ANGULARJS
PRISE EN MAIN

Durée : 2 journées
Objectif : Maitriser les bases d'AngularJS

SOMMAIRE

JOUR 1

SOMMAIRE

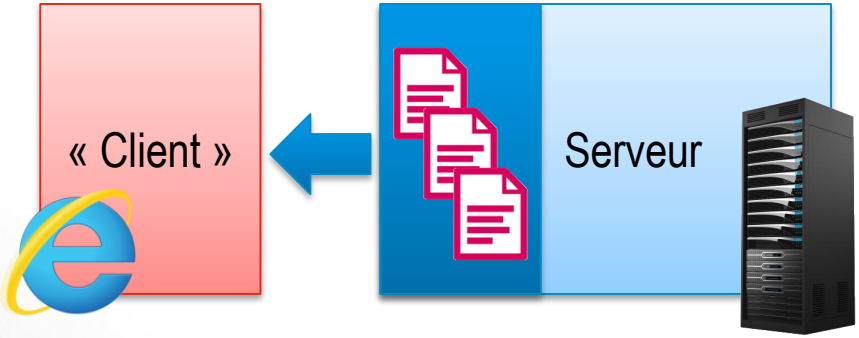
JOUR 2

SQLI GROUP

INTRODUCTION À L'ARCHITECTURE WEB D'AUJOURD'HUI

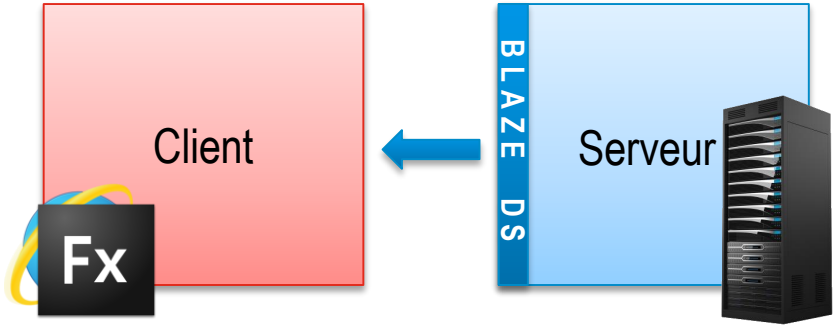
SINGLE PAGE APPLICATION

ALTERNATIVES



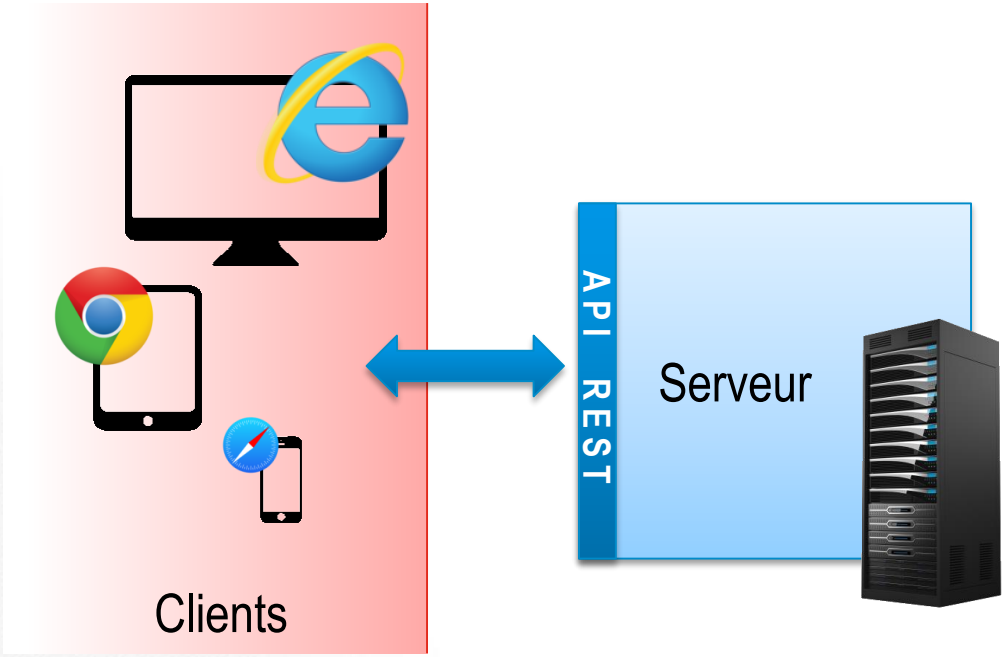
IHM GÉNÉRÉE ET GÉRÉE CÔTÉ SERVEUR

IHM CÔTÉ CLIENT, NON STANDARD



SINGLE PAGE APPLICATION

L'IHM CÔTÉ CLIENT BASÉE SUR LES STANDARDS DU WEB



+ Application Web constituée d'une seule page HTML (et de ressources Javascript, CSS...)
+ Routage, changement de vue géré côté client
+ Echanges avec le serveur limités aux données

SINGLE PAGE APPLICATION

AVANTAGES

+ Expérience utilisateur
+ Performance
+ Compétences dédiées
+ Interopérable
+ URL-friendly

REST

EXPOSITION DE SERVICES SIMPLE ET INTEROPÉRABLE

+ REST (REpresentational State Transfer) n'est pas un protocole ou un standard, mais un style d'architecture pour concevoir des applications pour les systèmes hypermédia.

+ REST repose sur les standards URI et HTTP (et liens hypertextes, média types, etc.)

  Le Web est un espace d'information dans lequel les données -Ressources- sont désignés par des identificateurs globaux -URI- ("Uniform Resource Identifier"). Les ressources sont manipulées à l'aide des verbes HTTP (GET, POST, PUT, DELETE...)

+ La technologie d'implémentation d'un serveur (ou d'un client) REST n'importe pas (.NET, Java, PHP, C, Ruby, Python, Node.js, etc.). Le respect des principes d'architecture permet aux architectures REST d'être interopérables.

SQLI GROUP

PRÉSENTATION D' ANGULAR JS

HISTORIQUE ET OBJECTIFS

- AngularJS est un framework open-source pour le langage JavaScript créé par Google
- Il a pour but de faciliter la création d'applications web modernes, à page unique
- Il s'appuie sur des principes déclaratifs et propose d'étendre le langage HTML
- AngularJS se veut extensible et pleinement testable
- Il a plus de 5 ans d'existence, le projet a débuté en 2009
  - 20 OCTOBRE 2010: VERSION 0.9.0
  - 13 JUIN 2012: VERSION 1.0.0
  - 8 NOVEMBRE 2013: VERSION 1.2.0
  - 13 OCTOBRE 2014: VERSION 1.3.0

PRINCIPES

DÉCLARATIF VS IMPÉRATIFS

+ **Impératif** : principe de programmation décrivant les opérations de façon séquentielle afin qu'elles soient exécutées dans un certain ordre par la machine pour modifier l'état du programme
    - Exemple : **jQuery**

+ **Déclaratif:** principe de programmation consistant à créer des applications sur la base de composants unitaires indépendant du contexte global, mais dépendant uniquement des arguments qui lui sont passés
    - Exemple : **AngularJS**

+ La programmation déclarative permet de décrire le « quoi » (texte, titres, paragraphes), quand la programmation impérative permet de décrire le « comment » (positionnement, couleurs, polices de caractères)

PRINCIPES

MV*

+ AngularJS se base sur un pattern d'architecture MVVM (Modèle-Vue-VueModèle)
  - Le concept a été défini à l'origine par Microsoft pour WPF (Windows Presentation Foundation) et Silverlight en 2005
  - Comme tout pattern MV*, il permet de séparer la vue, de la logique et de l'accès aux données
  - Il se base sur des bindings et des évènements pour faire communiquer les différentes couches

La catégorisation des patterns a assez peu d'importance, ce qui compte étant la séparation des logiques métier et de présentation de manière à faciliter la maintenance des applications.

PRINCIPES

INJECTION DE DÉPENDANCES

+ L'injection de dépendance (DI) est un pattern qui permet d'injecter de façon dynamique des dépendances, cela permet de :

  - Ne plus être responsable de la création des instances
  - Facilite l'écriture des tests unitaires
  - Changer ou charger les dépendances dynamiquement au runtime ou à la compilation
  - Améliore la maintenabilité et la compréhension du code

PRINCIPES

MODULES

+ La plupart des applications sont conçues avec un point d'entrée principal qui instancie, câble, et démarre l'application.
+ Pour AngularJS l'approche est différente : l'application est constituée de modules. Chaque module est responsable d'un périmètre technique ou fonctionnel; il ya plusieurs avantages à cette approche :
    - Le principe de déclaration rend le code plus simple à comprendre → maintenabilité
    - Pas besoin de charger tous les modules lors des tests unitaires, ce qui facilite leur production
    - Facilite le packaging de code sous forme de composants externes réutilisables → capitalisation
    - Le chargement des modules peut se faire dans n'importe quel ordre ou même être chargés en parallèle (de par la nature asynchrone de l'exécution des modules)

SQLI GROUP

PREMIERS PAS AVEC ANGULAR JS

CRÉER UNE APPLICATION

DIRECTIVE NG-APP

+ Démarrage (bootstrap) automatique : directive **ng-app** (un seul ng-app par application)

```
<html ng-app>
```

+ Indique l'élément **racine de l'application** : page ou portion

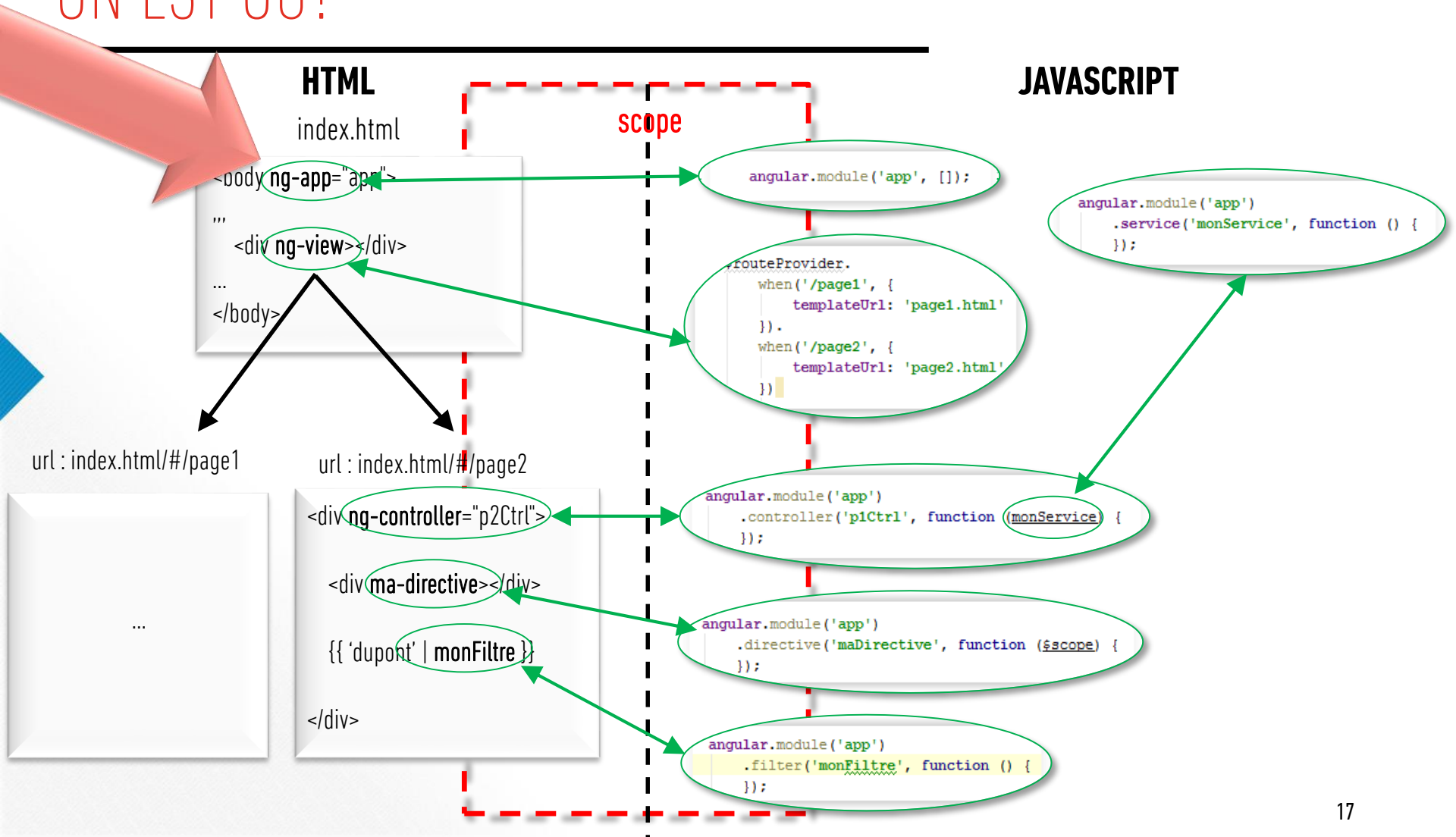
+ Peut indiquer le nom du **module d'application**

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

+ Bootstrap manuel : pas de déclaration d'attribut ng-app dans le HTML, initialisation par code

```
angular.bootstrap(document, ['myApp']);
```

ON EST OÙ?


HTML
index.html
scope
JAVASCRIPT
<body ng-app="app">
...
<div ng-view>
...
</body>
angular.module('app', []);
angular.module('app')
.service('monService', function () {
});
routeProvider.
when('/page1', {
templateUrl: 'page1.html'
}).
when('/page2', {
templateUrl: 'page2.html'
})
url : index.html/#/page1
...
url : index.html/#/page2

<div ma-directive>
{{ 'dupont' | monFiltre }}

angular.module('app')
.controller('p1Ctrl', function (monService) {
});
angular.module('app')
.directive('maDirective', function ($scope) {
});
angular.module('app')
.filter('monFiltre', function () {
});

CRÉER UNE APPLICATION

INCLUSION JS

+ Référencer la librairie JS

```
<script src="lib/angular/angular.js"
```

+ Enregistre les callbacks qui seront exécutés lorsque la page sera entièrement chargée

+ Version **non-compressée** (développement) ou **minifiée** (prod), disponibles sur http://angularjs.org/

+ Multi-modules : les modules complémentaires sont à ajouter explicitement (routes, resources...)

MODÈLE ET VUE

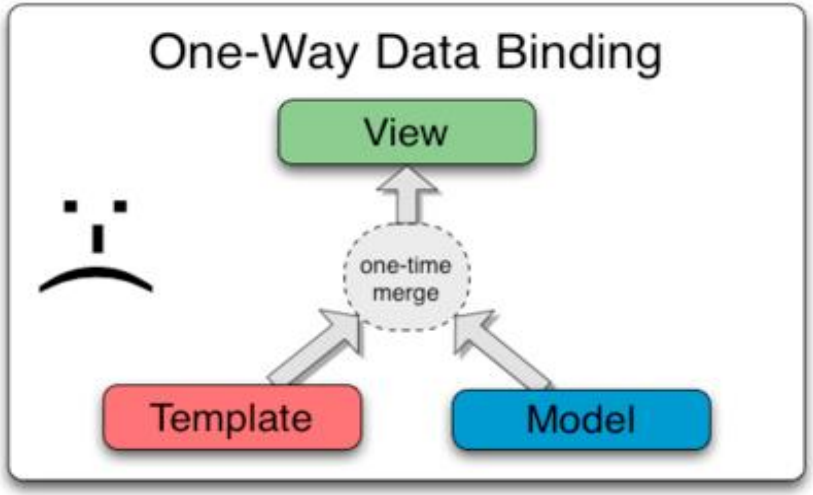
PRINCIPES DE FONCTIONNEMENT (1/2)

<u>Databinding Unidirectionnel</u>

+ Template + model = vue → merge
+ Changement de données → merge

Problèmes de cette approche :

+ Ecrasement des données utilisateurs
+ Association Vue vers le Model manuelle!



MODÈLE ET VUE

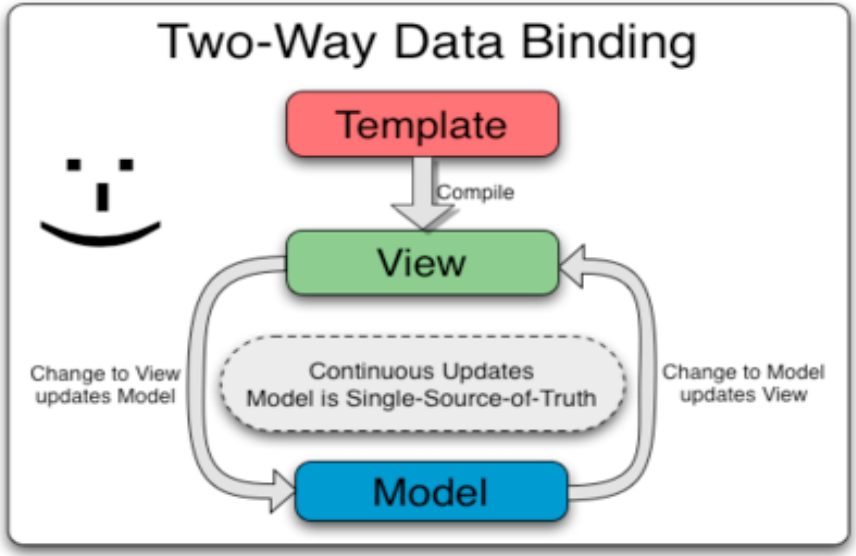
PRINCIPES DE FONCTIONNEMENT (2/2)

<u>Databinding Bidirectionnel</u>

+ Pas de consommation de *template*, mais de DOM
+ Compile : merge

Avantages de cette approche :

+ Vue dynamique
+ Binding transparent entre vue et modèle
+ Pas d'écrasement de données



MODÈLE ET VUE

DÉTAIL SUR LA VUE DYNAMIQUE

Compilation des vues

1. $compile traverse le DOM pour repérer les directives sur un élément
2. Les directives sont triées par priorités
3. Le template est lié avec le scope en appelant les fonctions de link créées
4. Binding dynamique (Live Binding) entre le scope et le DOM

A ce point, un changement dans le modèle sur le scope compilé **se reflète dans le DOM en continu** !

MODÈLE ET VUE

NG-MODEL

+ Directive pour lier une entrée utilisateur avec une propriété du scope courant

```
<input ng-model="name">
```

+ Valide sur **input** (text, checkbox, radio, number, email, url), **select**, **textarea**

MODÈLE ET VUE

OPÉRATEUR DE BINDING {{ ... }}

+ `{{...}}` : directive particulière pour le **binding** (liaison) de données

```
<p>hello {{ name }}</p>
```

+ Précise que l'**expression doit être évaluée** et que le résultat doit être **inséré dans le DOM**

+ Contient une **expression Angular**

SQLI GROUP

EXERCICES PRATIQUES

TP1. PREMIÈRE APPLICATION ANGULAR JS

INITIALISER L'APPLICATION BIBLIOTHÈQUE

- ✓ angular.js
- ✓ ng-app
- ✓ angular.module('bibliotheque',[]);

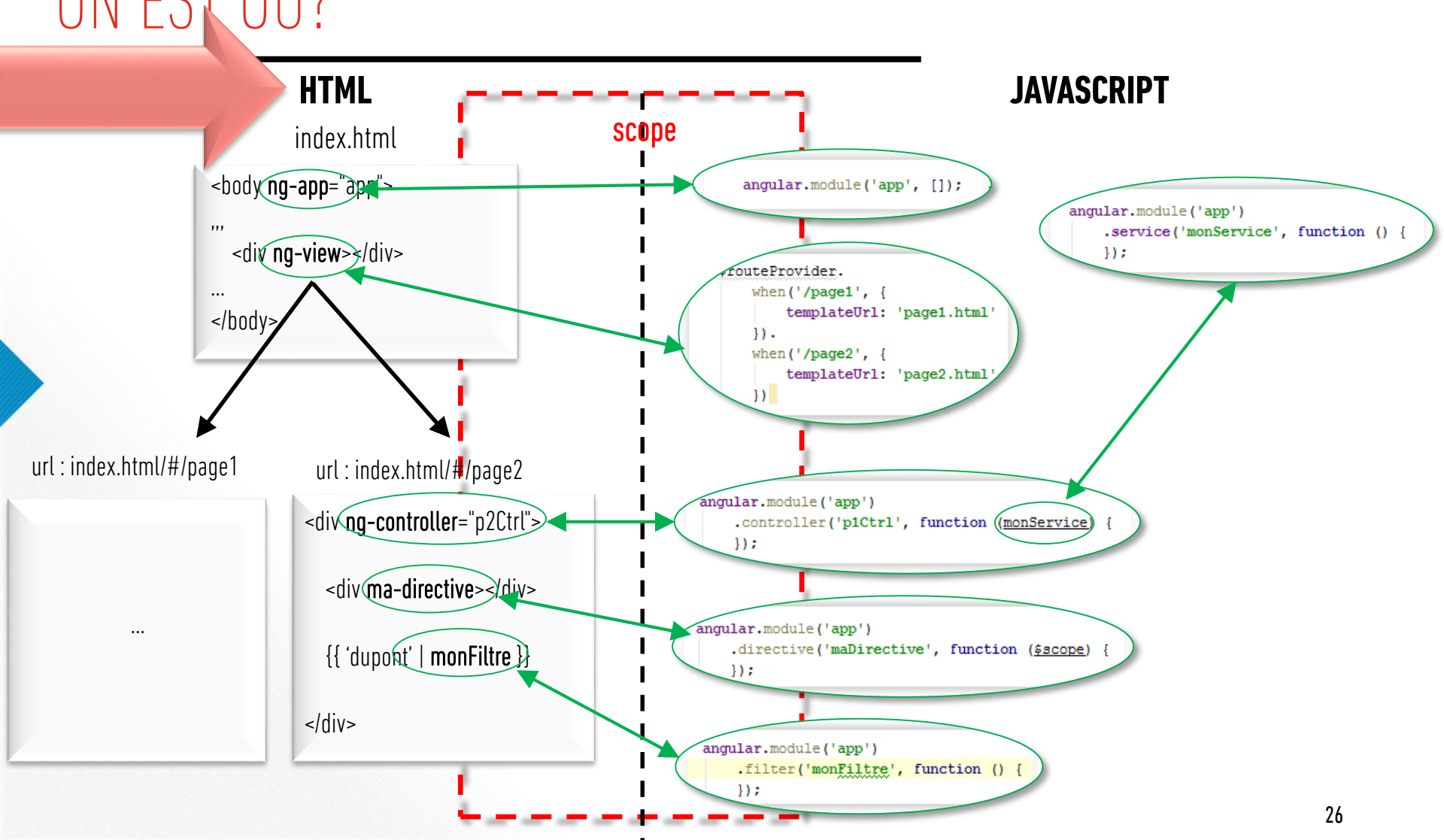
INTRODUCTION AUX VUES

PRÉSENTATION

+ La vue est la **projection du modèle** au travers du template HTML

+ Quand le modèle change, Angular rafraichit les bindings et met à jour la vue

ON EST OÙ?


HTML
index.html
<body ng-app="app">
...
<div ng-view>
...
</body>
scope
JAVASCRIPT
angular.module('app', []);
angular.module('app')
.service('monService', function () {
});
routeProvider.
when('/page1', {
templateUrl: 'page1.html'
}).
when('/page2', {
templateUrl: 'page2.html'
})
url : index.html/#/page1
...
url : index.html/#/page2

<div ma-directive>
{{ 'dupont' | monFiltre }}

angular.module('app')
.controller('p1Ctrl', function (monService) {
});
angular.module('app')
.directive('maDirective', function ($scope) {
});
angular.module('app')
.filter('monFiltre', function () {
});

INTRODUCTION AUX VUES

EXPRESSIONS ANGULAR

+ Les expressions utilisées dans le binding **{{...}}** sont JS-like, mais... ne sont pas des expressions JS. Ce sont des **expressions Angular**.

+ Elles sont :
  - **Sécurisées** : accès réduit au $scope et variables locales
  - **Protégées** pour les types **undefined** ou **null** : évite les TypeError
  - **Indépendantes des flots de contrôles** : pas de if, for, throw. Seuls certains appels sont autorisés,

```
{{ 1 + 2 }}
{{'hello' + 'world'}}
{{user.name}}
{{product.item.name}}
```

INTRODUCTION AUX VUES

OPÉRATEURS D'EXPRESSIONS ANGULAR

- **Mathématiques :** + - * / %
- **Binaires :** & | ^
- **Booléens :** && || !
- **Comparaisons :** == === != !== < > <= >=
- **Affectation :** =
- **Mot-clés :** null, true, false, undefined
- **Propriétés et méthodes, tableaux**
- **Filtres** lowercase,...

- **Non disponibles :** eval(), if, while, etc.

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – DIRECTIVES D'AFFICHAGES

+ **ng-show / ng-hide** : affiche / cache un élément selon une condition

```
<p ng-show="isVisible">affiché</p>
```

+ **ng-if** : crée un élément DOM selon une condition

```
<p ng-if="isVisible">présent</p>
```

+ **ng-switch on** : alterne l'élément inclut selon la condition

```
<div ng-switch on="isVisible">
    <div ng-switch-when="true">Affiché</div>
    <div ng-switch-default>default</div>
</div>
```

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – DIRECTIVES D'AFFICHAGES

+ **ng-messages** : affiche / cache un ensemble d'éléments selon l'existence de sous éléments

```
<div ng-messages="myForm.myName.$error">
    <div ng-message="required">You did not enter a
field</div>
    <div ng-message="minlength">Your field is too
short</div>
    <div ng-message="maxlength">Your field is too long</div>
  </div>
```

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – TABLEAUX

+ **ng-repeat** : répète la création d'un élément DOM
  Permet de parcourir les collections

+ Attention **ng-repeat** entraine la création d'un **$scope enfant** : les variables créées dans l'expression **ng-repeat** ne sont visibles que dans ce scope

```
<div ng-repeat="item in [0,1,2]">
    <p>Item</p>
    <p>N°{{item}}</p>
</div>
```

+ Des propriétés spéciales sont accessibles dans le scope local du repeater
  - **$index** : nombre
  - **$first, $middle, $last, $even, $odd** : booléens

+ Possibilité de séparer le repeat sur 2 blocs avec **ng-repeat-start** et **ng-repeat-end**

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – INCLUSION DE PAGE EXTERNE

+ **ng-include** : inclusion de HTML partiel (template) dont l'appel se traduit toujours de la même façon quelque soit la méthode d'inclusion

```
<div ng-include="'chemin_vers_la_page_html/page1.html'"></div>
```

+ La directive se base sur « chemin_vers_la_page_html/page1.html » qui sera

  - soit un chemin répondant à une requête http **http://hostname/chemin_vers_la_page_html/page1.html** dans le cas d'une inclusion externe

  - soit un **identifiant** permettant à la directive de récupérer le contenu html du template auprès d'un service de cache : **$templateCache**

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – RÉPONSES À DES ACTIONS UTILISATEURS

+ **ng-click** : permet de spécifier un comportement spécifique quand un élément est cliqué.

```
<img ng-click="count = count + 1"></div>
```

```
<img ng-click="add()"></div>
```

ADD ÉTANT UNE FONCTION PUBLIÉE DANS LE SCOPE

+ Voir aussi ng-dlbclick, ng-focus, ng-keydown, ng-keypress, ng-keyup, ng-change, ng-check...

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – CLASSE CSS

+ **ng-class** : modifie dynamiquement la classe CSS d'un élément par databinding

+ Peut prendre plusieurs syntaxes
  - Valeur unique : **style**
  - Tableau : **[style1, style2, style3]**
  - Objet contenant des tuples **style:test** : la classe **style** est appliquée si **test** est vrai.

```
<div ng-class="{style1:test1, style2:test2}">…</div>
```

+ Peut également être pris en compte au sein de l'attribut class

```
<div class="ng-class:{style:test}">…</div>
```

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – INCLUSION D'IMAGES

- **ng-src** : liaison sur l'attribut **src** de l'image
- Valeur : template à évaluer

- Permet de fournir un nom d'image dynamique, qui sera construit avant sa récupération

```
<img ng-src="image{{id}}.jpg"></div>
```

- Permet d'effectuer le binding avant que le navigateur n'essaie de télécharger l'image

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – INVISIBILITÉ AVANT CHARGEMENT

+ **ng-cloak** : n'affiche pas l'élément tant que le template n'est pas compilé

+ Evite que des éléments (artefacts visuels) apparaissent furtivement à l'affichage d'une page tant que l'application n'est pas entièrement chargée

  - Utile pour la page index.html et le template initial

```
<div id="template1" ng-cloak>{{ 'hello' }}</div>
```

INTRODUCTION AUX VUES

DIRECTIVES LES PLUS COURANTES – INVISIBILITÉ AVANT CHARGEMENT

+ **ng-bind** : similaire à **{{ }}** sauf que le binding reste invisible tant que la page n'est pas chargée et le template compilé (même effet que **ng-cloak**)

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

+ **ng-bind-template** : permet en plus de spécifier un template

```
<pre ng-bind-template="{{salutation}} {{name}}!"></pre>
```

SQLI GROUP

EXERCICES PRATIQUES

TP2. VUES ET EXPRESSIONS

AJOUTER 3 VUES

- ✓ ng-include & ng-if
- ✓ une vue « Home »
- ✓ une vue « Add »
- ✓ une vue « Dashboard »

SQLI GROUP

CONTRÔLEURS ET SCOPES

CONTRÔLEURS

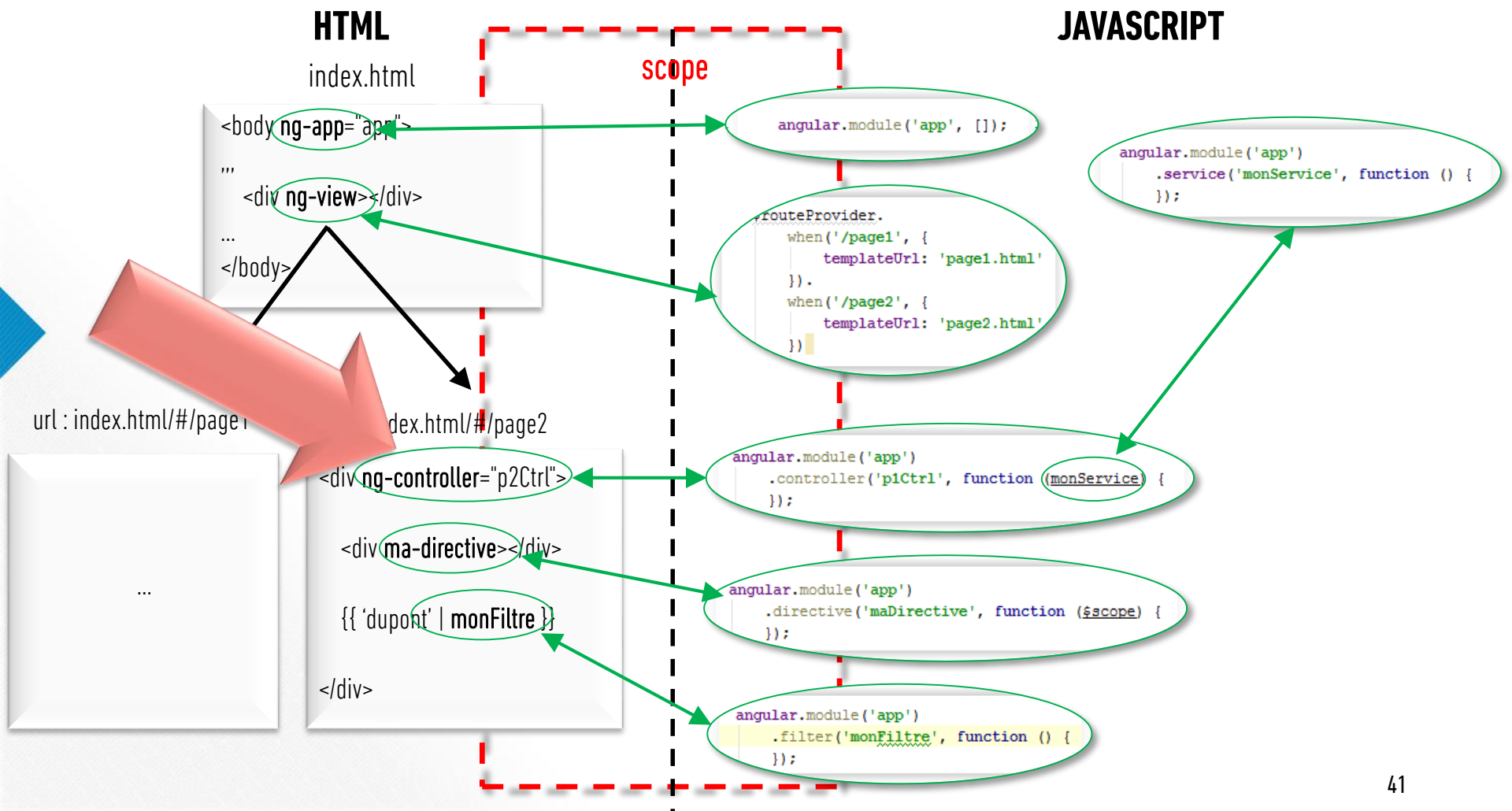
DÉFINITION

+ Le contrôleur contient la **logique de présentation**
  - Initialise l'état initial de la vue
  - Ajoute des comportements à la vue

+ La présence d'un contrôleur entraine la **création d'un scope** associé
  - Le scope est un objet qui **réfère au modèle applicatif** : le modèle présenté
  - Le scope est la **glue entre le contrôleur et la vue**

+ Les propriétés et fonctions définies au sein du **$scope** du contrôleur sont accessibles à tous les éléments du DOM du template sur lequel le contrôleur est enregistré

ON EST OÙ?


HTML
index.html
<body ng-app="app">
...
<div ng-view>
...
</body>
scope
JAVASCRIPT
angular.module('app', []);
angular.module('app')
.service('monService', function () {
});
routeProvider.
when('/page1', {
templateUrl: 'page1.html'
}).
when('/page2', {
templateUrl: 'page2.html'
})
url : index.html/#/page1
dex.html/#/page2
...

<div ma-directive>
{{ 'dupont' | monFiltre }}

angular.module('app')
.controller('p1Ctrl', function (monService) {
});
angular.module('app')
.directive('maDirective', function ($scope) {
});
angular.module('app')
.filter('monFiltre', function () {
});

CONTRÔLEURS

IMPLÉMENTATION

+ La directive **ng-controller** positionnée sur un élément et référence le nom du contrôleur

```
<div ng-controller="HelloCtrl">…
```

+ Le contrôleur est un module

```
app.controller('HelloCtrl', ['$scope', function($scope) {
    $scope.greeting = 'Bonjour!';
}]);
```

+ Par convention, suffixe en **Ctrl** ou **Controller**

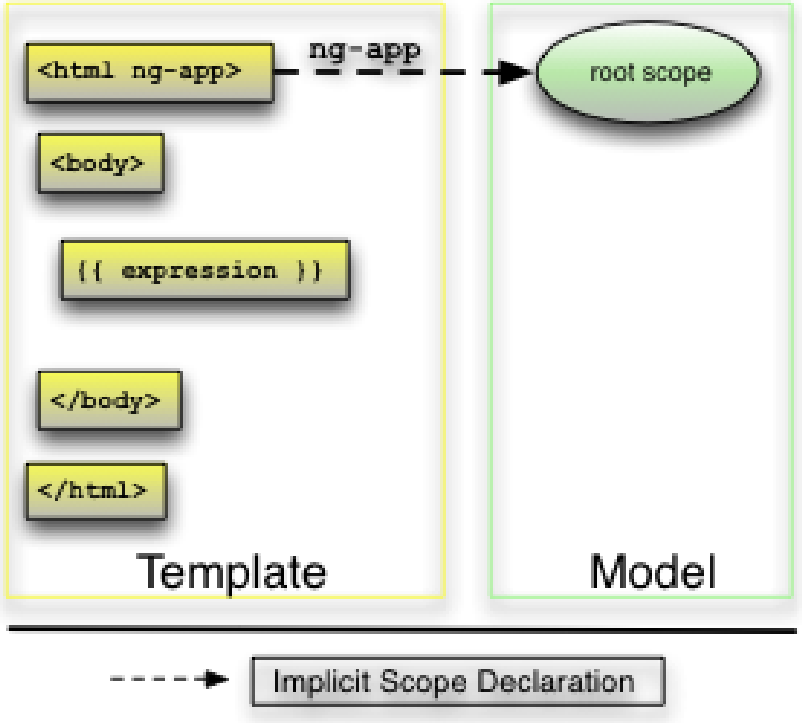
HÉRITAGE DE SCOPES

ROOT SCOPE

+ Le **root scope** est le contexte du modèle de données associé à l'application

+ **Scope** associé au **ng-app**

+ Il n'existe qu'un seul **root scope**



HÉRITAGE DE SCOPES

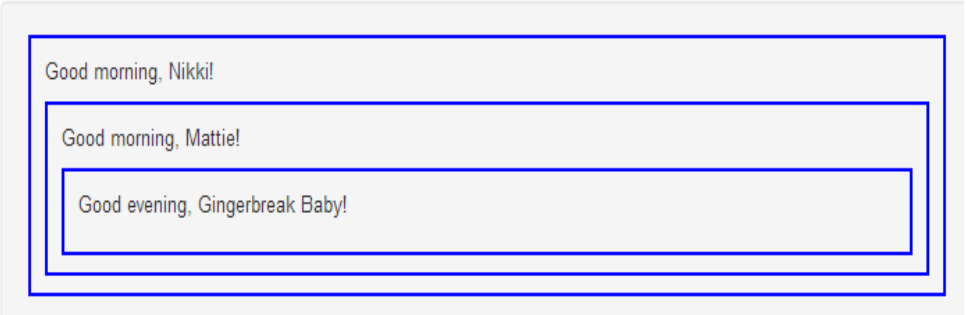
HÉRITAGE DE SCOPES

- Les contrôleurs peuvent être **créés à différents niveaux** dans la hiérarchie (imbriqués).
- On obtient une **hiérarchie de scope qui héritent les uns des autres**, jusqu'au **root scope** qui est le parent de tous les scopes

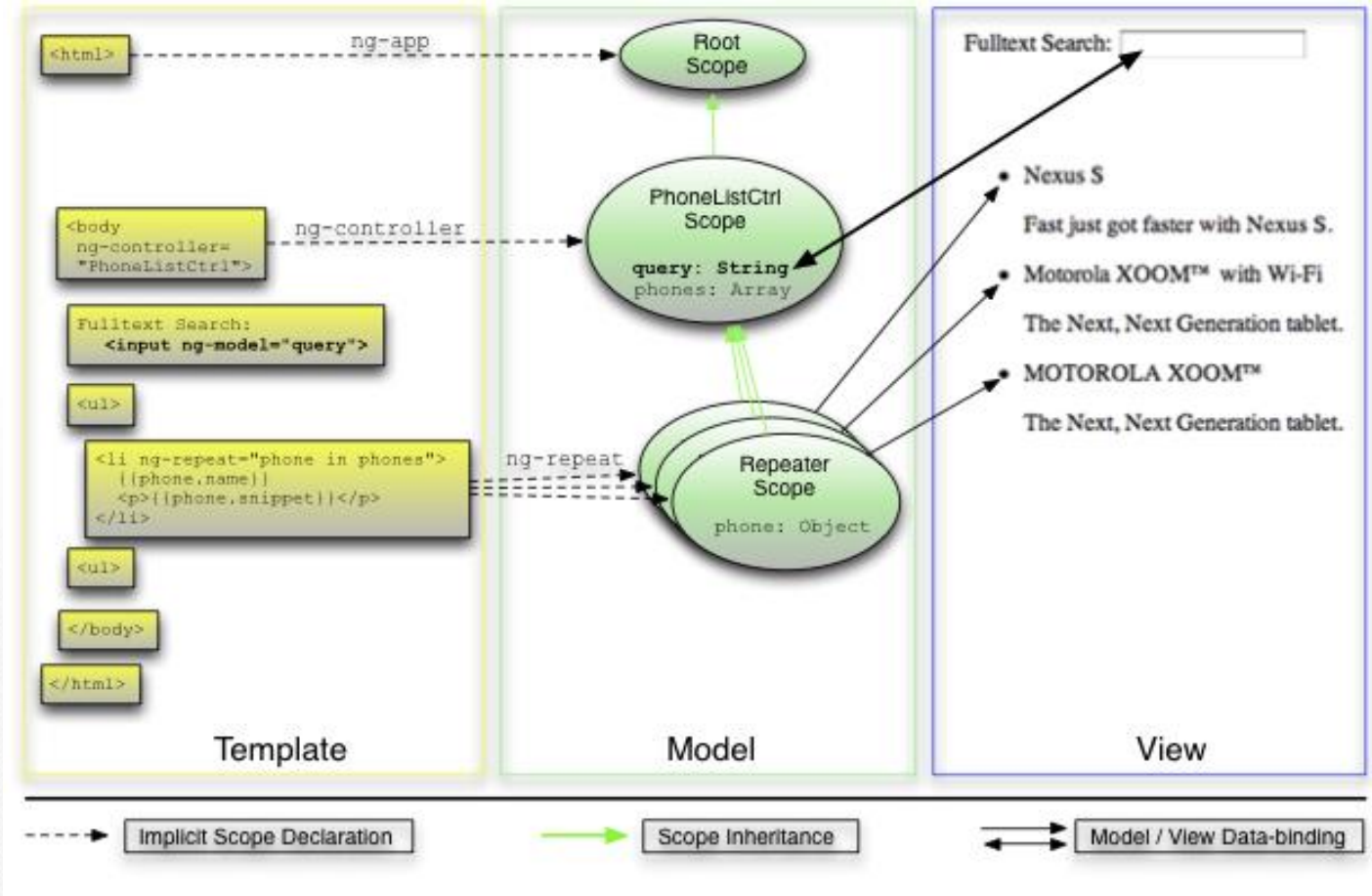
```
<div ng-app="scopeInheritance" class="spicy">
  <div ng-controller="MainCtrl">
    <p>Good {{timeOfDay}}, {{name}}!</p>

    <div ng-controller="ChildCtrl">
      <p>Good {{timeOfDay}}, {{name}}!</p>

      <div ng-controller="GrandChildCtrl">
        <p>Good {{timeOfDay}}, {{name}}!</p>
      </div>
    </div>
  </div>
</div>
```



HÉRITAGE DE SCOPES



SCOPE

IMPLÉMENTATION

+ Un scope permet au template, modèle et contrôleur de travailler ensemble : **isole modèles et vues tout en assurant leur synchronisation : data-binding**

+ La directive **ng-controller** est utilisée pour créer implicitement un scope pour un template

+ Ce scope sera enrichi dans le contrôleur à partir de **$scope**

```
app.controller('MovieCtrl',function($scope) {
    $scope.greeting = 'Bonjour!';
}
```

```
<div ng-controller="MovieCtrl">
    {{greeting}}
</div>
```

SQLI GROUP

EXERCICES PRATIQUES

TP3. VUES & CONTROLLEURS

AJOUTER 4 CONTRÔLEURS

- ✓ 1 contrôleur par vue
- ✓ 1 contrôleur pour traiter la navigation

CONTRE-INDICATIONS

A NE PAS FAIRE VIA UN CONTRÔLEUR

- Manipuler le DOM
  - Pour cela utiliser le databinding d'AngularJS ainsi que les directives pour encapsuler la manipulation de DOM
- Formater les entrées (de la vue vers le contrôleur)
  - Utiliser les contrôles de formulaires (type=« number », ng-pattern, etc.)
- Filtrer les sorties (du contrôleur vers la vue)
  - Utiliser les filtres
- Partager du code ou un état entre deux contrôleurs
  - Utiliser des services
- Gérer le cycle de vie d'autres composants (services...)
  - Utiliser l'injection de dépendance
- Beaucoup de code dans les contrôleurs
  - Le contrôleur ne doit qu'initialiser le scope et réagir aux actions utilisateurs. Le code « complexe » doit être déporté dans des services

SQLI GROUP

FORMULAIRES

FORM

PRESENTATION

+ **ngForm** : directive ajoutée implicitement sur l'élément **form**

+ Les différents inputs constituent le formulaire sur lesquels on associe des directives **ng-model**

```
<form novalidate>
    Name: <input type="text" ng-model="user.name" /><br />
    E-mail: <input type="email" ng-model="user.email" /><br />

    <button ng-click="reset()">RESET</button>
    <button ng-click="update(user)">SAVE</button>
</form>
```

+ **novalidate** : directive pour **désactiver la validation native** du formulaire par le navigateur

VALIDATION

HTML 5 & ANGULAR

- Requis
- Types : email, nombre, url, date
- Longueur : minimum, maximum
- Expression régulière

```
<input type="text" required />

<input type="email" name="email" ng-model="user.email" />
<input type="number" name="email" ng-model="user.age" />
<input type="url" name="homepage" ng-model="user.facebook_url" />

<input type="text" ng-minlength="5" />
<input type="text" ng-maxlength="20" />

<input type="text" ng-pattern=“/[a-zA-Z]/" />
```

VALIDATION

ETATS D'UN CHAMP ET D'UN FORMULAIRE

- Valid
  - Champ valide
  - Formulaire valide (tous les champs sont valides)
- Invalid
  - Champ invalide
  - Formulaire invalide (au moins un champ est invalide)
- Pristine
  - Champ intact
  - Formulaire intact (tous les champs sont intacts)
- Dirty
  - Champ modifié
  - Formulaire modifié (au moins un champ a été modifié)

VALIDATION

FORM CONTROLLER : FONCTIONS DISPONIBLES

- $pristine renvoie un boolean
- $dirty renvoie un boolean
- $valid renvoie un boolean
- $invalid renvoie un boolean

- **$error renvoie un object** : le champ contient des erreurs
  - Les erreurs sont indiquées en tant que propriétés de l'objet sous la forme clé/valeur

```
{
  required: false,
  pattern: true
}
```

VALIDATION

FORM CONTROLLER

+ **FormController**, présent sur la directive ngForm, fournit un ensemble d'API de validation.

+ Les propriétés sont disponibles sur les éléments de formulaire
  - Nécessite de déclarer l'attribut **name** des éléments

+ Il est possible de les utiliser pour afficher les messages d'erreur

```
<form name="myForm">
  <input type="email" name="uEmail" required/>
  <div ng-show="myform.uEmail.$dirty && myform.uEmail.$invalid">
    Invalid:
    <span ng-show="myform.uEmail.$error.required">Required Email</span>
    <span ng-show="myform.uEmail.$error.email">Invalid email.</span>
  </div>
</form>
```

VALIDATION

CLASSES CSS

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

```
<div>
  <input type="text" name="username" ng-model="username" required />
  <span class="ok">☑</span>
  <span class="ko">☒</span>
</div>
```


```
.ng-pristine { border:1px solid blue; }
.ng-dirty.ng-valid { border:1px solid green; }
.ng-dirty.ng-invalid { border:1px solid red; }
.ng-dirty.ng-valid ~ span.ok { color:green; display:inline; }
.ng-dirty.ng-invalid ~ span.ko { color:red; display:inline; }
```

VALIDATION

CONTRÔLES AVANCÉS

+ La création de contrôles de validation plus avancés nécessite la création de directives personnalisées

```
app.directive('ensureUnique', ['$http', function($http) {
  return {
    require: 'ngModel',
    link: function(scope, elt, attrs, ctrl) {
      scope.$watch(attrs.ngModel, function() {
        $http({
          method: 'POST',
          url: '/api/check/' + attrs.ensureUnique,
          data: {‘value': ctrl.$viewValue}
        }).success(function(data, status, headers, cfg) {
          ctrl.$setValidity('unique', data.isUnique);
        }).error(function(data, status, headers, cfg) {
          ctrl.$setValidity('unique', false);
        });
      });
    }
  }
}]);
```

Cf. cours sur les directives

VALIDATION

CONSERVATION DE SAISIE

+ Dans certains cas il peut être utile de sauvegarder un l'objet du formulaire. Pour ne pas travailler directement sur un objet « propre ».

```
function Controller($scope) {
    $scope.user= {name:'DUPONT', email:'dupont@orange.fr'};

    backup = angular.copy($scope.user);

    $scope.reset = function() {
        $scope.user = backup;
    };
 }
```

+ A noter l'utilisation de **angular.copy()** qui assure une copie « profonde »

SQLI GROUP

EXERCICES PRATIQUES

TP4. VALIDATION DE FORMULAIRES

METTRE EN PLACE LE FORMULAIRE D'AJOUT

- ✓ Titre : requis, > 3 caractères
- ✓ ISBN : requis, 13 chiffres
- ✓ Catégorie : requis
- ✓ Auteur – Prénom : requis
- ✓ Auteur – Nom : requis
- ✓ Editeur – Nom : requis, > 3 caractères
- ✓ Image (url) : requis
- ✓ Résumé : requis

SQLI GROUP

COMMUNICATION SERVEUR

COMMUNICATION SERVEUR

SERVICE $HTTP

+ Service facilitant la communication HTTP avec un serveur via XmlHttpRequest (AJAX) ou JSONP

+ Prend en paramètre un objet de configuration pour générer une requête HTTP

```
$http({method:'GET', url:'/someUrl', …})
```

+ Plusieurs fonctions raccourcies sont disponibles

```
$http.get
$http.post
$http.put
$http.delete
$http.head
$http.jsonp
```

COMMUNICATION SERVEUR

SERVICE $HTTP : FONCTIONS DE CALLBACK

+ Résultat récupéré via les fonctions asynchrones **success** et **error**

```
$http({method: 'GET', url: '/someUrl'})
    .success(function(data, status, headers, config) {
      // this callback will be called asynchronously
      // when the response is available
    })
    .error(function(data, status, headers, config) {
      // called asynchronously if an error occurs
      // or server returns response with an error status
    });
```

+ A noter l'existance d'un **.then()**. Cf. cours sur les promesses.

SQLI GROUP

EXERCICES PRATIQUES

TP5. COMMUNICATION SERVEUR

AFFICHER LES LIVRES SUR LA « HOME »

- ✓ $http
- ✓ Flux : http://localhost/back/books
- ✓ ng-repeat

COMMUNICATION REST

SERVICE $RESOURCE

+ **$resource** est une variante de **$http** pour accéder à des ressources REST
  S'appuie sur les standards **RESTFUL** : URL identique mais accès avec les verbes HTTP

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

- **url** : url éventuellement paramétrée
- **paramDefaults** : valeurs par défaut des paramètres de l'url, sous forme de paires clé/valeur
  Peuvent être surchargées dans le paramètre actions
  Mappés sur les paramètres définis dans l'url, les autres sont ajoutés en tant que search query
- **actions** : actions personnalisées, sous forme de liste d'objets de configuration http

```
var CreditCard = $resource('/user/:userId/card/:cardId',
      {userId:123, cardId:'@id'}, {
       charge: {method:'POST', params:{charge:true}}
      });
```

COMMUNICATION REST

OBJET RESSOURCE

+ **$resource** retourne un objet avec les fonctions suivantes

```
{
  'get':    {method:'GET'},
  'save':   {method:'POST'},
  'query':  {method:'GET', isArray:true},
  'remove': {method:'DELETE'},
  'delete': {method:'DELETE'}
}
```

+ Ces fonctions peuvent être personnalisées

```
$resource('phones/:phoneId.json', {}, {
    query: {
          method:  'GET',
          params:  {phoneId:'phones'},
          isArray: true
    }
});
```

COMMUNICATION REST

OBJET RESSOURCE

+ Paramètres des fonctions « **statiques** » de type **GET**

```
Resource.<action>([parameters], [success], [error])
```

+ Paramètres des fonctions « **statiques** » de type **NON GET**

```
Resource.<action>([parameters], postData, [success], [error])
```

+ Paramètres des fonctions « **d'instance** » de type **NON GET**

```
instance.<$action>([success], [error])
```

```
var User = $resource('/user/:userId', {userId:'@id'});
User.get({userId:123}, function(user) {
   user.abc = true;
   user.$save();
});
```

COMMUNICATION REST

POINT NOTABLE

+ En cas de binding d'un objet ressource
    - La ressource est assignée au modèle , qui est associée à la vue
    - Tant que l'objet est vide, la vue n'est pas dessinée : **chargement asynchrone** de la ressource
    - Une fois la donnée disponible, la **vue est automatiquement redessinée**

SQLI GROUP

EXERCICES PRATIQUES

TP6. COMMUNICATION REST

TRAITER TOUS LES FLUX

- ✓ $resource
- ✓ Home : GET « back/books »
- ✓ Add : POST « back/books »
- ✓ Update : POST « back/books/<id> »
- ✓ Delete : DELETE « back/books/<id> »

SQLI GROUP

SERVICES

SERVICES

INTRODUCTION

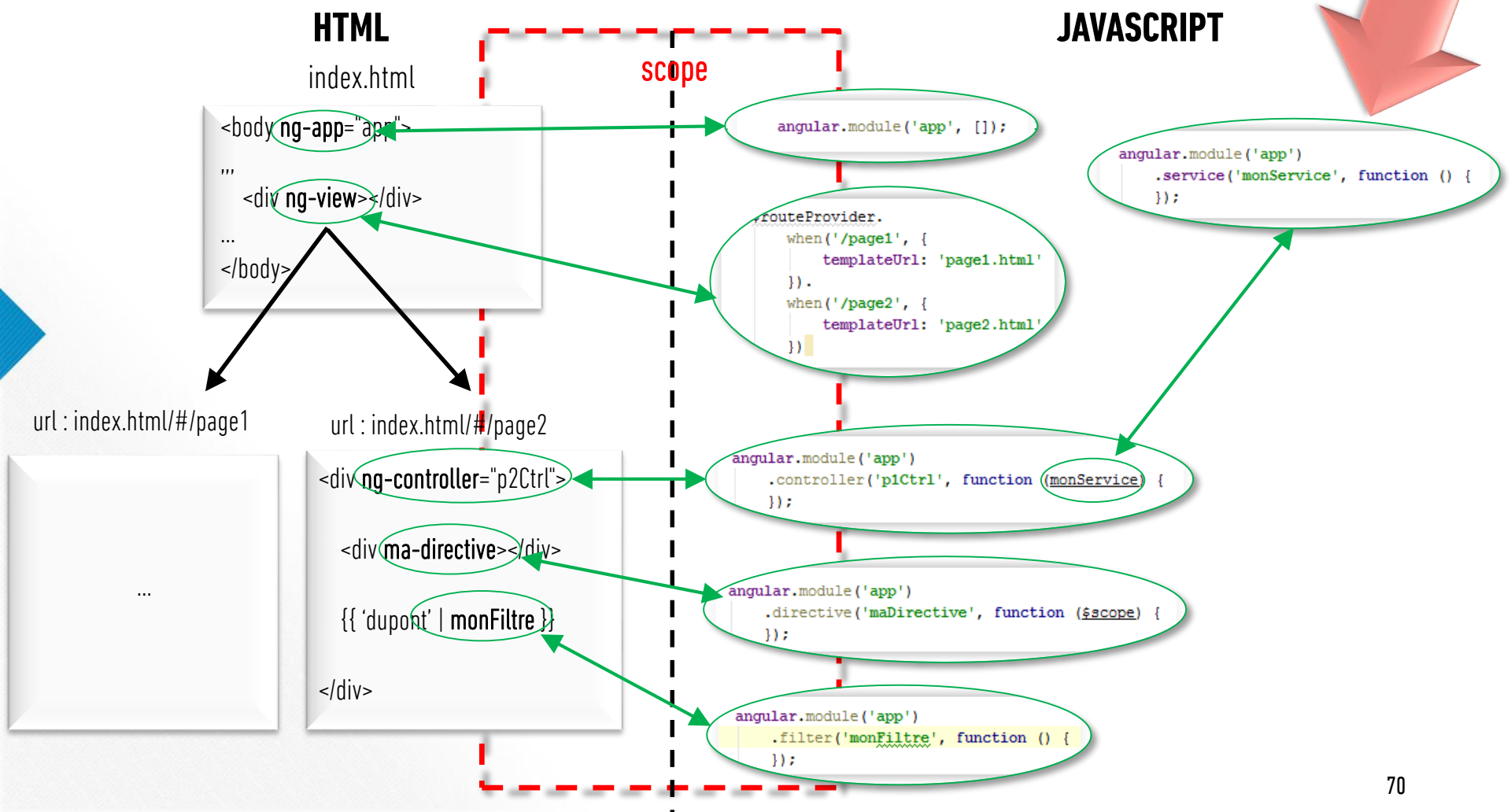
+ Les services sont des objets qui sont
  - gérés sous forme de **singletons** (cache géré par Angular)
  - **instanciés à la demande (lazy)** uniquement, via un **provider**
  - utilisés via l'**injection de dépendance** (géré dans Angular par le service **$injector**)

+ Ils permettent d'**organiser** et **partager** du code transverse, technique ou métier

+ AngularJS apporte son lot de services natifs, comme par exemple **$http** qui permet d'effectuer des appels Ajax. Tous les services AngularJS commencent par un « **$** »

+ Les services sont une pièce majeure d'une bonne structure applicative

ON EST OÙ?


HTML
index.html
scope
JAVASCRIPT
<body ng-app="app">
...
<div ng-view>
...
</body>
angular.module('app', []);
angular.module('app')
.service('monService', function () {
});
routeProvider.
when('/page1', {
templateUrl: 'page1.html'
}).
when('/page2', {
templateUrl: 'page2.html'
})
url : index.html/#/page1
...
url : index.html/#/page2

<div ma-directive>
{{ 'dupont' | monFiltre }}

angular.module('app')
.controller('p1Ctrl', function (monService) {
});
angular.module('app')
.directive('maDirective', function ($scope) {
});
angular.module('app')
.filter('monFiltre', function () {
});

SERVICES

INTRODUCTION

+ Angular propose 5 fonctions pour créer un service, mise à disposition au travers d'un **module**

  - **provider** : fonction de base, la plus complète, les autres sont des méthodes simplifiées pour les cas d'usage courants
  - **factory**
  - **service**
  - **value**
  - **constant**

SERVICES

PROVIDER

+ Objet utilisé pour instancier un service

+ Définit une méthode **$get** pour créer l'instance du service

+ Porte le nom du service suffixé par « **Provider** »
  - Le service Angular **$http** est associé au provider **$httpProvider**

+ Permet de configurer le service au travers de propriétés et méthodes personnalisées, lors de la phase de configuration du module associé

SERVICES

PROVIDER : CRÉATION

+ Méthode **provider(name, provider)**

```
module.provider(name, provider)
```

+ Paramètre **name** : nom du provider, **sans le suffixe « Provider »** (ajouté automatiquement par Angular)
+ Paramètre **provider**
  - **Objet** : utilisé comme instance du provider, doit définir la méthode $get , et les éventuelles propriétés et fonctions de configuration
  - **Fonction** : utilisée comme constructeur du provider, doit définir la méthode $get, et les éventuelles propriétés et fonctions de configuration, via **this**

```
module.provider('ApiClient', function (…) {
    this.$get = function (…) {
        ...
    };
});
```

SERVICES

FACTORY

+ Simplification de la fonction **provider**, pour les cas où aucune configuration spécifique n'est nécessaire

+ Méthode **factory(name, $getFn)**
  - Paramètre **name** : nom de la factory
  - Paramètre **$getFn** : fonction qui doit instancier le service, et qui sera utilisée en tant que fonction **$get** associée à un **provider généré automatiquement**

```
module.factory('ApiClient', function () {
    var apiClient = new ApiClient();
    ...
    return apiClient;
});
```

SERVICES

SERVICE

+ Similaire à la fonction factory, mais le second paramètre fait cette fois directement référence au constructeur du service. L'instanciation via new sera effectuée par Angular.

+ Méthode **service(name, constructor)**

```
module.service('ApiClient', function () {
    this.property1 = '...';
    ...
    this.function1 = function() { ... }
    ...
});
```

SERVICES

VALUE

+ Similaire aux fonctions précédentes, mais le second paramètre fait cette fois référence à l'instance existante du service

+ Méthode **value(name, value)**

```
module.value('ApiClient', {key:value, …});
```

+ Ne peut pas bénéficier d'injection d'autres services, ni de dépendances au reste de l'application, ni même au framework : l'objet sera donc relativement simple et autonome

SERVICES

CONSTANT

- Similaire à la fonction value, mais
  - l'objet en second paramètre est enregistré en tant que **son propre provider**, sous le **même nom que le service** (sans le suffixe « Provider »)
  - ce service **peut être configuré** lors de la phase de configuration du module
    Cf. ci-après.

- Méthode **constant(name, value)**

```
module.constant('ApiClientUrl', {key:value, …});
```

SERVICES

RÉCAPITULATIF

Méthode	Configurable	Injection de dépendances	Création du service	Nom du provider
module.provider()	Oui	Oui	Au premier accès, par appel de la méthode $get du provider	nom du service + 'Provider'
module.factory()	Non	Oui	Au premier accès, par appel de la fonction fournie	nom du service + 'Provider'
module.service()	Non	Oui	Au premier accès, en utilisant la fonction fournie comme un constructeur	nom du service + 'Provider'
module.value()	Non	Non	Le service est un objet préexistant	nom du service + 'Provider'
module.constant()	Oui	Non	Le service est un objet préexistant	nom du service

SERVICES

INJECTION DU SERVICE

+ Un service est utilisé, dans un contrôleur ou un autre service, via l'**injection de dépendance**

+ Angular gère l'injection de dépendances via une fonction **injector** de type « **service locator** »

```
angular.module('myModule', []).
  factory('greeter', function($window) {
    return {
      greet: function(text) { $window.alert(text); }
    };
  });

// New injector is created from the module.
// (This is usually done automatically by angular bootstrap)
var injector = angular.injector(['myModule', 'ng']);

// Request any dependency from the injector
var greeter = injector.get('greeter');
```

SERVICES

INJECTION DU SERVICE

- La déclaration des dépendances se fait généralement via des **annotations inline**
  - Instanciation et injection automatique via l'**injector**
  - Il est important de respecter le même **ordre dans la déclaration des dépendances** et la déclaration des paramètres à injecter, pour que l'injection fonctionne après **minification/obfuscation**

```
myApp.controller('ServiceController', ['$scope', 'myService',
    function($scope, myService) {
        $scope.users = myService.getAllUsers();
}]);
```

SQLI GROUP

EXERCICES PRATIQUES

TP7. SERVICES

IMPLÉMENTER LES SERVICES

- ✓ Exposition de la ressource « Book »
- ✓ Exposition de l'url du backend
- ✓ Service métier : calcul du dahboard
  → liste des auteurs, éditeurs, catégories (attention doublons)

SQLI GROUP

LA MODULARISATION

MODULARISATION

DÉCLARATION D'UN MODULE ANGULAR

+ Un module est défini par son nom et un ensemble de modules dont il dépend

```
angular.module('module-name', [dependencies]);
```

+ La liste des dépendances peut être vide, mais doit être présente pour la création
+ Cette injection de dépendances permet de lier les différents modules entre eux

+ La récupération d'un module existant peut se faire via la même fonction mais sans la liste de dépendances en paramètre

```
var myModule = angular.module('module-name');
```

MODULARISATION

CONTENU D'UN MODULE ANGULAR

+ Un module Angular va contenir un ensemble d'éléments de 2 types
  - **Configuration** : exécuté lors de la phase de configuration initiale (du service **$injector**) ; ne peut contenir que des **providers** et des **constantes** ; les providers correspondent aux **services.**
  - **Initialisation** : exécuté au chargement du module, après la phase de configuration ; ne peut contenir que des **instances**; permet d'initialiser l'application

```
angular.module('myModule', [dependencies])
  .config(function(injectables) { // provider-injector
    // You can have as many of these as you want.
    // You can only inject Providers (not instances) into the config blocks.
  })
  .run(function(injectables) { // instance-injector
    // You can have as many of these as you want.
    // You can only inject instances (not Providers) into the run blocks
  });
```

MODULARISATION

ELÉMENTS DE CONFIGURATION

+ Déclarés via des **fonctions prédéfinies** ou par la fonction générique « **config** »

```
angular.module('myModule', []).
  value('a', 123).
  factory('a', function() { return 123; }).
  directive('directiveName', ...).
  filter('filterName', ...);
```

```
angular.module('myModule', []).
  config(function($provide, $compileProvider, $filterProvider) {
    $provide.value('a', 123);
    $provide.factory('a', function() { return 123; });
    $compileProvider.directive('directiveName', ...);
    $filterProvider.register('filterName', ...);
  });
```

+ Appliqués dans l'ordre de déclaration sauf pour les constantes qui sont déplacées au début

MODULARISATION

CHARGEMENT ET GESTION DES DÉPENDANCES

+ Les dépendances d'un module sont chargées avant le module lui-même, et ce récursivement
+ Tous les éléments de configuration d'un module et de toutes ses dépendances sont chargées en premier
+ Puis toutes les fonctions Run() sont exécutées, en commençant par les dépendances
+ Un module n'est chargé qu'une seule fois, même s'il est déclaré comme dépendance au niveau d'un autre module
    - Sa méthode d'initialisation (Run()) ne sera exécutée qu'une seule fois au 1er chargement

MODULARISATION

RECOMMANDATIONS

+ Pour les applications limitées en fonctionnalités :
  - Le module applicatif porte l'ensemble des composants

```
angular.module('MyApp', [])
       .config(...)
       .controller(...)
       .factory(...)
       .directive(...);
```

  - Ou éventuellement un module par type de composant technique

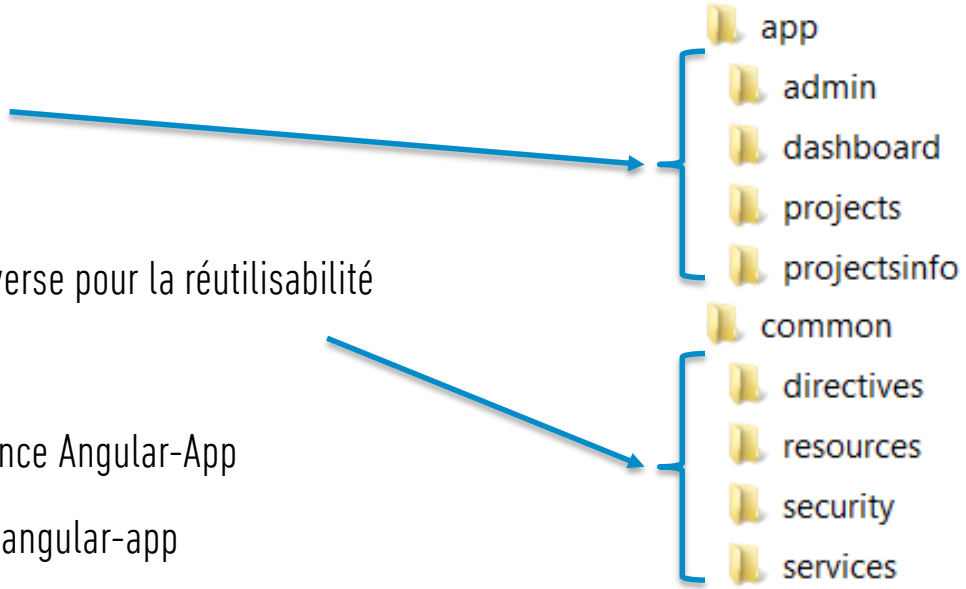
MODULARISATION

RECOMMANDATIONS

+ Pour les applications plus conséquentes, Google préconise un découpage fonctionnel

- 1 module par fonctionnalité
- Éventuellement 1 module transverse pour la réutilisabilité
- Exemple : l'application de référence Angular-App

  https://github.com/angular-app/angular-app



MODULARISATION

RECOMMANDATIONS

+ Pour les composants réutilisables (modules externes)
  - 1 module par composant réutilisable
    - directive
    - service
    - ensemble cohérent

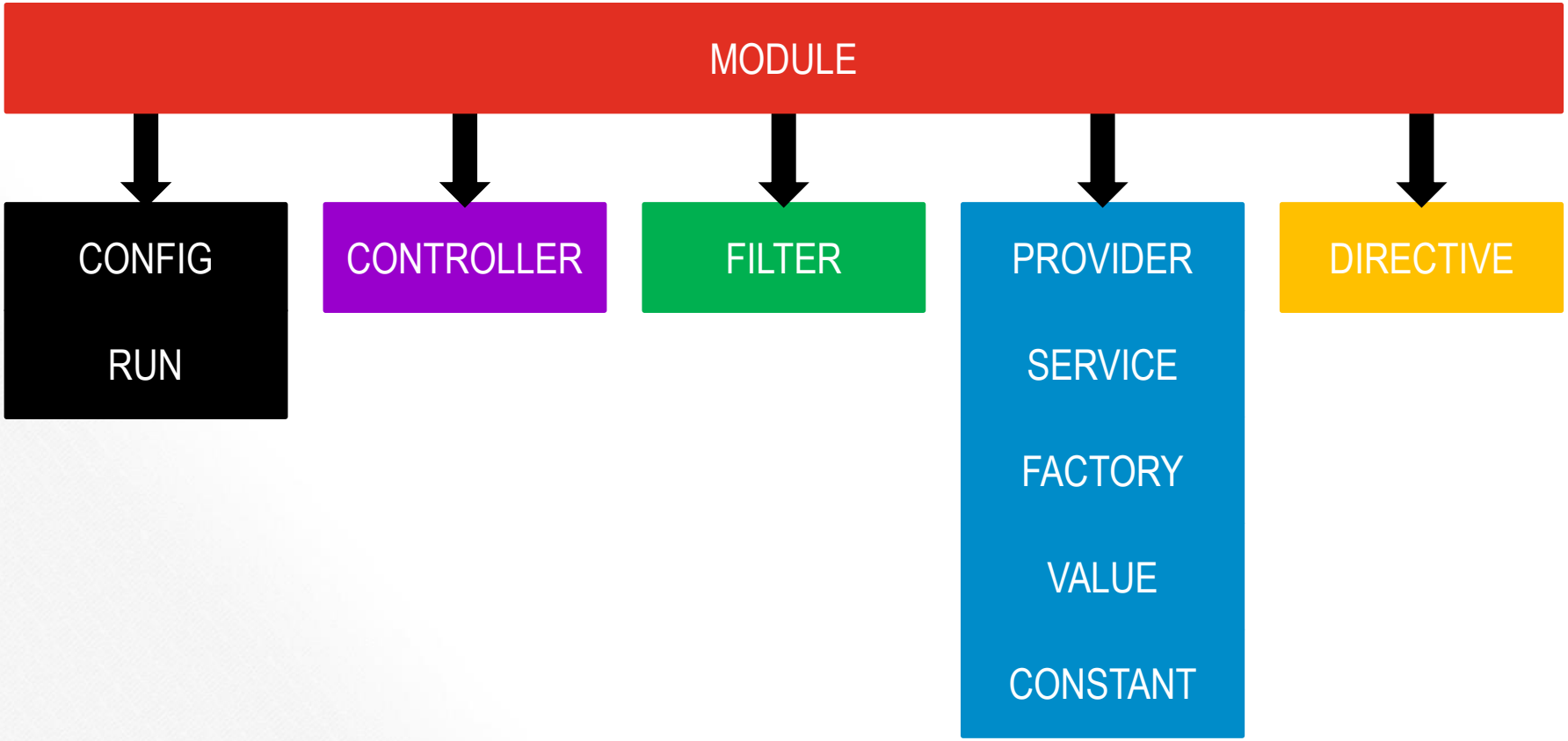
```
var app = angular.module('MyApp',
                  ['MyAwesomeComponent', ...];

angular.module('MyAwesomeComponent', [])
       .service(...)
       .directive(...)
       .directive(...);
```

  - Pour exemple, voir le code source de angular-bootstrap

MODULARISATION

SYNTHÈSE : LES MODULES SONT DES CONTENEURS



SQLI GROUP

EXERCICES PRATIQUES

TP8. MODULARISATION

MODULARISER

- ✓ 1 fichier = 1 responsabilité

SQLI GROUP

ROUTAGE

ROUTES

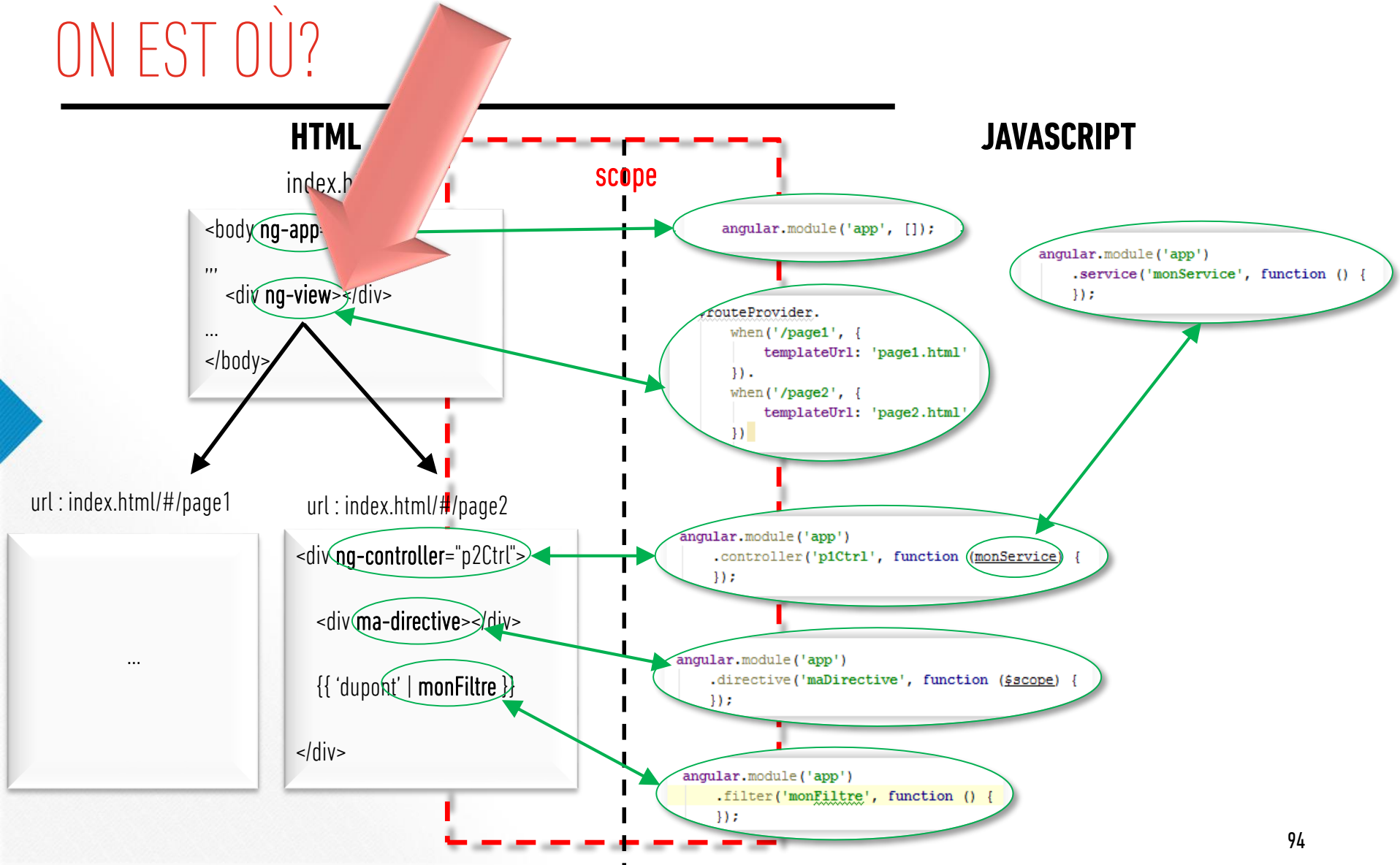
PRINCIPES

+ A une route correspond une vue affichée à l'utilisateur

+ Les routes permettent d'aller à une vue précise (elles peuvent indiquer un contexte d'exécution)

+ Les routes sont URL-Friendly

ON EST OÙ?


HTML
JAVASCRIPT
scope
index.h
<body ng-app=
,,,
<div ng-view>
...
</body>
angular.module('app', []);
angular.module('app')
.service('monService', function () {
});
routeProvider.
when('/page1', {
templateUrl: 'page1.html'
}).
when('/page2', {
templateUrl: 'page2.html'
})
url : index.html/#/page1
...
url : index.html/#/page2

<div ma-directive>
{{ 'dupont' | monFiltre }}

angular.module('app')
.controller('p1Ctrl', function (monService) {
});
angular.module('app')
.directive('maDirective', function ($scope) {
});
angular.module('app')
.filter('monFiltre', function () {
});

ROUTES

DÉCLARATION

+ Les routes de l'application sont gérées par le provider **$routeProvider** du module **ngRoute**, défini dans **angular-route.js**

+ Méthodes de **$routeProvider**
  - **`when(path, route)`** : lorsque **path** est trouvé dans l'URL
  - **`otherwise(params)`** : lorsqu'aucun **path** n'est trouvé via **when**

+ Concerne un module applicatif, la configuration se fait via la méthode **config()** (c'est un provider)

+ Une bonne pratique est de placer cette configuration dans un fichier portant le module applicatif ou alors dans un fichier dédié pour cette configuration.

ROUTES

DÉCLARATION : WHEN(PATH, ROUTE)

+ **path** : fragment d'URL présent dans **$location.path**

  - Peut contenir des variables préfixées avec "**:**"

```
when('/phones/:phoneId', …
```

  - Variables accessibles dans l'objet **$routeParams**

```
myMovieControllers.controller('MovieCtrl', ['$scope',
'$routeParams', function($scope, $routeParams) {
    $scope.doSomething = function() {
        var id = $routeParams.phoneId
        // do something with id
    }
}]);
```

ROUTES

DÉCLARATION : WHEN(PATH, ROUTE)

+ **path** (suite)
  - peut contenir des « **wildcards** »

« **?** » permet de rendre optionnel le paramètre qui le précède

```
phones/:phoneId?
```

« ***** » permet de passer en paramètre une chaîne qui contiendra des « **/** »

```
phones/:phoneName*/save
```

Exemple de path qui match : **phones**/name/mon/tel/**save**
On disposera dans **$routeParams** d'un attribut **phoneName** disposant de la valeur "**name/mon/tel**"

ROUTES

DÉCLARATION : WHEN(PATH, ROUTE)

+ **route** : données associées à la route (**$route.current**) lorsque le **path** est trouvé

  - **controller** : nom du contrôleur à associer à la vue

  - **template | templateUrl** : template HTML ou chemin vers le template à utiliser par la vue
    - » Le paramètre template est prioritaire sur templateUrl

  - **resolve** : liste de dépendances à injecter au contrôleur, sous forme de paires clé/valeur
    - » **Clé** : nom de la dépendance qui sera injecté au contrôleur
    - » **Valeur** : nom d'un **service** sous forme de chaine de caractères, ou nom d'une **fonction**, ou fonction anonyme ; si la valeur est une **promesse** (fonction asynchrone), le résultat de la promesse est attendu avant de traiter le routage

```
{ key1: 'service1', key2: function2, key3: function() { ...}, ... }
```

ROUTES

DÉCLARATION : WHEN(PATH, ROUTE)

+ **route** (suite)

  - **redirectTo** : redirige vers une autre route, en mettant à jour **$location.path**
  - **reloadOnSearch** : indique s'il faut recharger la route lorsque **$location.search** ou **$location.hash** est modifié ; **true** par défaut
  - **caseInsensitiveMatch** : indique s'il ne faut pas tenir compte de la casse dans le chemin recherché ; **false** par défaut

ROUTES

DÉCLARATION : OTHERWISE(PARAMS)

+ **params** : données associées à la route (**$route.current**) lorsqu'aucun **path** défini précédemment n'est trouvé.

  → cf. paramètre **route** de la méthode **when**

ROUTES

DÉCLARATION DES ROUTES APPLICATIVES

```
var myApp = angular.module('myApp', ['ngRoute']);
myApp.config(['$routeProvider', function($routeProvider) {
    $routeProvider.
                when('/phones', {
                     templateUrl: 'partials/phone-list.html',
                     controller: 'PhoneListCtrl',
                        resolve: {key1:'Service1', key2:function2}
                }).
                when('/phones/:phoneId', {
                     templateUrl: 'partials/phone-detail.html',
                     controller: 'PhoneDetailCtrl'
                }).
                otherwise({
                     redirectTo: '/phones'
                });
}]);
myApp.controller('PhoneListCtrl', [‘$scope’, ‘Service1’, function ($scope,
Service1) { … }]);
```

ROUTES

SERVICE $ROUTE

- Service utilisé par Angular pour la mise en œuvre du routage défini via le **$routeProvider**

- Propriétés
  - **current** : définition de la route courante ; permet notamment d'accéder au **$scope**
  - **routes** : liste de toutes les définitions de routes

- Fonctions
  - **reload()** : recharge la route courante

ROUTES

SERVICE $ROUTE

- Evènements
  - **$routeChangeStart** : juste avant un changement de route, résolution des dépendances
  - **$routeChangeSuccess** : après la résolution des dépendances ; déclenche l'instanciation du contrôleur et le rendu de la vue
  - **$routeChangeError** : erreur lors de la résolution des dépendances
  - **$routeUpdate** : si propriété reloadOnSearch positionnée à false, déclenche cet événement lors que les paramètres search changent.

Courants dans un app.run()

```
app.controller("AppCtrl", function ($rootScope) {
  $rootScope.$on("$routeChangeStart",
                 function (event, current, previous, rejection) {
    console.log($scope, $rootScope, $route, $location);
  });
  $rootScope.$on("$routeChangeSuccess",
                 function (event, current, previous, rejection) {
    console.log($scope, $rootScope, $route, $location);
  });
});
```

ROUTES

LAYOUT TEMPLATE

+ Un point d'entrée commun pour toutes les vues : **Layout Template**

+ Un ensemble de **Partial Templates** qui seront inclus dans le Layout Template, dépendamment de la route

+ La directive **ng-view** permet d'inclure la vue référencée par la route dans le layout principal (index.html)

+ A chaque changement de route, la vue incluse change

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

PARTIAL TEMPLATES

PARTIAL TEMPLATES

+ Les templates qui prendront place dans le layout template sont des portions de vues appelées "**Vues partielles**"

+ Les vues ont accès au scope du contrôleur associé dans la configuration du **$routeProvider**

$LOCATION

ACCÈS A LA BARRE D'ADRESSE

+ Le service **$location** parse l'URL de la barre d'adresse et la rend accessible depuis l'application

+ Modifications **bidirectionnelles**

```
$location.path()    // récupère le path actuellement visible
$location.path('/products')  // affecte le path
```

+ Permet un accès fin aux données de l'URL : port, server, path, etc... (**$location.port()**...)

+ Abonnements aux évènements de changements d'URL
  - **$locationChangeStart**
  - **$locationChangeSuccess**

$LOCATION

ACCÈS A LA BARRE D'ADRESSE

```
moviesControllers.controller('MoviesCtrl', ['$scope', '$location'],
  function($scope, $location) {
    $scope.changeLocation = function() {
        var loc = $location.path();
        . . .
        $location.path('/movies/movie-1')
    }
  }]);
```

MODES DE ROUTAGE

HTML5 VS HASHBANG

+ Configuration du routage AngularJS pour assurer la **navigation sans rechargement de page**
+ Pour assurer un changement d'URL sans rechargement de page : **History API d'HTML5**
+ Support navigateur :

Session history management - Candidate Recommendation

Method of manipulating the user's browser's session history in JavaScript using history.pushState, history.replaceState and the popstate event

*Usage stats:	Global
Support:	73.36%
Partial support:	3.68%
Total:	77.04%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	IE Mobile
								2.1		
								2.2		
						3.2		2.3		
						4.0-4.1		3.0		
	8.0					4.2-4.3		4.0		
	9.0	26.0	31.0			5.0-5.1		4.1		
	10.0	27.0	32.0			6.0-6.1		4.2-4.3	7.0	
Current	11.0	28.0	33.0	7.0	19.0	7.0	5.0-7.0	4.4	10.0	10.0
Near future		29.0	34.0		20.0					
Farther future		30.0	35.0		21.0					
3 versions ahead		31.0	36.0							

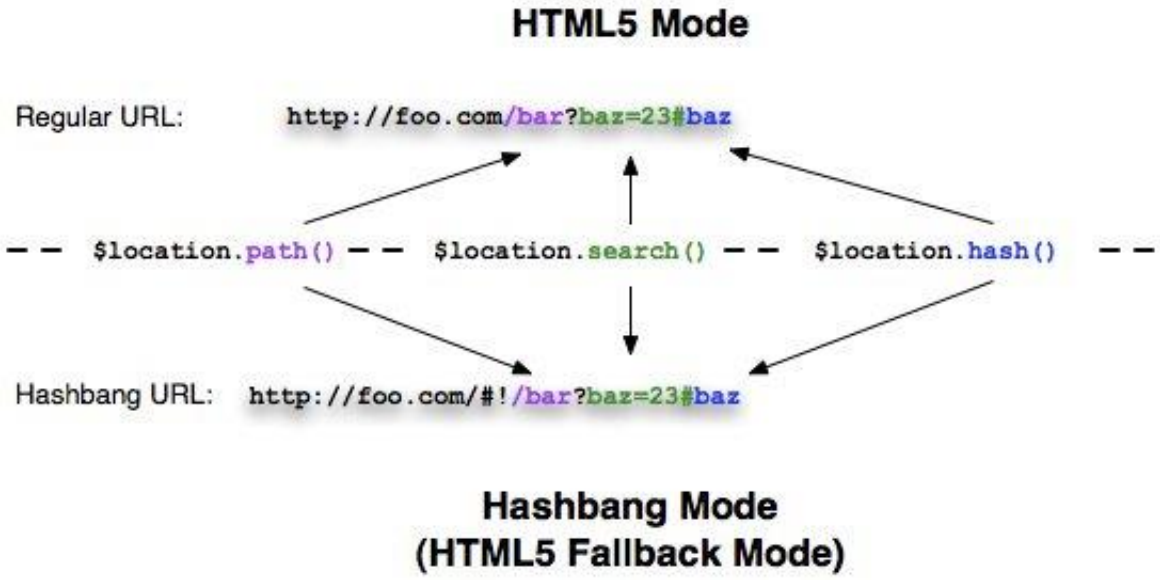
+ Pour les navigateurs "historiques": seul l'utilisation de **window.location.hash()** permet un changement d'URL sans modification
+ Angular créé un hash "dynamique" pour simuler une navigation historisée

MODES DE ROUTAGE

HTML5 VS HASHBANG : 2 MODES

+ Mode **HTML5** : s'appuie sur l'API History Html5

+ Mode **HashBang** : s'appuie sur une modification du hashbang d'URL



MODES DE ROUTAGE

CONFIGURATION DU MODE DE ROUTAGE

+ Utilisation de l'API disponible sur **$location**

  - **html5Mode(mode)** : true = mode HTML5 / false = mode Hashbang
  - **hashPrefix(prefix)** : préfixe utilisé pour la partie hash (path & search)

SQLI GROUP

EXERCICES PRATIQUES

TP9. ROUTES

AJOUTER LES ROUTES AUX VUES

- ✓ 3 vues = 3 routes
- ✓ $routeProvider
- ✓ ng-view
- ✓ $routeParams (Update)

SQLI GROUP

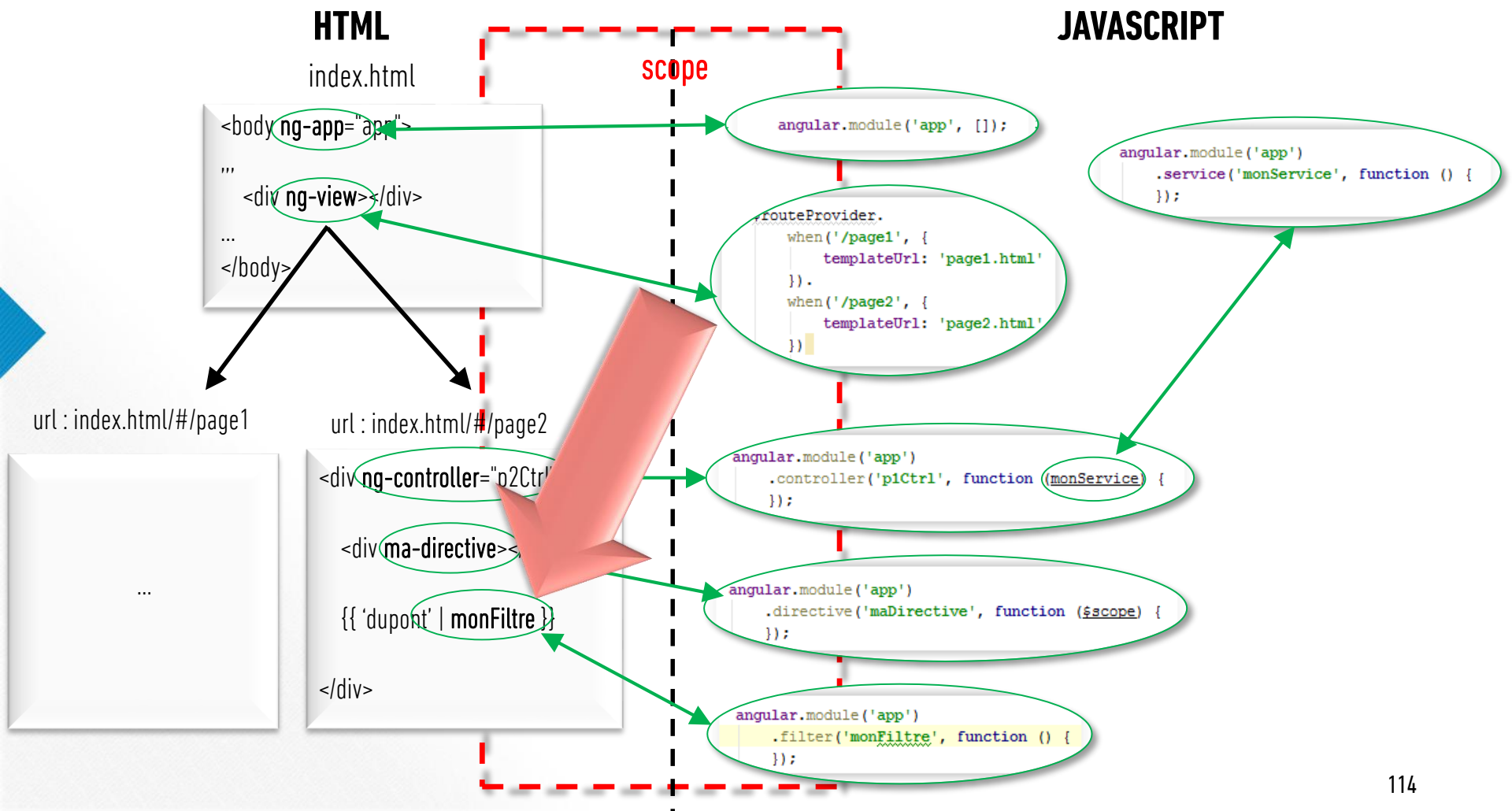
FILTRES

FILTRES

PRINCIPES

- Les filtres ont pour objectif de transformer une données en une autre.
- En entrée, on a une donnée... en sortie, une autre (obtenue à partir de la 1ère).
- Ils permettent notamment de modifier l'affichage d'une données dans la vue.

ON EST OÙ?


HTML
index.html
<body ng-app="app">
...
<div ng-view>
...
</body>
url : index.html/#/page1
...
url : index.html/#/page2
<div ng-controller="p2Ctrl"
<div ma-directive></
{{ 'dupont' | monFiltre }}

scope
JAVASCRIPT
angular.module('app', []);
routeProvider.
when('/page1', {
templateUrl: 'page1.html'
}).
when('/page2', {
templateUrl: 'page2.html'
})
angular.module('app')
.service('monService', function () {
});
angular.module('app')
.controller('p1Ctrl', function (monService) {
});
angular.module('app')
.directive('maDirective', function ($scope) {
});
angular.module('app')
.filter('monFiltre', function () {
});

FILTRES

FILTRES DISPONIBLES

- Filtres natifs disponibles dans le **$filterProvider**
  - **currency, date, filter, json, limitTo, lowercase, number, uppercase, orderBy**
- Utilisation dans les **templates** via les **expressions** Angular
- Chainage possible des filtres avec « | »

```
{{ myUser | json }}
{{ 'toto' | uppercase}}
{{ {foo:23, bar:"toto"} | json | uppercase }}
```

FILTRES

USAGE DANS LES CONTRÔLEURS ET LES SERVICES

+ Via **injection de dépendance : <nom_du_filtre>Filter**

```
angular.module('FilterInControllerModule', []).
controller('FilterController', ['uppercaseFilter',
    function (uppercaseFilter) {
      var uc = uppercaseFilter('machaine');
}]);
```

FILTRES

FILTRES SUR LES TABLEAUX

- **orderBy** : tri
- **filter** : filtre selon les valeurs des propriétés de l'objet traité
  - **Filtre par l'exemple :** à partir d'un objet disposant des mêmes noms de propriétés que les objets traités
- **limitTo** : limitation du nombre d'éléments à récupérer

- Utilisés dans les templates pour un affichage dynamique sur les directives **ng-repeat**

```
<div>
    <label>Nom</label>
    <input type="text" ng-model="query.name">
</div>
[...]
<ul>
    <li ng-repeat="phone in phones | filter:query | orderBy:order | limitTo:limit">
       {{phone.description}}
    </li>
</ul>
```

FILTRES

CRÉATION DE FILTRES

+ Fonction **filter()** disponible sur l'API du module pour enregistrer de nouveaux filtres

+ Associe un nom de filtre à une **fonction** qui prend en entrée un objet et retourne un objet

```
angular.module('myFilters', []).filter('checkmark', function()
{
    return function(input) {
        return input ? '\u2713' : '\u2718';
    };
});


{{ user.available | checkmark }}
```

EXERCICES PRATIQUES

TP10. FILTRES

FILTRE SUR LES LIVRES ET SUR LE NOM DE L'AUTEUR

- ✓ Livres : recherche & tri (filter, orderBy)
- ✓ Auteur : 1ère lettre en majuscule (app.filter)

SQLI GROUP

INTERNATIONALISATION

INTERNATIONALISATION

PRINCIPES

+ Angular supporte l'internationalisation et la localisation pour les filtres **date, number, currency**

+ Ne gère pas la notion d'externalisation des messages
  → Se référer à des librairies externes (ex : angular-translate)

+ Service **$locale**
  - fournit les informations sur la locale courante
  - par défaut pour la langue **en-us**

LOCALISATION

LOCALISER SON APPLICATION

+ Scripts de localisation fournis par défaut avec les sources d'AngularJS (répertoire **i18n**)
  Cf. http://code.angularjs.org/X.Y.Z/i18n/

+ Pour localiser l'application dans une autre langue, spécifier le fichier de localisation en tant que script

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

SQLI GROUP

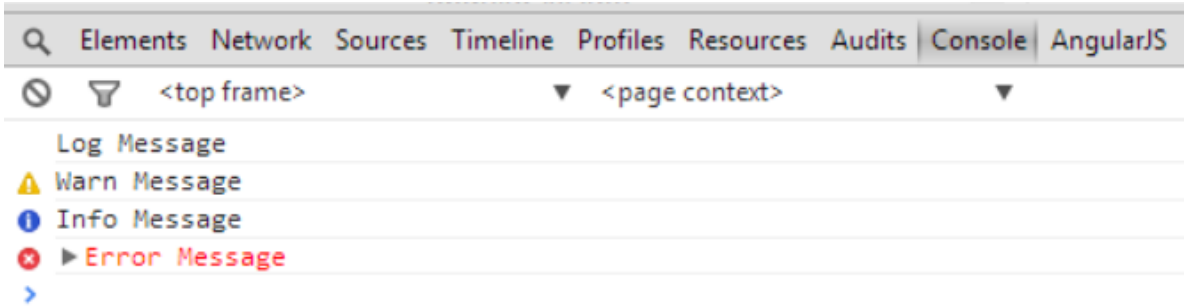
LOGGING

LOGGING

SERVICE $LOG

+ Service natif du module de base **ng**

+ Permet d'écrire des traces dans la console du navigateur, via des méthodes dédiées à différents niveaux d'importance
  - log(message)
  - debug(message)
  - info(message)
  - warn(message)
  - error(message)



+ Peut être injecté dans les controlleurs et mis à disposition de la vue via le scope

```
function LogCtrl($scope, $log) {
    $scope.$log = $log;
}
```

FIN