
</div>
```

Utilisation du contrôleur dans la vue

Afin d'utiliser le contrôleur dans la vue, l'instance du contrôleur est nommée. On utilise pour cela une syntaxe particulière de `ng-controller`

```
<div ng-controller="MyFirstController as ctrl"></div>
```

Cet exemple doit se lire comme cela : "instancie le contrôleur **MyFirstController** et nomme cette instance **ctrl**"

$parent en cascade

```
function FirstController($scope) {
  $scope.title = 'Title 1';
}

function SecondController($scope) {
  $scope.title = 'Title 2';
}

app.controller('FirstController', FirstController);
app.controller('SecondController', SecondController);
```

```
<div ng-controller="FirstController">
  {{ title }}
  <div ng-controller="SecondController">
    {{ title }}
    {{ $parent.title }}
  </div>
</div>
```

$parent.$parent.$parent == mauvaise pratique

dot access

```
function FirstController($scope) {
  $scope.firstModel = {
    title: 'Title 1'
  };
}
function SecondController($scope) {
  $scope.secondModel = {
    title: 'Title 2'
  };
}
```

```
<div ng-controller="FirstController">
  {{ firstModel.title }}
  <div ng-controller="SecondController">
    {{ secondModel.title }}
  </div>
</div>
```

this vs $scope

Avec la syntaxe `controller as`, tout ce qui est assigné sur l'instance du contrôleur est accessible dans la partie de la vue sur laquelle elle est positionnée.

```
angular.module("app", [])
.controller("MyFirstController", function () {
  this.name = "Thomas";
})
```

```
<body ng-app="app">
  <div ng-controller="MyFirstController as ctrl">
    Bonjour, {{ ctrl.name }}
  </div>
</body>
```

Souvent **vm** (pour ViewModel) est préféré à **ctrl**. Voir même **myFirst** qui permet de savoir clairement à quelle entité on fait référence et évite les collisions. Dans le contrôleur, il est pratique d'utiliser `var ctrl = this;` afin d'avoir la même syntaxe dans le contrôleur et dans la vue.

Partage de données

Ce qui est assigné sur l'instance du contrôleur est disponible dans la vue, quelque soit type de la propriété :

- Scalaire (string, number)
- Objet
- Array

```
angular.module("app", [])
.controller("UserController", function () {
  var user = this;

  user.name = "Thomas"; // string
  user.profile = { // object
    email: "tmoyse@gmail.com"
  };
  user.friends = [ // array
    "Nicolas",
    "Bruno"
  ]
})
```

Partage de méthodes

Ca marche aussi pour les méthodes :

```
angular.module("app", [])
.controller("UserController", function () {
  var user = this;

  user.getName = function () {
    return "Thomas";
  };
})
```

```
<body ng-app="app">
  <div ng-controller="UserController as user">
    Bonjour, {{ user.getName() }}
  </div>
</body>
```

Templates

AngularJS est en grande partie déclaratif. Une part importante de la logique de présentation peut être définie via des déclarations dans le code HTML.

Pour ce faire les templates sont pourvus de 2 mécanismes :

- Les directives
- Les expressions

Directives

Les **directives** sont le mécanisme proposé par *AngularJS* pour augmenter HTML et lui adjoindre des extensions liées à votre app.

Par exemple, on peut imaginer un nouvel élément `<user>` qui serait en charge d'afficher une div contenant toutes les informations d'un utilisateur donné.

```
<user id="42"></user>
```

ng-app ng-controller

Nous avons déjà rencontré ces 2 directives. Elles permettent respectivement d'initialiser un module pour en faire une app et d'instancier un contrôleur.

ng-show ng-hide ng-if

Ces directives permettent d'afficher ou cacher une portion du DOM.

```
<div ng-show="condition_show">
  Lorem ipsum dolor sit amet, consectetur adipisicing elit.</div>
<div ng-hide="condition_hide">
  Lorem ipsum dolor sit amet, consectetur adipisicing elit.</div>
<div ng-if="condition_if">
  Lorem ipsum dolor sit amet, consectetur adipisicing elit.</div>
```

- La 1ère div est affichée si **condition_show** est évaluée à **truthy**, cachée sinon
- La 2nd div est cachée si **condition_hide** est évaluée à **truthy**, affichée sinon
- La 3ème div est affichée si **condition_if** est évaluée à **truthy**, cachée sinon

ng-show vs ng-if

La différence entre `ng-show` et `ng-if` repose sur le fait que `ng-if` retire du DOM l'élement si l'évaluation est **falsy**. `ng-hide` se contente de cacher l'élement.

Attention donc au nouveau scope créé par `ng-if`, particulièrement son héritage.

ng-repeat

`ng-repeat` permet de boucler sur les éléments d'une collection. Elle recopie autant de fois que nécessaire l'élément du DOM sur lequel elle est positionnée.

`ng-repeat` offre de nombreuses options. Parmi celles-ci, retenons la clé `$index` donnant l'index du parcours.

```
angular.module("app", [])
.controller("UserController", function () {
  var user = this;
  user.friends = [{ name: "Bruno" }, { name: "Nicolas" }, { name: "Lilian"
})
```

```
<body ng-app="app">
  <div ng-controller="UserController as user">
    <ul>
      <li ng-repeat="friend in user.friends">{{ $index + 1 }} - {{ friend.n
    </ul>
  </div>
</body>
```

track by

Il est possible de parcourir un objet ou un tableau (boucle sur les propiétés énumérables). Mais `ng-repeat` ne supporte pas d'avoir 2 fois le même élément dans la collection.

Pour parcourir des objets avec plusieurs fois le même élément, il faut utiliser le mot clé `track by $index`

ng-repeat-start et ng-repeat-end

`ng-repeat` ne permet pas de répeter plusieurs éléments du DOM. Pour ce faire, il existe `ng-repeat-start` et `ng-repeat-end`

```
angular.module("app", [])
.controller("ComputerController", function () {
  var computer = this;
  computer.editors = [{ term: "Sublime", def: "Shiny and quick" }, { term:
})
```

```
<body ng-app="app">
  <dl ng-controller="ComputerController as computer">
    <dt ng-repeat-start="editor in computer.editors">{{ $index + 1 }} - {{
    <dd ng-repeat-end>{{ editor.def }}</dd>
  </dl>
</body>
```

ng-click

`ng-click` éxécute la fonction qui lui est passée en argument.

Il est possible de passer des paramètres à la méthode éxécutée.

```
angular.module("app", [])
.controller("UserController", function () {
  var user = this;
  user.action = function (msg) {
    console.log(msg || "Vous avez cliqué");
  };
})
```

```
<body ng-app="app"><div ng-controller="UserController as user">
    <button ng-click="action()" type="button">Clic me !</button>
    <button ng-click="action('Vous avez encore cliqué')" type="button">Clic
</div></body>
```

En fait l'*expression* `action()` est évaluée.

Autres actions

D'autres directives agissent de la même manière :

- `ng-focus` / `ng-blur`
- `ng-change`
- `ng-copy`
- `ng-dblclick` / `ng-mousemove`
- `ng-keypress`
- `ng-keydown` / `ng-keyup`
- `ng-submit`...

ng-class

`ng-class` permet d'ajouter des classes CSS à l'objet du DOM sur lequel elle est positionnée.

Elle peut s'utiliser comme :

- **une chaîne de caractères** : la (ou les) classes correspondantes sont ajoutées.
- **un tableau de chaînes** : les classes correspondantes sont ajoutées.
- **un objet** : le classes correspondant aux clés sont ajoutées si leur valeur est **truthy**.

ng-class

```
angular.module("app", [])
.controller("StupidController", function () {
  var stupid = this;
  stupid.status = false;
  stupid.toggleStatus = function () {
    status = !status;
  };
})
```

```
<body ng-app="app">
  <div ng-controller="StupidController as stupid">
    <div ng-class="'bold strike'">Option 1</div>
    <div ng-class="['bold', 'strike']">Option 2</div>
    <div ng-class="{ bold: stupid.status, strike: !stupid.status }">Option
    <button ng-click="stupid.toggleStatus()" type="button">Clic me !</butto
  </div>
</body>
```

ng-include

Il est possible (et même conseillé) de découper son code HTML. `ng-include` permet d'utiliser un même template à plusieurs endroits : elle ajoute le HTML venu d'une autre source à l'endroit où elle est utilisée.

La récupération du code à inclure se fait par 3 mécanismes :

- En cherchant dans le cache de *AngularJS*
- En utilisant une balise
- Via un appel `XMLHttpRequest`

ng-include

`ng-include` peut s'utiliser en attribut ou comme élément.

```
<html>
<script src="angular.js"></script>
<script type="text/ng-template" id="one">
  <div>First template</div>
</script>
<body ng-app>
  Inclusion via script
  <ng-include src="'one'"></ng-include>
  Inclusion via appel XMLHttpRequest
  <div ng-include="'two.html'"></div>
</body>
</html>
```

l'attribut `src` (ou `ng-include`) attend une expression à évaluer comme une chaîne de caractères. C'est pour cela qu'il ne faut pas oublier l'usage des simples quotes à l'intérieur des doubles quotes.

ng-src et ng-href

Si vous souhaitez utiliser une variable comme source d'une image ou comme lien, il est recommandé d'utiliser `ng-href` et `ng-src`. Celles-ci évitent l'affichage intempestif d'une resource fausse, voire même une requête inutile.

```
<a ng-href="{{ user.web }}">Go</a>
Le lien ne sera cliquable que lorsque user.web aura une valeur.
<img ng-src="{{ user.gravatar }}">
l'image ne sera affichée que lorsque user.gravatar aura une valeur.
```

ng-bind

Lorsque vous utilisez `{{ }}`, avant le démarrage de votre app *AngularJS*, le navigateur affiche les accolades et le code non interprété. Pour éviter cela, il est possible d'utiliser `ng-bind`

```
<span>{{ user.name }}</span>
<span ng-bind="user.name"></span>
```

Note : le résultat affiché par `ng-bind` est nettoyé de balises HTML. Pour afficher du HTML, il faut utiliser la directive `ng-bind-html` coijointement avec le `$sanitize`

`ng-non-bindable` indique explicitement à *AngularJS* de ne pas tenter le parsage de l'expression.

Du JavaScript dans les templates

En plus des directives, *AngularJS* propose un autre mécanisme pour afficher des données dynamiques dans un template : **les expressions**.

Une expression est un bout de JavaScript qui est évalué et éventuellement affiché. Une expression peut être utilisée pour de l'affichage ou en combinaison avec une directive.

```
<div ng-show="user.age >= 18">Contenu choquant</div>
<div ng-hide="user.age < 18">
  <p>Vous avez seulement {{ user.age }} ans !</p>
  <p>Revenez dans {{ 18 - user.age }} ans</p>
</div>
```

Limites aux expressions

Il est possible d'utiliser beaucoup de choses du JavaScript dans une expression :

- `1 + 2`
- `var1 || var2`
- `mafonction()`

Toutefois, il faut tenir compte des différences suivantes :

- Les variables sont liées au Scope (cf. chapitres suivants)
- Pas d'erreur sur `undefined` ou `null`
- Pas de structure de contrôle (boucles, if…)

Pour accéder à une variable globale (ex `md5()`), il faut venir l'attacher au scope courant au préalable.

One time data binding

En ajoutant `::` devant une expression, elle n'est évaluée qu'une seule fois. Pratique pour i18n ou les dates.

```
<p>Salut {{::user.name}} !</p>
```

Services

- `$parse`: lecture / écriture de **1** expression
- `$interpolate`: lecture seule - strings avec **plusieurs** expressions
- `$compile`: HTML strings -> DOM

$parse

Le service `$parse` assure que l'évaluation des expressions est **sandboxée**.

Ceci afin d'éviter des problèmes de **sécurité**.

Au sein de ce code critique (et complexe), on retrouve un couple **lexer** / parser responsable de génerer un **AST**.

$parse - get

```
// une fonction de lecture acceptant context et/ou locals
var getter = $parse('book.title');

// locals a la priorité sur context (un $scope le plus souvent)
var context = {book: {title: '1984'}};
var locals = {book: {title: 'Germinal'}};

getter(context) === '1984'; // true
getter(context, locals) === 'Germinal'; // true
```

$parse - set

Seulement disponible si l'expression peut être assignée.

```
var getter = $parse('book.title');

// fonction permettant l'écriture dans le context

var setter = getter.assign;

var context = {book: {title: '1984'}};

setter(context, 'Les fleurs du mal');
context.book.title === 'Les fleurs du mal'; // true
```

$interpolate

En charge de strings contenant une ou des expressions **délimitées** :

```
// php
$name = 'Foo';
$bar = "My name is $name";
```

```
# ruby
"three plus three is #{3+3}"
```

```
// ES6
`three plus three is #{3+3}`
```

$interpolate attributs

```
//                                  $parse             $parse
var getter = $interpolate('Hello {{name | uppercase}} {{41 + 1}}!');
expect(getter({name:'Angular'}).toEqual('Hello ANGULAR 42!');
```

- `mustHaveExpression`
- `trustedContext`
- `allOrNothing`

$compile

Filtres

Un filtre permet de transformer une expression dans la vue

- Syntaxe `|` héritée du pipeline Unix

```
<span>{{ expression | filtre }}</span>
```

- On peut enchaîner les filtres :

```
{{ expr | filtre1 | filtre2 }}
```

- On peut paramétrer les filtres avec le séparateur `:`

```
{{ expr | filtre:param1:param2 }}
```

Filtres hors de la vue

- On peut utiliser un filtre en dehors d'un template
- Peut être utile pour de la réutilisation de code simple
- Il suffit d'injecter la dépendance avec un suffixe "`Filter`"

```
// Supposons que le filtre "camelize" existe
app.controller("myCtrl", function (camelizeFilter) { // ← la dépendance "ca
  this.camelized = camelizeFilter("ce-texte_à_camelizer", param1, param2, …
});

// On peut aussi utiliser le service générique "$filter"
app.controller("myCtrl", function ($filter) {
  var camelize = $filter("camelize");
  this.camelized = camelize("ce-texte_à_camelizer", param1, param2, …);
})
```

Quelques filtres fournis par AngularJS

Filtre	Type d'entrée	Paramètres	Description
`uppercase`	string	-	Passe le texte en majuscules
`lowercase`	string	-	Passe le texte en minuscules
`currency`	number	symbol (string) decimals (number)	Formate un prix en fonction des paramètres de localisation
`date`	Date / number / string	format (string)	Formate une date ou un timestamp
`filter`	Array	filtre (expression) comparator (expression)	Filtre un tableau
`orderBy`	Array	predicate (expression) reverse (boolean)	Trie un tableau
`json`	expression	spacing (number)	Formate en JSON (utile pour le debugging)

Exemple d'utilisation avec ng-repeat

```
angular.module("app", [])
.controller("usersCtrl", function ($scope) {
  this.users = [
    { "name": "Bob",    "birth": 785796403199 },
    { "name": "John",   "birth": 889947622140 },
    { "name": "Jesus",  "birth": 397906446324 },
    { "name": "Arnold", "birth": 8640052377 }
  ].map(function (u) {
    u.age = Math.round((Date.now() - u.birth) / (365*24*3600*1000));
    return u;
  });
  this.majeur = function (u) {
    return u.age >= 18
  };
})
```

```
<body ng-app="app">
  Utilisateurs majeurs triés par nom :
  <ul ng-controller="usersCtrl as users">
    <li ng-repeat="user in users.users | filter:users.majeur | orderBy:'nam
      {{ user.name }} ({{ user.birth | date:'yyyy' }})
    </li>
  </ul>
</body>
```

Créer un filtre AngularJS

```
angular.module("app", [])
.filter("sum", function () { // ← Les éventuelles dépendances à injecter ic

  // Pattern "module" : les variables privées vont ici
  function plus (a, b) {
    return a + b
  }

  // On retourne la fonction de filtrage
  return function (array) {
    return array.reduce(plus)
  }
})

// Exemple de réutilisation
.filter("avg", function (sumFilter) {
  return function (array) {
    if (!array.length) {
      return null;
    }

    return sumFilter(array) / array.length
  }
})
```

```
Les utilisateurs ont en moyenne {{ users.ages | avg }}
```

Exercice : Implémenter le filtre pour pouvoir écrire

```
Les utilisateurs ont en moyenne {{ users.users | pluck:'age' | avg }}
```

Bâtir sur HTML5

- Récuperer le contrôle total avec `novalidate`
- Ne pas oublier de nommer le `<form>` et les `<input>` pour créer une instance de `FormController`
- Attention au confusion : `name` != `id`

```
<form name="userForm" novalidate>
  <p><label>Login : <input name="login"></label></p>
  <p><label>Mot de Passe : <input name="password" type="password"></label><
  <p><label>Rester connecté : <input type="checkbox"></label></p>
  <input type="submit" value="Connexion">
</form>
```

ng-model

En charge du 2 ways binding en écoutant la `value` de l'`<input>`

Ici la variable `$scope.user.name` sera mise à jour à chaque nouvelle lettre.

```
<p><label>Name : <input name="name" required type="text" ng-model="user.nam
```

Attributs booléens

Spécification HTML destructive :

- présent = `true`
- absent = `false`

Comportement problématique puisqu'on perd le binding lorque l'attribut est retiré.

Donc préferer `ng-required`, `ng-checked`, `ng-readonly`, `ng-selected`...

ng-options

- Sur les `<select>` évite les pièges du `ng-repeat` sur `<option>`
- Iterations sur *array* mais aussi sur *object* (key, value)
- Formatage possible en début de compréhensions :

```
label for value in array
select as label for value in array
label group by topic for value in array
select as label group by group for value in array track by expression
```

Soumission

L'attribut natif `action` trop direct est peu souvent désiré.

Plutôt choisir entre :

- `ng-submit` dans la balise `<form>` (se déclenche en appuyant sur *Entrée* si champs unique)
- `ng-click` sur un `<input>` du type `submit`

Validation

- Mix de validation HTML5 et interne AngularJS
- `type` de l'input utilisé (email, date, number, url, ...)
- Ajout d'un attribut sur l'input
- `ng-minlength`, `ng-maxlength`, `ng-pattern`
- custom

```
<form name="userForm" ng-submit="action(userForm)" novalidate>
  <p><label for="name">Name : </label><input id="name" name="name" required
  <p><label for="email">Email : </label><input id="email" name="email" type
  <input type="submit" value="Save">
</form>
```

État d'un formulaire

- `$pristine` (parfait, vierge)
- `$dirty` (sale, modifié)
- `$valid`
- `$invalid`
- `$pending` (la validation asynchrone d'un des champs est en cours)
- `$submitted` (pratique pour validation différée, en bloc)

```
<form name="userForm" novalidate>
  …
  <input type="submit" value="Save" ng-disabled="userForm.$invalid">
</form>
```

État d'un champ de formulaire

- idem : `$pristine`, `$dirty`, `$valid`, `$invalid`, `$pending`
- `$touched` (focus puis blur sur le champs)

```
<form ng-controller="myCtrl" name="userForm" ng-submit="action(userForm, us
  <p><label for="name">Name : </label><input id="name" name="name" required
  <p><label for="email">Email : </label><input id="email" name="email" type
  <input type="submit" value="Save">
</form>
```

```
var myCtrl = function($scope) {
  $scope.user = {
    name: "",
    email: ""
  }
  $scope.action = function(form, user) {
    console.log("--- email state ---");
    console.log("pristine : " + form.email.$pristine);
    console.log("dirty : " + form.email.$dirty);
    console.log("valid : " + form.email.$valid);
    console.log("invalid : " + form.email.$invalid);
  }
};
```

Affichage d'erreurs avec ng-if (ou ng-show)

```
<form name="userForm">
  <input type="text" name="city" ng-model="city" minlength="6" ng-pattern="
  <div ng-if="userForm.city.$touched">
    <div ng-if="userForm.city.$error.required">City is required</div>
    <div ng-if="userForm.city.$error.minlength">City length must be > 6</di
    <div ng-if="userForm.city.$error.pattern">City must start with B</div>
  </div>
  <input type="submit">
</form>
```

Avec ng-messages

Ne pas oublier de charger et déclarer la dépendance :

```
<script src="angular-messages.js"></script>
<script>
  angular.module("app", ["ngMessages"]);
</script>
```

Moins verbeux :

```
<form name="userForm">
  <input type="text" name="city" ng-model="city" minlength="6" ng-pattern="
  <div ng-messages="userForm.city.$error" ng-if="userForm.city.$touched">
    <div ng-message="required">...</div>
    <div ng-message="minlength">...</div>
    <div ng-message="pattern">...</div>
  </div>
  <input type="submit" />
</form>
```

ng-messages-multiple

Par défaut seul le premier message d'erreur s'affiche.

`ng-messages-multiple` permet de contourner cette restriction.

```
<form name="userForm">
  <input type="text" name="city" ng-model="city" minlength="6" ng-pattern="
  <div ng-messages="userForm.city.$error" ng-messages-multiple>
    <div ng-message="required">...</div>
    <div ng-message="minlength">...</div>
    <div ng-message="pattern">...</div>
  </div>
  <input type="submit" />
</form>
```

ng-messages-include

`ng-messages-include` se couple à un template et facilite la réutilisation de vos messages pour plusieurs champs.

```
<form name="userForm">
  <script type="text/ng-template" id="error-messages">
    <div ng-message="required">...</div>
    <div ng-message="minlength">...</div>
    <div ng-message="pattern">...</div>
  </script>

  <input type="text" name="country" ng-model="country" minlength="10" ng-pa
  <div ng-messages="userForm.country.$error" ng-messages-include="error-mes

  <input type="text" name="city" ng-model="city" minlength="6" ng-pattern="
  <div ng-messages="userForm.city.$error" ng-messages-include="error-messag
  <input type="submit" />
</form>
```

Cascade sur ng-messages-include

Si un champs nécessite un message d'erreur moins générique, on peut venir écraser localement les infos du template.

```
<form name="userForm">
  <script type="text/ng-template" id="error-messages">
    <div ng-message="required">Le champs est obligatoire</div>
    <div ng-message="minlength">Le champs n'est pas assez long</div>
    <div ng-message="pattern">Le champs ne respecte pas le pattern</div>
  </script>

  <input type="text" name="city" ng-model="city" minlength="6" ng-pattern="
  <div ng-messages="userForm.city.$error" ng-messages-include="error-messag
    <div ng-message="pattern">Le champs doit commencer par B</div>
  </div>
  <input type="submit" />
</form>
```

Classes CSS automatiques

- .ng-pristine
- .ng-dirty
- .ng-valid
- .ng-invalid

```
<style>
  input.ng-valid { background: lightgreen; }
  input.ng-invalid { background: pink; }
</style>
```

Performance

Maitriser les opérations couteuses

- Filtrer un `<table>` volumineux
- Effectuer une recherche / validation sur un serveur

ng-model-options

`debounce` permet de laisser un délai d'attente entre chaque saisie de caractères.

```
<input type="text"
  name="username"
  ng-model="username"
  minlength="10"
  pattern="^[-\w]+$"
  validate-username-availability
  ng-model-options="{ debounce : { "default" : 500 } }"
  required />
```

des exceptions peuvent être passées à debounce :

```
ng-model-options="{ debounce : { default : 500, blur : 0 } }"
```

updateOn

`updateOn` restreint les événements que l'on souhaite écouter :

```
ng-model-options="{ updateOn : 'blur' }"
```

Formulaires imbriqués

Il n'est pas possible d'imbriquer des `<form>` en HTML.

`ng-form` permet de simuler des `<fieldset>` logiques, mais pas d'envoi vers le serveur.

Principe

Les **services** ont pour rôle de recevoir toute la logique métier d'une app. Par exemple les échanges avec un serveur via `XMLHttpRequest`. Il existe plusieurs façon de créer un **service**.

Il est important de bien comprendre qu'un service est toujours un **singleton**. Ainsi, un service est partagé par l'ensemble de l'app.

Une bonne convention de nommage est d'utiliser le PascalCase et de suffixer par "Service". Par exemple "MyWonderfullService"

AngularJS propose beaucoup de services prédéfinis. Dans les slides suivants, nous allons en détailler quelques uns. Les services fournis par *AngularJS* commencent par **`$`**. Pour éviter la confusion, il est préférable d'éviter de créer un service commençant par `$`.

Pour utiliser un service il suffit de l'injecter là où on en a besoin.

```
angular.module("app", [])
.controller(function ($http, $timeout) {
  // les services natifs $http et $timeout sont disponibles
})
```

Communication avec le serveur

`$http` permet de communiquer avec un serveur. C'est un wrapper au dessus de `XMLHttpRequest`. Il s'agit d'une fonction qui prend un objet de config en entrée et qui retourne une **promesse**.

```
$http({
  method: "GET",
  url: "/api/users"
}).then(
  function (result) {
      // En cas de succès de la requête HTTP
  },
  function (error) {
      // En cas d'échec de la requête HTTP
  }
)
```

$http options

method (GET, POST, DELETE…), url, params (query), data, headers, cache, timeout, withCredentials (CORS), responseType ("json", "document").

La promesse est alimentée avec la réponse complète de l'appel `XMLHttpRequest`. Pour ne récupérer que les résultat, il faut donc utiliser `response.data`.

La promesse retournée est augmentée de 2 méthodes : **`success`** qui donne en 1er argument les données en cas de succès. **`error`** qui fait la même chose en cas d'échec. Attention, contrairement à `then` le chainage devient vite compliqué. Elles risquent d'ailleurs d'être dépréciées puis retirées en 1.5.

Méthode du service $http

Comme toute fonction dans JavaScript, `$http` est aussi un objet à qui on a ajouté des méthodes. Ces méthodes sont des helpers pour les verbes HTTP.

```
$http.get(url, config)
$http.head(url config)
$http.post(url, data, config)
$http.put(url, data, config)
$http.delete(url, config)
$http.jsonp(url, config)
```

Configuration globale du service $http

`$http` supporte beaucoup d'options de config dans le paramètre passé lors de l'appel. Pour les défininir de manière globale dans la config de l'app, on utilise le service `$httpProvider`

```
angular.module("app", [])
.config(function ($httpProvider) {

  $httpProvider.defaults.headers.post['Accept'] = 'application/json, text/j
  $httpProvider.defaults.headers.put['Content-Type'] = 'application/json; c
  $httpProvider.defaults.headers.post['Access-Control-Max-Age'] = '1728000'
  $httpProvider.defaults.headers.common['Content-Type'] = 'application/jso

  $httpProvider.defaults.cache = true;
})
```

Intercepter les appels HTTP

`$httpProvider` offre aussi la possibilité d'intercepter les appels (entrants ou sortants) HTTP. Par exemple on peut imaginer vouloir enregistrer une donnée chaque fois qu'un erreur HTTP apparaît Pour cela, il suffit d'enregistrer un service particulier dans le tableau `interceptors` du service `$httpProvider`

Interceptor HTTP

```
angular.module("app", [])
.config(function ($httpProvider) {
  $httpProvider.interceptors.push("myHttpInterceptor");
})
.factory("myHttpInterceptor", function ($q) {
  return {
    responseError: function (response) {
      console.error(response.data);
      return $q.reject(response);
    }
  };
})
```

Il est possible d'utiliser les 4 méthodes suivantes : `request`, `requestError`, `response`, `responseError`

Manipulation des promesses

AngularJS propose une librairie inspirée de **Q** de Kris Kowal pour créer et manipuler des **promesses**.

Pour créer une promesse on utilise un manager. On en obtient en utilisant la méthode `.defer()` du service `$q`

$q.defer

Un manager de promesses dispose d'une propriété (la promesse que l'on souhaite créer) et 3 méthodes :

- `.resolve(value)` : passe la promesse en statut **résolu**. Ne peut être appelée qu'une fois.
- `.reject(value)` : passe la promesse en statut **échec**. Ne peut être appelée qu'une fois.
- `.notify(value)` : déclenche le 3ème callback de la **promesse**. Peut être appelée plusieurs fois.

$q.defer

```
function getAsync() {
  var deferred = $q.defer();
  setTimeout(function () { // simulate async call
      deferred.resolve("ok");
  }, 500);
  return deferred.promise;
}

var promise = getAsync();

promise.then(function (value) {
  // success
}, function () {
  // failure
})
```

Méthodes du service $q

`$q` propose les 3 méthodes pour manipuler les promesses

$q.all

Cette méthode permet de transformer plusieurs promesses en une seule. La promesse créée est résolue avec un tableau des résolutions des promesses fournies. Ou passe en échec avec la raison de la 1ere promesse qui échoue

```
$q.all([
  $http.get("/api/users/" + id),
  $http.get("/api/users/" + id + "/friends")
]).then(function (responses) {
  var user = angular.extend({}, responses[0].data, {
    friends: responses[1].data
  });
})
```

On peut utiliser `_.spread` de lodash pour donner des noms plus explicites aux valeurs obtenues :

```
$q.all([
  $http.get("/api/users/" + id),
  $http.get("/api/users/" + id + "/friends")
]).then(_.spread(function (userRes, friendsRes) {
  var user = angular.extend({}, userRes.data, {
    friends: friendsRes.data
  });
}))
```

$q.when

Cette méthode transforme une valeur en une promesse auto-évaluée positivement. Utile lorsqu'on manipule un objet qui peut éventuellement être une promesse

```
var cache;
function getWithCache() {
  if (cache) return $q.when(cache);
  return $q.get("/api").then(function (response) {
    cache = response.data;
    return cache;
  });
}
```

$q.reject

Similaire à `.when()` mais évalue la promesse comme un échec

$q.when - code

```
var when = function(value, callback, errback, progressBack) {
  var result = new Deferred();
  result.resolve(value);
  return result.promise.then(callback, errback, progressBack);
};
```

Service $timeout

`$timeout` est un wrapper au dessus de la méthode `window.setTimeout`. Il retourne une promesse et la fonction de callback est optionnelle

```
$timeout(10000)
  .then(function () {
    console.log("ok"); // méthode appelée au bout de 10 secondes
  });
```

La méthode `.cancel()` annule une promesse retournée par `$timeout`. La promesse est alors évaluée comme un échec

```
var promise = $timeout(10000)
  .then(function () {
      console.log("ok"); // méthode appelée au bout de 10 secondes
  }, function () {
      console.log("annulé");
  });
$timeout.cancel(promise);
```

Service $location

`$location` est d'une part un wrapper de la fonction `window.location` et d'autre part permet de charger une route ou connaitre la route courante

La méthode la plus utile sur ce service est la méthode `.path()` qui agit comme un getter / setter

```
console.log($location.path()); // affiche la route courante
$location.path("/about"); // positionne le navigateur sur la route /about
```

Les autres méthodes du service sont : `absUrl()`, `url([url])`, `protocol()`, `host()`, `port()`, `search([search])`, `hash()`, `replace()`, `state()`

Service d'accès aux filtres

2 façons pour acceder aux filtres dans un contrôleur ou un service

D'une part chaque filtre expose sa fonction de filtre comme un service qui se nomme [nomDuFilltre]Filter par exemple le filtre date expose un service qui s'appelle **dateFilter**

```
angular.module("app", [])
.controller("MyController", function (dateFilter) {
  var formattedDate = dateFilter(new Date(), "mediumDate");
})
```

D'autre part il existe un service `$filter` :

```
angular.module("app", [])
.controller("MyController", function ($filter) {
  var filter = $filter("date");
  var formatedDate = filter(new Date(), "mediumDate");
})
```

Créer un service : factory

Un service de type factory expose un Objet.

La méthode `.factory()` attend 2 arguments :

- Le nom du service (par convention en PascalCase et finissant par "Service")
- Une function injectable qui doit retourner un Objet

```
angular.module("app", [])
.factory("MyService", function ($http) {
  return {
    version: "1.0",

    getUsers: function () {
      return $http.get("/api/users")
        .then(function (response) {
          return response.data;
        });
    }
  };
})
```

Factory

Il est possible d'utiliser une construction plus complexe avec une partie privée, non retournée.

```
angular.module("app", [])
.factory("MyService", function ($http) {
  return {

    version: "1.0",

    getUsers: function () {
      return $http.get("/api/users")
        .then(function (response) {
          return enhance(response.data);
        });
     }
  };

  function enhance (users) {
    return users.map(function (user) {
      return angular.extend({}, user, {
        fullname: user.firstname + " " + user.lastname;
      });
    });
  }
})
```

Créer un service : provider

Il est impossible d'utiliser un service `factory` lors de l'étape de *config* d'une app. À cette étape, il est possible de paramétrer la façon dont certains services vont fonctionner.

Pour créer un tel service configurable on utilise un **provider**. Par exemple `$httpProvider` permet de customiser le fonctionnement du service `$http`

Un service de type **provider** fourni en fait 2 services. Le service lui-même et un service de config qui sera nommé "[NomDuService]Provider" qui sera utilisable lors de l'étape de config.

Provider

La syntaxe pour créer un service de type **`provider`** est :

- Le nom du service (par convention en PascalCase et finissant par "Service")
- Une function injectable qui doit retourner un Objet contenant une méthod `.$get()`

`$get` est en fait le service final (avec ses dépendances comprises).

```
angular.module("app", [])
.provider("MyService", function () {
  var api = "";
  return {

    setApi: function (_api) { // cette méthode sera uniquement disponible c
      api = _api;
    },

    $get: function ($http) { // le service réel
      return {
        version: "1.0",
        getUsers: function () {
          return $http.get(api + "/api/users")
            .then(function (response) {
              return response.data;
            });
        }
      };
    }
  };
})
.config(function (MyServiceProvider) {
  MyServiceProvider.setApi("http://myrestapi.fr");
})
```

Créer un service : value

Un service de type **`value`** est très simple. Il se contente d'exposer une valeur.

```
angular.module("app", [])
.value("configuration", {
  version: "1.0",
  name: "Ma belle application",
  lang: "fr"
})
```

Note : Il est possible d'exposer un objet, ou une valeur quelconque.

Créer un service : constant

Un service de type **`constant`** est équivalent à un service de type **`value`** à la différence près qu'un service **`constant`** est utilisable durant la phase de config.

```
angular.module("app", [])
.constant("api_key", "Vght165u888")
.config(function (api_key) {
  // Faire bon usage de l'api_key
})
```

Note : Il est possible d'exposer un objet, ou une valeur quelconque.

Service vs Factory vs Provider

Déclarations

```
app.factory('f', fn);
app.service('s', fn);
app.provider('p', fn);
```

Cache de l'injecteur

```
cache.f = fn()
cache.s = new fn()
cache.p = (new fn()).$get()
```

Synthese Services

	Phase config	**Phase run**
Simple	const	value
Complexe	provider	factory

Scope

- C'est un objet permettant le **lien** entre le modèle et la vue.
- C'est un contexte d'éxecution pour les **expressions**.
- Il prend place dans une arborescence de **scopes** qui s'appuie sur celle du du DOM.
- Il peut **surveiller** une expression.
- Il peut propager et attraper des **événements**.

Héritage des scopes

Chaque **scope** est inscrit dans une hiérarchie de **scopes**, avec un *scope* père et des *scopes* fils.

Au sommet de cette arborescence se trouve le `$rootScope`

L'héritage entre *scopes* est similaire à l'héritage prototypal des objets en JavaScript.

```
var parent = $rootScope;
var child = parent.$new();

// Surcharge des types scalaires
parent.aString = "Bonjour";
console.log(child.aString); // Bonjour
child.aString = "Au revoir";
console.log(child.aString); // Au revoir
console.log(parent.aString); // Bonjour

// Surcharge des types objets
parent.anObject = { name: "Thomas" };
console.log(child.anObject.name); // Thomas
child.anObject.name = "Nicolas";
console.log(child.anObject.name); // Nicolas
console.log(parent.anObject.name); // Nicolas

// Mais
parent.anObject = { name: "Thomas" };
child.anObject = { name: "Nicolas" };
console.log(child.anObject.name); // Nicolas
console.log(parent.anObject.name); // Thomas
```

La méthode `new()` n'est pas utilisée dans un projet standard, elle est utilisée en interne par *AngularJS* pour créer l'arborescence des *scopes*. Attention à l'héritage des *scopes* lors de l'utilisation dans les contrôleurs ou les vues. L'usage de la syntaxe `controller as` évite ce problème.

Propagation des événements

AngularJS utilise la hiérarchie créée par les **scopes** pour propager des événements.

- `.$emit()` crée un événement qui est envoyé à tous les ancêtres du scope courant et ce jusqu'au `$rootScope`
- `.$broadcast()` crée un événement qui est envoyé à tous les fils.

```
var message = "Un message à transmettre";
var data = { value: "des données à transmettre" };
$scope.$emit("eventName", message);
$scope.$broadcast("otherEvent", message, data);
```

Il est possible de passer plusieurs arguments de tout type après le nom de l'évenement.

Réagir à un événement

- `.$on()` écoute un évenement est déclenche un callback appelé avec 1+n paramètres. Le 1er est un objet événement donnant des informations sur l'événement (target, name, stopPropagation()...). Les arguments suivants sont ceux donnés lors de la création de l'événement.

```
$scope.$on("eventName", function (event, message) {
    console.log(message); // Un message à transmettre
});
```

Événements AngularJS

AngularJS déclenche plusieurs événements selon le contexte : `$routeChangeStart`, `$routeChangeError`, `$routeChangeSuccess`, `$routeUpdate`, `$destroy`, `$locationChangeStart`, `$locationChangeSuccess`

Observer les changements sur un scope

`$scope` offre la possibilité d'écouter et de réagir à chaque changement d'expression.

`$scope` offre 3 méthodes avec le même fonctionnement. Passer une expression ou série d'expressions et un callback.

Ce dernier reçoit en arguments la nouvelle valeur et l'ancienne.

$watch

Observe une expression. Une expression est considérée comme modifiée si l'ancienne valeur est `!==` de la nouvelle valeur.

```
$scope.name = "Thomas";
$scope.user = { name: "Thomas" };

$scope.$watch("name", function (newValue, oldValue) {
  console.log("La chaîne de caractères 'name' a changé");
});
$scope.$watch("user", function (newValue, oldValue) {
  console.log("L'objet 'user' a changé");
});
$scope.$watch("user", function (newValue, oldValue) {
  console.log("Une propriété de 'user' a changée");
}, true);
$scope.$watch("user.name === 'Bruno'", function (newValue, oldValue) {
  if (newValue) console.log("Le user a comme nom 'Bruno'");
  else console.log("Le user n'a plus comme nom 'Bruno'");
});

$timeout(1000).then(function () {
  $scope.name = "Nicolas";
  $scope.user.name = "Nicolas";
});
$timeout(2000).then(function () {
  $scope.user = { name: "Bruno" };
});
```

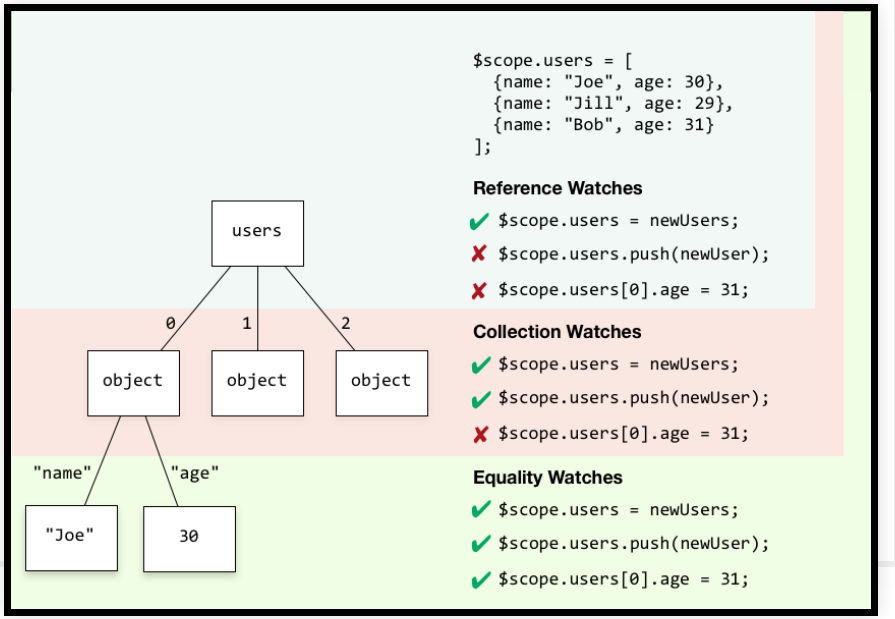
$watchGroup

Équivalent à `.$watch()` mais pour un tableau d'expressions

$watchCollection

Observe une collection (un objet) et réagit aux changements des propriétés de cet objet. C'est le même fonctionnement (mais préférable) que `$watch` avec le 3ème paramètre à `true`.

$watch - synthese



$scope.$apply

`$eval()` puis `$rootScope.$digest()`

```
$apply: function(expr) {
  try {
    beginPhase('$apply');
    return this.$eval(expr);
  } catch (e) {
    $exceptionHandler(e);
  } finally {
    clearPhase();
    try {
      $rootScope.$digest();
    } catch (e) {
      $exceptionHandler(e);
      throw e;
    }
  }
}
```

$scope.$eval

Simple délégation à `$parse`

```
$eval: function(expr, locals) {
  return $parse(expr)(this, locals);
},
```

Principe du routage

Scénario similaire à une app web client / serveur classique.

Trois étapes :

1. Requête utilisateur (URL)
2. Le Routeur décide de la route
3. Passe le relais à un contrôleur et une vue (template)

Templating

- Single Page App, donc un seul fichier HTML
- Chargement de templates en XHR par *AngularJS* pour assembler des layouts simples.

ng-include

```
<div ng-include="'template1.html'"></div>
<hr />
<div ng-include="'template2.html'"></div>
```

ng-view

```
<div ng-view><!-- Content provided by the router --></div>
```

Configuration des routes

Le module n'est plus fourni par défaut, il ne faut pas oublier de le charger. La déclaration des routes se fait dans un bloc `.config()`

```
angular.module("app", ["ngRoute"])
.config(function($routeProvider) {
  $routeProvider
    .when("/", {
      templateUrl: "views/home.html",
      controller: "HomeCtrl"
    })
    .when("/page", {
      templateUrl: "views/page.html",
      controller: "PageCtrl"
    });
});
```

Otherwise

Route par défaut quand aucune autre n'a été retenue. L'ordre est important pour la gestion des jockers.

```
angular.module("app", ["ngRoute"])
.config(function($routeProvider) {
  $routeProvider
    .when("/", {
      templateUrl: "views/home.html",
      controller: "HomeCtrl"
    })
    …

    .otherwise({
      redirectTo: "/"
    });
  });
```

Déclaration d'un template

- `templateUrl` : via une URL (chargement d'un template externe).
- `template` : Directement du code HTML.

```
angular.module("app", ["ngRoute"])
.config(function($routeProvider) {
  $routeProvider
    .when("/", {
      templateUrl: "views/home.html",
      controller: "HomeCtrl"
    })
    .when("/page", {
      template: "<div>{{ variable }}</div>",
      controller: "PageCtrl"
    });
});
```

Déclaration d'un contrôleur

- Option 1 : le nom d'un contrôleur défini par ailleurs
- Option 2 : une fonction servant de contrôleur

```
angular.module("app", ["ngRoute"])
.config(function($routeProvider) {
  $routeProvider
    .when("/", {
      templateUrl: "views/home.html",
      controller: "HomeCtrl"
    })
    .when("/page", {
      templateUrl: "views/page.html",
      controller: function($scope) { /* … */ }
    });
});
```

Syntaxe ControllerAs

Pour obtenir l'équivalent de `ng-controller="UserListCtrl as userList"`

```
angular.module("app", ["ngRoute"])
.config(function($routeProvider) {
  $routeProvider
    .when("/users", {
      templateUrl: "views/userList.html",
      controller: "UserListCtrl",
      controllerAs: "userList"
    })
});
```

Passer des paramètres via l'URL

```
angular.module("app", ["ngRoute"])
.config(function ($routeProvider) {
  $routeProvider
    // monsite.net/#/users/134/friends/lilian
    .when("/users/:id/friends/:friend", {
      template: "views/user.html",
      controller: "UserCtrl"
    });
    // monsite.net/#/users/134
    .when("/users/:id", {
      template: "views/user.html",
      controller: "UserCtrl"
    });
});
```

Options de déclaration (jokers)

- /users/:name? (optionnel)
- /users/:name*/\edit (gourmand)

Paramètres dans le contrôleur

```
.controller("UserCtrl",
  function ($scope, $routeParams) {
    // URL   http://monsite.net/#/users/134/friends/lilian?search=meetings
    // ROUTE /users/:id/friends/:friend"
    // $routeParams {id: 134, friend: 'lilian', search: 'meetings'}

    $scope.id = $routeParams.id;
  })
```

$location

- interface avec `window.location`
- `$location.path()`, `$location.path("/page")`
- `$location.replace()`
- `$location.hash()`, `$location.hash("fragment")`
- absUrl, host, port, protocol, search, url
- jamais de rechargement complet de la page
- `$window.location.href = "/reload/page";`

Question: quel est l'interêt d'utiliser `$window` au lieu de `window` ?

Evénements

- `$routeChangeStart`
- `$routeChangeSuccess`
- `$routeChangeError`
- `$routeUpdate`

```
$rootScope.$on("routeChangeStart", function (event, next, current) {
  // …
});
```

- `$viewContentLoaded`

Instanciation du contrôleur

Pour effectuer des initialisations ou des vérifs lors de l'utilisation d'une route avant d'instancier le contrôleur.

`resolve` doit être un objet simple dont chaque proriété peut être soit le nom d'un service, soit une fonction.

Les items résolus sont injectables dans le contôleur

```
angular.module("app", [])
.config(function ($routeProvider) {
  $routeProvider
    .when("/", {
      templateUrl: "views/home.html",
      controller: "HomeCtrl",
      resolve: {
        key1: "service",
        key2: function () {
          return "bonjour";
        }
      }
    })
})
.controller("HomeCtrl", function ($scope, key1, key2) {
  // key1 représente le service
  // key2 aura la valeur "bonjour"
});
```

Resolve et promesse

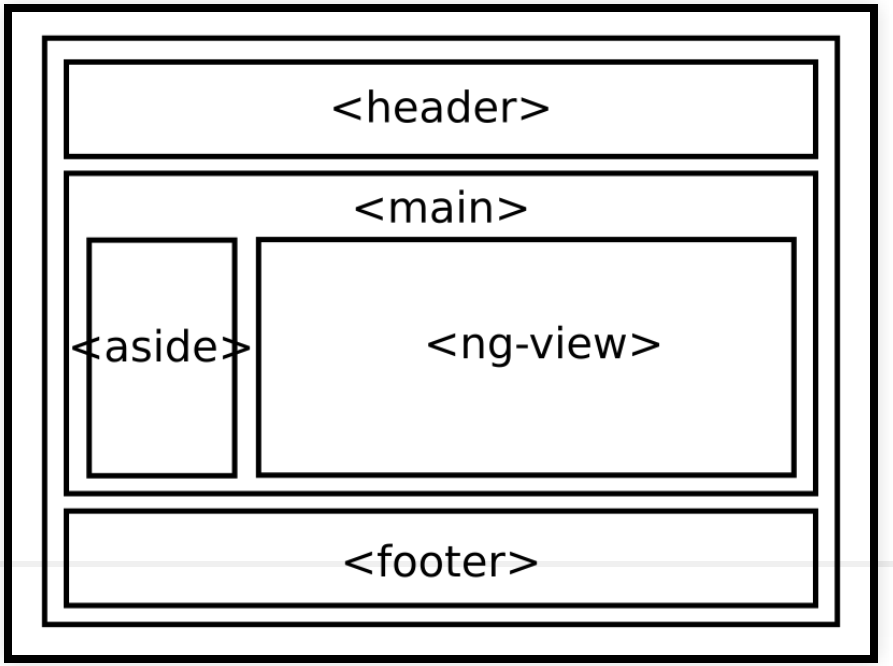
Si la propriété d'un objet `resolve` est une promesse, *AngularJS* attend sa résolution avant d'instancier le contrôleur. La valeur passée est alors le résultat de la promesse.

En cas de rejet de la promesse, le contrôleur n'est pas instancié : un évenement `$routeChangeError` est lancé.

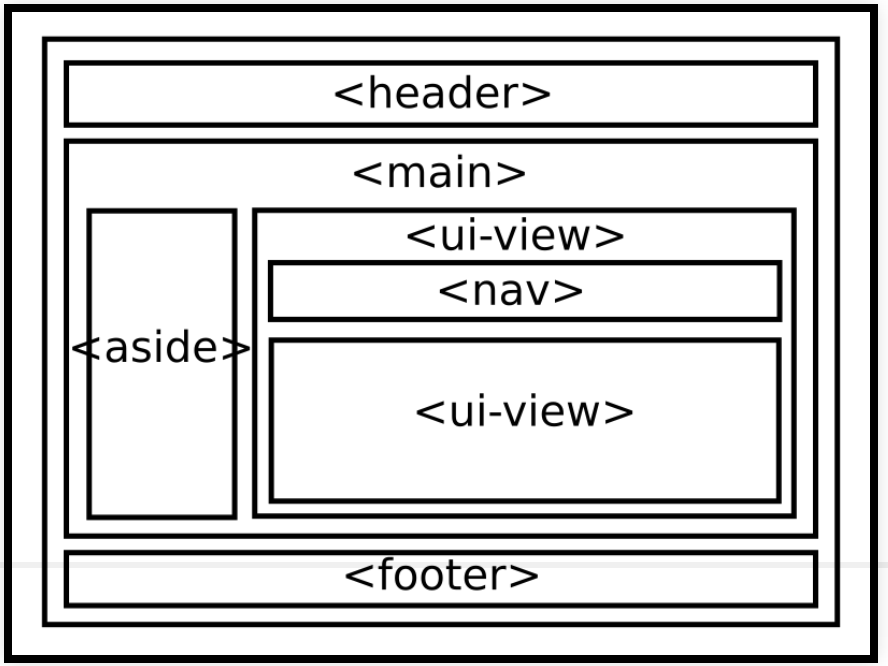
UI Router

- Projet membre d'angular-ui
- Facilite le découpage en vues **imbriquées** ou **parallèles**.
- Gère une machine à **états**, place les URLs au second plan
- Inpiré par *Ember* et constitue lui même la base de travail pour *AngularJS* 2

Layout simple



Layout complexe



Vues parallèles

`ng-view` laisse la place à `ui-view`

Cette directive peut apparaître à de multiples endroits.

Ainsi, un nom est nécessaire pour distinguer 2 occurences au sein d'un même élement :

```
<body>
  <div ui-view="firstView"></div>
  <div ui-view="secondView"></div>
</body>
```

Vues imbriquées

Ce nommage est facultatif quand il n'y a pas d'ambiguité.

```
<!-- index.html -->
<body>
  <div ui-view></div>
</body>
```

```
<!-- partial.html -->
<div>
  <h2>Partielle</h2>
  <div ui-view></div>
</div>
```

Routes vs Etats

```
$routeProvider.when('/users/:id', {
  templateUrl: 'users.html',
  controller: function($scope){ ... },
})
```

Désormais on décrit des **états nommés**, sur lesquels on *peut* attacher des URLs

```
$stateProvider.state('user.detail', {
  url: '/users/:id',
  templateUrl: 'users.html',
  controller: function($scope){ ... },
})
```

Hierarchie des états

L'utilisation du **point** dénote le lien entre états imbriqués.

```
$stateProvider
  .state('home', { … })
  .state('users', { … })
  .state('users.profile', { … })
  .state('users.profile.edit', { … })
```

`users.profile` est un état fils de `users`

une autre façon de décrire ce lien de filiation entre états est d'utiliser la propriété `parent`

```
$stateProvider
  .state('users', { … })
  .state('profile', {
    parent: 'users'
  })
```

Activation des états

- `$state.go()`
- Cliquer sur un lien avec `ui-sref`
- Naviguer vers l'**URL** de l'état s'il existe

$state.go()

Se réalise dans les controllers :

```
.controller('userCtrl', ['$scope', '$state', function($scope, $state) {
     $scope.goToProfile = function() {
        // le nom, puis les params du state si besoin
        $state.go('users.profile', {id: $scope.selectedId});
     }
  })
```

$state.go() - raccourcis

- vers le parent : `$state.go('^')`
- vers un fils: `$state.go('.profile')`
- relatif: `$state.go('^.profile')`
- absolu: `$state.go('users.profile')`

ui-sref

Cette directive s'utilise à la place de `href`

```
<a ui-sref="users">Utilisateurs</a>
```

Elle génère `href` si une URL est associée à l'état

```
<a ui-sref="users" href="#/users">Utilisateurs</a>
```

ui-sref arguments

Pour passer des paramêtres:

```
<li ng-repeat="user in users">
  <a ui-sref="users.profile({id: user.id})">{{user.name}}</a>
</li>
```

Les raccourcis fonctionnent aussi:

```
<a ui-sref="^">Utilisateurs</a>
```

via URLs

Les URLs des états fils, se concatènent à celle de l'état parent:

```
$stateProvider
 .state('about', {
    url: '/about',
 })
 .state('about.company', {
    url: '/company',
 });
```

L'URL de about.company est `/about/company`

Pour construire des URLs absolus :

```
$stateProvider
 .state('about', {
    url: '/about',
 })
 .state('about.company', {
    url: '^/company',
 });
```

`/company`

$stateParams

URL de l'état:

http://monsite/#users/42/profile

/users/:id/profile

$stateParams: `{id: 42}`

Vue nommées

```
$stateProvider
  .state('users', {
    url: "users"
    views: {
      // clés: nom de la vue
      // valeurs: config pour chaque vue
      'main': { ... },
      'sidenav': { ... }
    }
  })
```

Noms des vues

Relatif au parent

- `'main'` - vue *main* dans le template parent
- `''` - vue anonyme dans le template parent

Absolu (@)

- `'profile@users'`- vue *profile* dans le template de l'état *users*
- `'profile@'`- vue *profile* dans index.html
- `'@users'`- vue anonyme dans le template de l'état *users*

Config des vues

```
$stateProvider
  .state('users', {
    url: 'users'
    views: {
      'main': { ... },
      'sidenav': {
         templateUrl: 'sidenav.html',
         controller: 'SideNavCtrl'
      }
    }
  })
```

Faire ses propres directives

Faire ses propres directives est un des sujets les plus ardus de *AngularJS*. C'est pourtant une façon de mettre un pied dans le futur monde des **WebComponents**.

Avec les directives, il est possible de créer de nouveaux éléments ou de modifier le comportement d'éléments existants avec de nouveaux attributs.

> "*AngularJS is what HTML would have been, had it been designed for building webapps.*"

Syntaxe d'une directive

Pour créer une directive et l'attacher à un module, on utilise la méthode `.directive()` du module. Cette méthode prend 2 arguments :

- Le nom de la directive en PascalCase (qui sera ensuite utilisée en kebab-case dans le HTML).
- Une fonction retournant un objet contenant le paramétrage de la directive.

```
angular.module("app", [])
.directive("maDirective", function () {

    return {

        // Options de la directive
    }

})
```

Exemple

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angu
    <script>
angular.module("app", [])
.directive("randomGame", function () {
    return {
        restrict: "E",
        templateUrl: "random-game.html",
        scope: {
            max: "@",
            win: "&",
            loose: "&"
        },
        link: function (scope, elem, attrs) {
            var color = attrs.borderColor || "red";
            elem.find("div").css({ "border": "3px solid " + color });
        },
        controller: "RandomGameDirectiveController",
        controllerAs: "ctrl",
        bindToController: true,
        transclude: true
    }
})
.controller("RandomGameDirectiveController", function () {
    var ctrl = this;
    ctrl.max = Number(ctrl.max) || 10;
    ctrl.buttons = [];
    var attempts = ctrl.max;

    var randomValue = Math.floor(Math.random() * ctrl.max + 1);

    for (var i = 0; i < ctrl.max; i++) {
        ctrl.buttons.push({ value: i+1 });
    }
```

Restrict

L'option `restrict` permet de définir la façon dont sera utilisée la directive :

- **"E"** : comme un nouvel élément.
- **"A"** : comme un nouvel attribut d'un élément existant.
- **"C"** : comme une class CSS.

Il est aussi possible de mixer ces valeurs :

- "EA" ou "AE" : comme élément ou comme attribut.
- "EC" ou "CE" : comme élément ou comme classe CSS.
- "AC" ou "CA" : comme attribut ou comme classe CSS.
- "EAC" ou ... : ...

Restrict

- "E": `<ma-directive></ma-directive>`
- "A": `<div ma-directive></div>`
- "C": `<div class="ma-directive"></div>`

Template

Si la directive doit afficher du HTML, celui-ci peut être définie dans une template. 2 options pour ce faire :

- `template` : la chaîne de caractères donnée sera utilisée comme template.
- `templateUrl` : nom du template à récupérer (via cache ou script ou XMLHttpRequest)

Ces options sont évidemment mutuellement exclusives.

Link

S'il est fortement déconseillé de manipuler le DOM dans une app *AngularJS*, il y a toutefois une exception. La méthode `link` d'une directive est l'endroit dans lequel il est fait le lien entre le monde *AngularJS* (en particulier le scope) et le DOM.

Cette méthode prend 3 arguments :

- **`scope`** : le scope courant de la directive.
- **`element`** : l'élément portant la directive (élément JQlite ou JQuery).
- **`attributs`** : la liste (objet) des attributs de la directive.

Link

Cette méthode est l'endroit idéal pour traiter les événements du DOM et les transformer en interaction *AngularJS* ou au contraire, réagir aux changements dans le scope et modifier le DOM en conséquence.

Il peut par exemple être fait usage de la méthode `.$watch()` du scope.

```
return {
  link: function (scope, element, attrs) {
    scope.$watch("user.name", function (nv, ov) {
      if (!ov && nv) {
        element.addClass("connected");
      }
    })
  }
}
```

Scope

Par défaut, une directive utilise le **scope courant**, le scope dans lequel elle est utilisée. C'est pratique, mais aussi très dangereux. Une directive est faite pour être utilisée dans plusieurs contexte, il est donc préférable de ne pas utiliser le scope par défaut. Au moins pour les directives que l'on veut partager.

Les valeurs possibles pour cette option sont :

- `false` (valeur par défaut) : aucune création de scope, le scope courant est utilisé.
- `true` : un scope fils est créé.
- `{}` : un scope isolé (sans lien avec le scope courant) est créé.

Scope isolé

La troisième option est la plus propre et c'est aussi la plus complexe à utiliser. En effet une fois que l'on a isolé une directive, il s'agit de définir la façon dont cette directive intéragit avec son contexte d'appel.

Pour ce faire, plutôt que donner un objet vide, il est possible de passer un objet contenant des liens entre la directive et son contexte. Ces liens peuvent être de 3 types :

- `"="` : Lien bi-directionnel.
- `"@"` : Copie de valeur.
- `"&"` : Expression du contexte appelant à évaluer.

Scope isolé : liaison bi-directionnel

En utilisant la valeur `=`, on définit une liaison entre l'objet du scope courant et l'objet créé dans le scope de la directive.

```
{
  scope: {
    name: "="
  }
}
```

```
<ma-directive name="user.name"></ma-directive>
```

Dans cet exemple, toute modification faite sur la propriété **user.name** du scope courant sera aussi faite sur la propriété **name** du scope de la directive. Et inversement.

Scope isolé : copie de valeur

En utilisant la valeur `@`, la valeur (chaine de caractères) passée dans l'attribut est recopiée dans le scope de la directive.

```
{
  scope: {
    color: "@"
  },
  template: "<span>{{ color }}</span>"
}
```

```
<ma-directive color="red"></ma-directive>
```

Scope isolé : évaluation d'une expression

Le mot clé `&` crée sur le scope de la directive une méthode qui lors de son appel évalue l'expression passée dans le contexte du scope courant.

```
{
  scope: {
    onAction: "&"
  },
  link: function (scope, element, attrs) {
    $timeout(function () {
      scope.onAction();
    }, 1000)
  }
}
```

```
<ma-directive on-action="doSomething()"></ma-directive>
<ma-directive on-action="display = false"></ma-directive>
```

Scope isolé : évaluation d'une expression

Ici, `doSomething()` est une méthode du scope courant. La directive n'a pas de méthode `doSomething()` mais l'appel à la méthode `onAction()` va évaluer l'expression sur le contexte courant et donc appeler la méthode `doSomething()`.

De la même manière, le scope de la directive ne possède pas de variable `display`. La variable `display` du scope courant sera positionnée à true lorsque l'expression sera évaluée, c'està dire lors de l'appel de la méthode `onAction` par la directive.

Définition d'un contrôleur

Une directive peut être considérée comme une véritable mini app. Ainsi il est possible (et même conseillé) de lui adjoindre un contrôleur. Celui-ci sera utilisé pour mettre en place toute la logique nécessaire.

L'ajout d'un contrôleur se fait par l'option `controller` qui peut revêtir 2 formes :

- Le nom d'un contrôleur déclaré ailleurs sur le module (solution préférée)
- Le code direct du contrôleur

Définition d'un contrôleur

```
angular.module("app", [])
.directive("maDirective", function () {
  return {
      // …
    controller: "SpecialController"
  }
})
.controller("SpecialController", function () {
    // Logique de la directive
})
```

Le lien entre la méthode `link` et le contrôleur se fait par l'intermédiaire du scope.

Pour les directives les plus importantes, il est même envisageable d'injecter un service spécialement écrit pour l'occasion qui sera le lieu de toute l'intelligence.

Une avantage important d'utiliser un contrôleur et un service dans une directive est de pouvoir aisément tester ces derniers.

Utilisation de la syntaxe "controller as"

Comme pour `ng-controller` ou pour les routes, il est possible d'utiliser la syntaxe "controller as" dans une directive. Ceci permet de donner un nom à l'instance du contrôleur.

Pour cela la clé **`controllerAs`** est utilisée. Elle prend tout simplement le nom de l'instance du contrôleur.

```
angular.module("app", [])
.directive("maDirective", function () {
    return {
        controller: function () {
            this.name = "Thomas";
        },
        controllerAs: "ctrl",
        template: "<strong>{{ ctrl.name }}</strong>"
    }
})
```

L'utilisation de la syntaxe "controller as" est obligatoire pour l'utilisation de `require`

Lier les entrées isolées au contrôleur

Avec la syntaxe **controller as**, il n'est plus nécessaire d'utiliser directement le scope. Par contre les données récupérées dans l'option `scope` sont toujours positionnée sur le scope de la directive. Pour qu'elles soient positionnées sur l'instance du contrôleur et non plus sur le scope, il faut utiliser l'option `bindToController` avec un valeur **`true`**

```
angular.module("app", [])
.directive("maDirective", function () {
  return {
    controller: function () {},
    controllerAs: "ctrl",
    binToController: true,
    scope: {
      color: "@"
    },
    template: "<strong>{{ ctrl.color }}</strong>"
  }
})
```

Récupération du contenu de l'élément

Inclusion par référence d'un document ou d'une partie d'un document dans un autre document.

Par défaut, le contenu d'une directive est ignoré et remplacé. Il est possible d'utiliser ce contenu :

- Option `transclude` à mettre à `true`
- Utiliser la directive `ng-transclude` dans le template

Transclusion

```
angular.module("app", [])
.directive("maDirective", function () {
  return {
    transclude: true,
    template: "<div>[…]<p ng-transclude></p></div>"
  }
})
```

```
    <ma-directive>Texte à "transcluder"</ma-directive>
```

Note : Le bloc à transcluder appartient toujours au scope appelant. C'est à dire que si vous utilisez une expression dans le HTML à transcluder, elle sera évaluée en fonction du scope appelant et non pas du scope de la directive.

Remplacer la directive dans le DOM

Par défaut, la directive reste visible dans le DOM une fois celle-ci interprétée. C'est à dire, par exemple, que la balise `<ma-directive></ma-directive>` reste visible dans le code source de la page. Pour éviter cela, mettre `replace` à `true`. (déprecié en 1.4)

```
angular.module("app", [])
.directive("maDirective", function () {
  return {
    replace: true,
    template: "<div>[…]<p ng-transclude></p></div>"
  }
})
```

Attention, en utilisant `replace`, l'élément passé dans la méthode `link` devient l'élément racine du template et non plus l'élément directive.

Interdépendance entre directives

Il est possible d'écrire une directive qui soit dépendante d'une autre directive. Par exemple pour modifier ou compléter son comportement. Pour cela *AngularJS* propose le mécanisme de `require`

Pour rendre une directive dépendante d'une autre il suffit de passer le nom de la directive requise à la directive dépendante. Il devient alors possible d'utiliser un 4ème paramètre à la méthode `link`. Ce paramètre est l'instance du contrôleur de la directive requise.

Require

```
angular.module("app", [])
.directive("master", function () {
  return {
    template: "<div>{{ ctrl.message }}</div>",
    controllerAs: "ctrl",
    controller: function () {
      this.message = "Houba Houba";
      this.appendMessage = function (msg) {
        this.message += " " + msg;
      }
    }
  }
})
.directive("optionalMessage", function () {
  return {
    require: "master",
    scope: true,
    link: function (scope, element, attrs, masterCtrl) {
      masterCtrl.appendMessage("Hop");
    }
  }
})
```

```
<master></master> <!--  Houba Houba -->
<master optional-message></master><!-- Houba Houba Hop -->
```

Il est possible de déclarer plusieurs directives en utilisant un tableau de chaînes de caractères. Dans ce cas le 4ème argument du link sera un tableau d'instances de contrôleurs.

Par défaut, la requête est attendue au même niveau du DOM, mais il est possible de chercher dans les parents en préfixant le nom de la directive par `^`. Par défaut, la directive requise est obligatoire. Il est possible de la rendre optionelle en suffixant le nom de la directive par `?`.

Définir son propre validateur

Un validateur n'est rien de plus qu'une directive requiérant la présence de la directive `ng-model` et modifiant le comportement de cette dernière.

```
angular.module("app", [])
.directive("twoWords", function() {
  return {
    require: "ngModel",
    link: function(scope, element, attrs, ngModelCtrl) {
      ngModelCtrl.$validators.weird = function(value) {
        value.split(" ").length === 2;
      };
    }
  }
});
```

```
<input type="text" ng-model="name" two-words>
```

Il existe d'autres options pour créer des validateurs. **$asyncValidators** par exemple qui permet d'ajouter un validateur asynchrone.

Principe : Classes CSS

AngularJS ajoute automatiquement des classes CSS aux champs de formulaires valides ou invalides.

La gestion des animations se base sur le même principe : le framework ajoute ou retire des classes CSS sur les éléments marqués par **certaines directives**.

enter & leave

Ces directives introduisent ou font disparaitre des éléments.

`ng-if`, `ng-switch`, `ng-include`, `ng-message`, `ng-view`

move

`ng-repeat` lors du déplacement d'un item.

add & remove

- `ng-class` (presence)
- `ng-show` / `ng-hide` (class value)
- `form` / `ng-model` (dirty, pristine, invalid etc.)
- `ng-messages` (ng-active / ng-inactive)

3 Approches

- **Transition CSS** (état initial -> état souhaité)
- **Animation CSS** (keyframes)
- **Animation JS** (via lib tiers comme GreenSock)

Transition CSS

```
<div ng-if="bool" class="fade">
   Fade me in out
</div>
<button ng-click="bool=true">Fade In!</button>
<button ng-click="bool=false">Fade Out!</button>
```

```
.fade.ng-enter {
  transition: 0.5s linear all;
  opacity: 0;
}
.fade.ng-enter.ng-enter-active {
  opacity: 1;
}
.fade.ng-leave {
  transition: 0.5s linear all;
  opacity: 1;
}
.fade.ng-leave.ng-leave-active {
  opacity: 0;
}
```

Animation CSS

Ici pas besoin de se soucier des `-active`.

```
.fade.ng-leave {
  animation: fade_animation 0.5s linear;
}

@keyframes fade_animation {
  from { opacity: 1; }
  to { opacity: 0; }
}
```

Animation JS

Ici avec jQuery, mais d'autres libs (GreenSock, velocity) peuvent être utilisées.

```
angular.module('app', [])
.animation('.slide', [function() {
  return {
    enter: function(element, doneFn) {
      // Ne pas oublier d'appeler doneFn
      jQuery(element).fadeIn(1000, doneFn);
    },

    move: function(element, doneFn) {
      jQuery(element).fadeIn(1000, doneFn);
    },

    leave: function(element, doneFn) {
      jQuery(element).fadeOut(1000, doneFn);
    }
  }
}]
```

$animate

AngularJS expose `$animate` pour gérer cette mécanique dans les directives.

```
<greeing-box active="onOrOff">Hi there</greeting-box>
```

```
ngModule.directive('greetingBox', ['$animate', function($animate) {
  return function(scope, element, attrs) {
    attrs.$observe('active', function(value) {
      value ? $animate.addClass(element, 'on') ? $animate.removeClass(eleme
    });
  });
}]);
```

```
[greeting-box].on {
  transition: 0.5s linear all;
  background: green;
}
```

Tests

- Apporter une garantie (et sérénité) pour vous et votre client
- Se prémunir contre les régressions
- Faire la distinction entre exceptions gérées et les fugitives

Couverture

- Le nombre de tests n'est pas le seul facteur important de qualité.
- Une autre métrique, le taux de couverture indique le pourcentage de code testé.
- Istanbul, blanket

Tests unitaires

- plus abstrait
- On se situe au niveau *micro*
- Venir blinder les unités de code les plus atomiques possibles (fonction souvent)
- Faciliter par les fonctions pures

Pièges unitaires

- Tests non isolés, perturbant ou pertubés par les autres
- Les coercions *falsy* avec `==`
- Eviter de se rassurer avec des tests inutiles (ex: assert(2 < 3))
- Asynchrone

Tests fonctionnels

- plus concret
- On s'attarde cette fois ci au niveau *macro*
- Implique un grand nombre de composants qu'il faut souvent simuler
- Réalisation de scénarios d'utilisation

Pièges fonctionnels

- N'explorer que les chemins de nav classiques
- Travailler sur des fixtures trop fictives

Frameworks de test

- TDD, BDD
- Nomenclature ou DSL permettant de catégoriser ses tests
- qUnit, Mocha, Jasmine

Lanceurs de tests

- JavaScript pur : un runtime correct suffit (Node)
- Interaction avec le DOM : doit être lancé dans le navigateur (Karma)

Angular Mock

- Permet de faciliter le débuggage
- Attention inclus séparement
- `$httpBackend`
- Raccourcis globaux : `dump`, `module`, `inject`

Jasmine

http://jasmine.github.io

Suites de tests

- Utilisation de la fonction `describe`
- Permettent de regrouper des unités logiques de test
- Peuvent s'imbriquer (pratique pour les modules)

```
describe("User module", function() {

  // …

  describe("User controller", function() {
    // …
  });

  describe("User service", function() {
    // …
  });
});
```

Spécifications

- Utilisation de la fonction `it`
- Les specs sont contenues dans les suites
- Se focalisent sur une règle métier précise

```
describe("User module", function() {

  it("should declare all sub-modules", function() {
    // …
  });

  describe("User controller", function() {

    it("should init with the right sorting", function() {
      // …
    });

  });

});
```

Setup

- Initialise l'environnement nécessaire aux tests
- `beforeAll`: pour chaque suite
- `beforeEach`: pour chaque spec

```
describe("User module", function() {
  var users;
  beforeEach(function() {
    users = ["Lilian", "Nicolas"];
  });
  // …
});
```

Teardown

- Nettoie, déconnecte proprement (mémoire)
- `afterAll`: pour chaque suite
- `afterEach`: pour chaque spec

Exclusion

- Pour désactiver certains tests, on peut préfixer d'un "**x**"
- `xdescribe`
- `xit`

Focus

- Inversement, on peut priviligier une séquence avec un "**f**"
- `fdescribe`
- `fit`

Assertions

- Sont contenues dans les specs
- Comparaisons entre un résultat obtenu et une valeur attendue
- Lancent des exceptions

```
describe("User controller", function() {

  it("should init with the right sorting", function() {
    expect(controller.sorting).to.be.a('string');
    expect(controller.sorting).to.equal('id');
  });

});
```

Expectations

- Une assertion `true` ou `false`
- 3 parfums, de plus brute au plus litteraire : `assert`, `expect`, `should`
- On peut choisir Chai mais Jasmine propose déjà ses outils

```
assert.lengthOf(king.musketeers, 3);
expect(king).to.have.property('musketeers').with.length(3);
king.should.have.property('musketeers').with.length(3);
```

Matchers

- Comparaisons booléenes ou mathématiques
- équalité (valeur ou référence)
- `<` `<=` `=>` `>`
- négation, personnalisée selon le type

```
expect(string).toBeSameLengthAs(other);
expect(number).toBeOddNumber();
expect(array).toBeArrayOfNumbers();
expect(element).toBeHtmlTextNode();
```

Espions

- Enveloppent fonctions et méthodes
- Enregistrent le nombre d'appels, les params passés etc...
- Sinon.js (mais certains aussi fournis par Jasmine)

```
it("should call subscribers on publish", function () {
  var callback = sinon.spy();
  PubSub.subscribe("message", callback);

  PubSub.publishSync("message");

  assert(callback.called); // true
  assert(callback.calledOnce); // true
});
```

Asynchrone

- Se déclenche avec la présence d'un **callback** dans la signature
- Attention aux timeouts trop longs

```
describe("Async", function() {
  var value;

  beforeEach(function(done) {
    setTimeout(function() {
      value = 0;
      done();
    }, 1);
  });

  it("should support async execution", function(done) {
    value++;
    expect(value).toBeGreaterThan(0);
    done();
  });

});
```

http://karma-runner.github.io

(ex Testacular)

Karma CLI

- Pas vraiment besoin d'installer la version globale
- Utiliser les scripts npm pour lancer la version locale

Plugins pour Karma

- adaptateurs avec les frameworks (Mocha, Jasmine...)
- lanceurs de navigateurs (Fx, Chrome, SlimerJs, PhantomJs...)
- reporters (terminal, HTML...)

Karma init

- Wizard créant la conf
- commiter karma.json

Karma conf

- ./test -> dossier par défaut
- ne pas oublier d'inclure *angular.mock*

Lancement

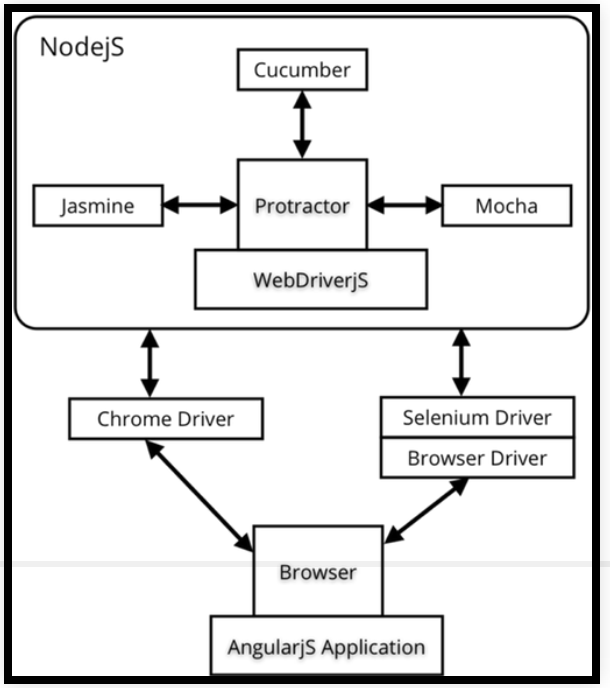
- d'abord `karma start` pour lancer le serveur
- puis `karma run` pour démarrer les tests

http://www.protractortest.org

Protractor

- Framework pour les tests **e2e**
- Un module **node.js**
- Se base sur WebDriverJS
- Peut se brancher sur Jasmine, Mocha, Cucumber

Inner working



Installation

- `npm i -g protractor`
- `webdriver-manager update`

Config

```
exports.config = {
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['angular-homepage.spec.js', 'tests/*.spec.js'],
  jasmineNodeOpts: {
    showColors: true
  }
};
```

DOM queries

Avec des selecteurs CSS (aka jQuery style)

- `element()`
- `element.all()`

```
element(by.css('#album-list'))
```

Ciblage des directives AngularJS

```
element(by.binding('album.title'))
```

```
<span ng-bind="album.title"></span>
```

```
element(by.model('album.title'))
```

```
<input ng-model="album.title">
```

Avec AngularJS repeater

```
element(
    by.repeater('album in albums').row(0).column('title')
)
```

```
<ul>
  <li ng-repeat="album in albums">
      <span>{{ album.title }}</span>
  </li>
</ul>
```

Simuler des événements

```
element(by.model('model')).click()
element(by.model('model')).sendKeys("Hello !", protractor.Key.ENTER)
```

Mocking

```
it('should mock the answer', function() {

    browser.addMockModule('httpBackendMock',
    function() {
        angular.module('httpBackendMock', ['app', 'ngMockE2E'])
        .run(function($httpBackend) {
            $httpBackend.whenGET('/results').respond('hello');
        });
    });

    browser.get('/');
    var result = element(by.binding('result'));
    expect(result.getText()).toEqual('hello');
});
```

httpbackend plus simple

```
var backend = new require('httpbackend')(browser);

it('should mock the answer', function() {
    backend.whenGET(/result/).respond('hello');

    browser.get('/');

    var result = element(by.binding('result'));
    expect(result.getText()).toEqual('hello');
});
```

Page pattern

```
// page/homepage.js
var AngularHomepage = function() {
  this.nameInput = element(by.model('yourName'));
  this.greeting = element(by.binding('yourName'));

  this.setName = function(name) {
    this.nameInput.sendKeys(name);
  };

  browser.get('http://www.angularjs.org');
};
```

Page pattern

```
//angular-homepage.spec.js
var AngularHomepage = require('page/homepage.js'),
describe('angularjs homepage', function() {
  it('should greet the named user', function() {
    var page = new AngularHomepage();

    page.setName('Lilian');

    expect(page.greeting.getText()).toEqual('Hello Lilian!');
  });
});
```

Soulager le DOM

`$compiler` ajoute des classes CSS pour faciliter le débug :

- ng-binding
- ng-scope
- ng-isolated-scope

```
<h2>Bienvenue {{user.name}}!</h2>
<p>Votre panier contient <strong ng-bind="cart.items.length"></strong> arti
```

```
<h2 class="ng-binding">Bienvenue Jeanne!</h2>
<p>Votre panier contient <strong class="ng-binding" ng-bind="cart.items.ler
```

En production

```
app.config(['$compileProvider', function ($compileProvider) {
  $compileProvider.debugInfoEnabled(false);
}]);
```

Peut être réactivé temporairement en tapant dans la console :

`angular.reloadWithDebugInfo()`

Simplifier le $digest

Lorsque plusieurs réponses **http** arrivent presque en même temps, il peut être pertinent de les traiter durant la même boucle de `$digest`

```
app.config(function ($httpProvider) {
  $httpProvider.useApplyAsync(true);
});
```

Conseils

Toujours utiliser la **version non minifiée** durant le dev.

Cette dernière est plus verbeuse sur les exceptions et stack trace.

Ne pas hésiter à **blackboxer** angular.js dans les devtools pour se concentrer sur votre code applicatif.

angular-hint

Des outils génériques comme **ESLint** signalent les erreurs statiques en amont.

angular-hint est spécifique à *AngularJS* et détectent les soucis courants au runtime.

Ce module est facultatif, ne pas oublier de l'inclure après angular.js :

```
npm i angular-hint
```

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

angular-hint modules

- **angular-hint-controllers**: global & nommage
- **angular-hint-directives**: fautes (ng-shwo -> ng-show)
- **angular-hint-dom**: APIs DOM dans les ctrls
- **angular-hint-events**: variables non déf dans les evts
- **angular-hint-interpolation**: variables non déf
- **angular-hint-modules**: namespace, non déf

ng-hint

```
<body ng-app="app" ng-hint>
```

```
<body ng-app="app" ng-hint-include="dom events">
```

```
<body ng-app="app" ng-hint-exclude="controllers">
```

Convention de nommage

- **camelCase**: JS idiomatique - `mapColors`, `trackLength`
- **PascalCase**: Constructeur - `UserService`, `InvoiceController`
- **kebab-case**: HTML, CSS - `ng-model="name"`, `text-align: center`

Propriétés AngularJS

- **$**: built-in, ne pas utiliser ce préfixe
- **$$**: privée, ne doit pas être manipulée directement

Bloggers à suivre

- Todd Motto
- John Papa
- Pascal Precht
- EggHead
- YearOfMoo
- Shidhin C R

Merci