SOAT

Angular JS

3 jours

Objectifs

- Qu’est-ce qu’une **SPA**, pourquoi **AngularJS** ?
- D'angular **<=1.4** à **1.5**
- Manipuler les **vues**
- Une hiérarchie de **components**
- Utiliser et créer des **Services**, organiser son application en **modules**
- Du **double data binding** au **flux unidirectionnel** avec **Redux**
- Manipuler les **formulaires**
- Mettre en place un **Routeur**

Single Page Application et Angular

Objectifs du chapitre

- Des sites webs classiques aux SPA
- Pourquoi AngularJS

Sites web classiques

NAVIGATEUR

Requête

URL

Réponse

HTML

SERVEUR WEB

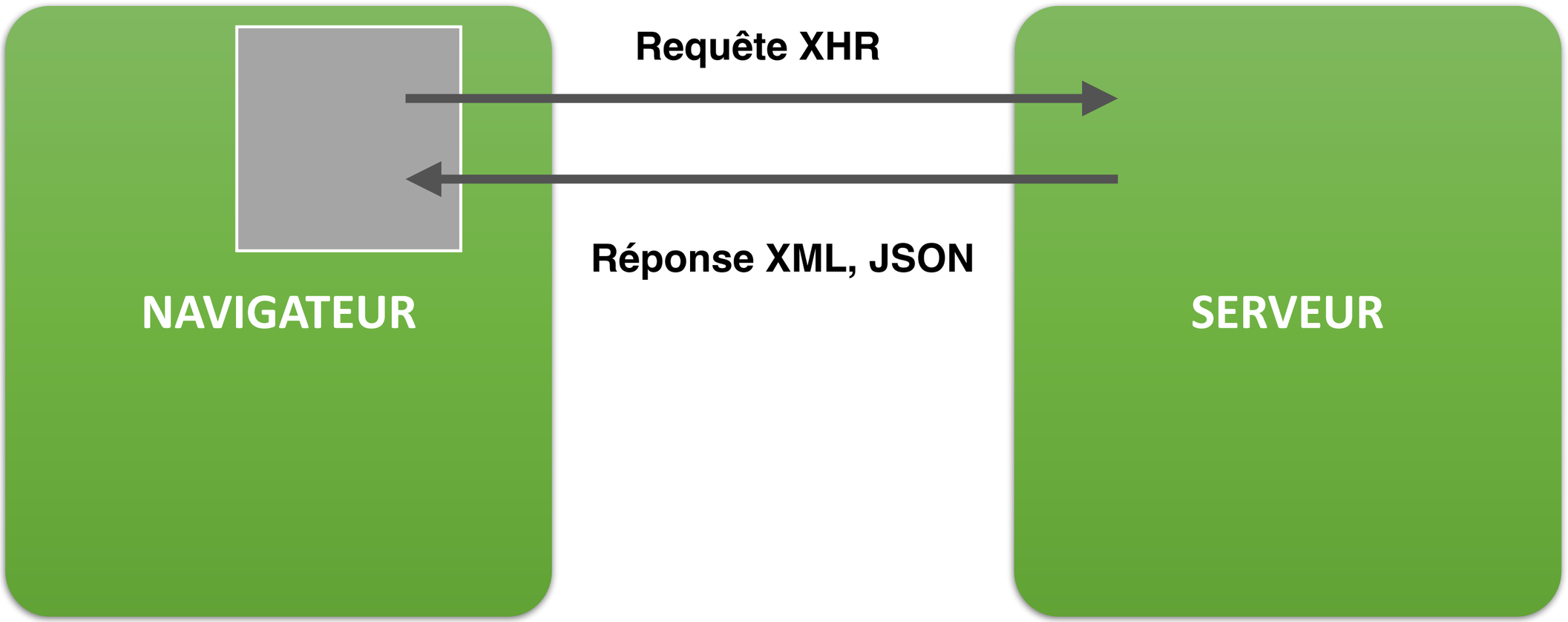
Sites web classiques

- La logique métier : Gérée par le serveur

- Gestion par page : Chaque requête **recharge** le navigateur

AJAX, Single Page Application

- AJAX = Asynchronous JavaScript and XML
- Basé sur xmlHttpRequest (xhr)



Single Page Applications

1. Eléments nécessaires chargés au **démarrage**
2. **Communication dynamique** : Le serveur ne renvoie que des éléments de la page
3. **Contexte** non perdu lors de la navigation

- Meilleure **expérience utilisateur**
- Indépendance Client / Serveur : **API REST**

D'Ajax aux frameworks MVC

- AJAX et **librairies** Javascript : jQuery, Prototype

  Pas d'éléments de structuration

- Frameworks MV* : Backbone, Ember, **AngularJS**

  Eléments de structure

Les caractéristiques d'AngularJS

- **Templating** : HTML
- **Approche déclarative :** extension du HTML à l'aide de directives
- **Components** : Lien entre la vue et les données
- **Services** : Logique métier
- **Injection de dépendance**
- **Redux : Flux unidirectionnel**
- **Router** : Naviguer dans son application
- **Modules** : Modulariser son application
- **Tests**

En conclusion, Angular c'est

- Un **framework**. Pas de dépendance avec d'autres librairies

- Une **architecture** modulaire facilement testable

- Un jeu de fonctionnalités qui **enrichit le HTML** et l’expérience utilisateur

Javascript et TypeScript

Objectifs du chapitre

- Rappels ES5
- ES6
- ES7
- TypeScript

ES5 - Historique

- Norme publiée le 3 décembre 2009
- Compatibilité tous navigateurs et IE >= v9

ES5 - Variables et fonctions

- Langage typé. Le type n'est pas déclaré
- Type implicite dépendant de la valeur affectée

```
toto = 1          // variable globale de type number
var tutu = 'a'    // variable globale de type string
var temp = {}     // variable globale de type objet
```

- "var" implique une déclaration locale

```
toto = 1          // variable globale à window
var tutu = 'a'    // variable globale à window

function maFct() {
   glob = 'ma globale'      // variable globale (pas de mot clé var)
   var temp = {}      // variable locale
}
```

ES5 - Fonctions et paramètres

- Les paramètres de type Objet passés à une fonction sont passés par référence
- Les paramètres de type litéraux sont passés par valeur

```
(function() {

	var monJson = {}
	var tutu = 'a'

	maFct(tutu, monJson)

	function maFct(parmT, parmO) {

		parmT = 'Titi'

		parmO.valeur = 1

	} // les fonctions sont "hoistées"

	console.log(tutu)		// "a"
	console.log(monJson.valeur)	// 1

})() // IIFE : module pattern
```

ES5 - Truthy Falsy

- Falsy : undefined, null, 0, false, NaN
- Truthy : autres valeurs

```
var a
if (!a) {
    console.log('a est falsy')
}

a = 'valeur'
console.log( a == true )  // true
console.log( a === true ) // false, === prend en compte le type de la variable

a = {}
console.log( a == true )  // true

a = []
console.log(a == true )   // true
console.log(a.length == true)    // false

a = null
console.log(a || 'valeur')// "valeur"
console.log(a ? 'vrai' : 'faux'  // "faux"
```

ES5 - Closures

- Fonction déclarée dans une fonction
- La fonction enfant a accès aux variables locales de son parent

```
function init() {

    var nom = "SOAT"                // nom est une variable locale créée par init

    function afficheNom() {         // afficheNom() est une fonction interne, une closure
        alert(nom)          // afficheNom() utilise une variable de la fonction parente
    }

    afficheNom()
}

init()
```

ES5 - Objet JSON

```
var personne = {
    prenom: 'Laurent',
    nom: 'Dupont',
    age: 25,
    couleurYeux: 'bleu'
}

console.log(personne.prenom)     // affiche "Laurent"

console.log(personne['age'])     // affiche 25

var methodes = {

                getYeux: function(obj) {
                    return obj.couleurYeux
                },

                getPrenom: function(obj) {
                    return obj.prenom
                }
};

console.log( methodes.getYeux(personne) )      // affiche "bleu"
console.log( methodes.getPrenom(personne) )    // affiche "Laurent"
```

ES5 - new Object

```
var MyClass = function(parm2) {
                this.parm1 = null
                this.parm2 = parm2
};

MyClass.prototype.myMethod = function(valeur) {
                this.parm1 = valeur
                console.log(this.parm1)
                console.log(this.parm2)
};

var myObj = new MyClass('première valeur')

myObj.myMethod('autre valeur')

console.log(myObj.parm2)
```

- Attention : "this" correspond au contexte de l'objet appelant

ES6

- Norme publiée en juin 2015. Appelé aussi ES2015.
- Support partiel des navigateurs, nécessite l'utilisation d'un **transpiler**

ES6 - Variables et constantes

Affectation par Let

```
{
    let letAffectation = "ES6";
}
letAffection === "ES6"; // ERREUR la variable est définie hors score
```

Constante

```
{
   const PI = 3.1415926; // immutable et pas de hoisting
}
```

- Attention : Avec une constante de type objet json, les propriétés sont mutables

ES6 - Templates de chaîne

```
var personne = { prenom: "Etienne", nom: "Martin" }
let message = `
    Bonjour ${personne. prenom} ${personne.nom},
    bienvenue dans notre boutique
`

/*
Les retours charriot sont conservés:
Bonjour Etienne Martin,
bienvenue dans notre boutique
*/
```

ES6 - Classes

Déclaration

```
class Personne {
    constructor(nom, age) {
        this.nom = age
        this.vieillir(age)
    }

   vieillir(annees) {
      this.annees = this.annees || 0 + annees
   }
}
let toto = new Personne('Le Héro', 14)
```

Héritage

```
class Employee extends Personne {
    constructor(nom, age, fonction) {
        super(nom, age);
        this.fonction = fonction;
    }
}
```

ES6 - Initialisations et destructurations (1)

Initialisation

```
function f (x, y = 5, z = 12) {
   return x + x + 12
}
f(23) === 42    // valeurs par defaut de y et z ont été utilisés
```

Paramètre REST

```
function f(guestStar, ...invites) {
   return guestStar + " et " + invites.length + "invités"
}
f("Alice", "Jean", "Boris") === "Alice et 2 invités"
```

Opérateur Spread

```
let nombre = "123456789";
let chiffres = [...nombre]; // Chiffres vaut
[1,2,3,4,5,6,7,8,9]
```

ES6 - Initialisations et destructurations (2)

```
var x = [1, 2, 3, 4, 5]     // Array
var [y, z] = x              // Affectation par décomposition
console.log(y)              // 1
console.log(z)              // 2
```

```
var o = {p: 42, q: true}
var {p, q} = o

console.log(p)              // 42
console.log(q)              // true

// Assigner de nouveaux noms
var {p: toto, q: truc} = o

console.log(toto)    // 42
console.log(truc)    // true
```

```
var toto = function(x) {
    return {x}
}

console.log( toto(12).x ) // retourne 12
```

ES6 - For … Of

```
var personnes = [
  {
    nom: "Alain Dupont",
    famille: {
      mère: "Isabelle Dupont",
      père: "Jean Dupont",
      sœur: "Laure Dupont"
    },
    âge: 35
  },
  {
    nom: "Luc Marchetoile",
    famille: {
      mère: "Patricia Marchetoile",
      père: "Antonin Marchetoile",
      frère: "Yann Marchetoile"
    },
    âge: 25
  }
]

for (var {nom: n, famille: { père: f } } of personnes) {
  console.log("Nom : " + n + ", Père : " + f)
}

// "Nom : Alain Dupont, Père : Jean Dupont"
// "Nom : Luc Marchetoile, Père : Antonin Marchetoile"
```

ES6 - Arrow functions (1)

```
let auCarre = valeur => valeur * valeur  // return implicite
auCarre(3) === 9
```

```
let incremente = valeur => {

    if (valeur <= 15)
      return valeur + 1

   }

    return valeur
}
incremente(12) === 13
```

ES6 - Arrow functions (2)

```
class MaClasse {
    constructor(x) {
      this.x = x
    }

    // La lambda garantit que this est l'instance de MaClasse

    callback = () => {
      return this.x
    }
}
```

ES6 - Promises

```
function msgAfterTimeout(msg, timeout) {
   return new Promise(resolve => {
      setTimeout(() => resolve(`-> ${msg}!`), timeout);
   })
 }
// Appel asynchrone
msgAfterTimeout("1er appel", 1000).then(() =>
   msgAfterTimeout("2eme appel", 500)).then(
      msg => console.log(`Après 1500ms ${msg}`)
)
// Affiche "Après 1500ms -> 2eme appel!"
```

ES6 - Modules

Fichier ./math.js = 1 module

```
export default function somme(x, y) { return x + y }
export function soustraction(x, y) { return x - y)
export const PI = 3.141593
```

Import global

```
import * as math from './math'
console.log('2π = ' + math.somme(math.PI, math.PI))
```

Import détaillé

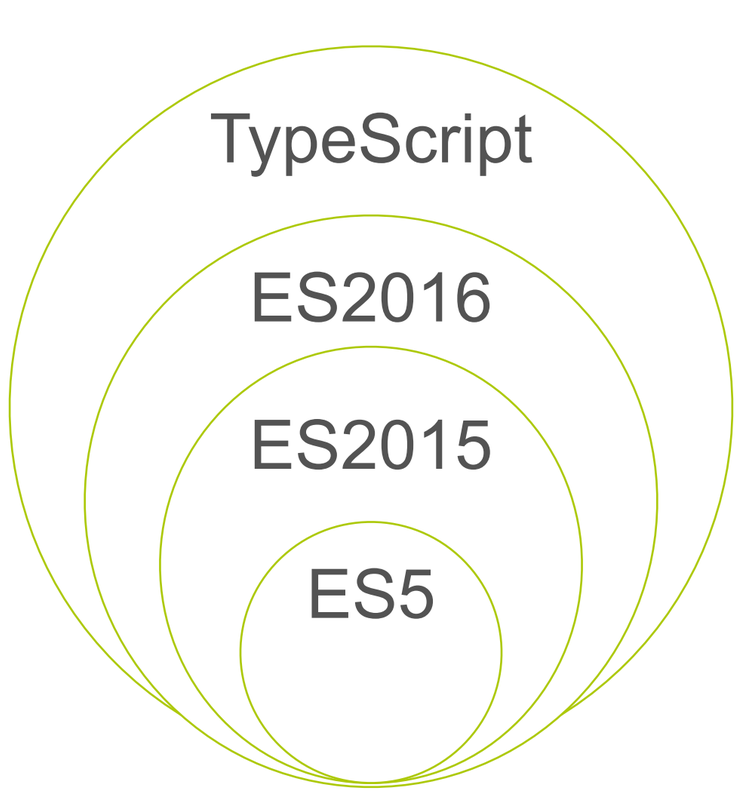
```
import somme, {soustraction} from './math'
console.log(somme(1, 2))
console.log(soustraction(2, 1))
```

ES7

- Norme en cours de conception.
- Appelée aussi ES2016.

TypeScript

- Open source, publié en octobre 2012, par **Microsoft**
- Superset de Javascript. Nécessite une "**transpilation**"
- Supporte toutes les versions de JS et apporte de **nouvelles** fonctionnalités
- **Angular 2** est développé en TypeScript : préconisé par Google



TypeScript - Décorateurs

- Annotation d'objet
- Ajout de metadata à ces objets

```
@Component({
     selector: 'app',
     providers: [NamesList],
     templateUrl: './app.html',
     directives: [RouterOutlet, RouterLink]
})
export class App {}
```

TypeScript - Portée (1)

```
export class App {
   nom             // variable publique déclarée implicitement
   public prenom    // variable publique
   private adresse    // variable privée
   static compteur = 0  // variable statique
   protected protegee // variable protected

   constructor(nom) {
      this.nom = nom   // construction obligatoire
      this.methode()
   }

   private methode = () => {
      // methode privée hoistée
   }

   public pubMethod() {
      // méthode publique
   }
}
```

TypeScript - Portée (2)

- **public** : variable ou méthode publique, accessible partout
- **private** : variable ou méthode privée, uniquement accessible dans l'instance de la classe
- **protected** : variable ou méthode protected, accessible dans la classe et dans les classes héritées (super())
- **static** : variable de classe indépendante des instances
- **abstract** : classe abstraite, non instanciable, pouvant être héritée

TypeScript - Types

```
// types primitifs : number, string, boolean, enum, void, null, undefined

const toto: number = 40
const tutu: string = 'literal'

enum state = { CONNECTING, CONNECTED }
if (state.CONNECTING === 'CONNECTING') {    // true
    ...
}

// any : type indéfini
const temp: any = {}
temp = 20

// Array types
let tableau: number[] // tableau de numériques

// fonctions
let fct = (parm:number) : boolean => {
    return !!number
}
```

TypeScript - Interfaces

```
interface TypePersonne {
    nom: string,
    getAddress(): string
}

class Person implements TypePersonne {
    nom: string
    getAddress = () : string => {
       return 'adresse'
    }
}


interface Humain {
    age: number,
    taille: number
}
interface Femme extends Humain {
    maquillage: boolean
}

let monHumain: Femme = {
    age: 20,
    taille: 180,
    maquillage: true
}
```

TypeScript - Typings

- Fichiers TSD : type definition. Fichiers de définition des types d'une lib js

En conclusion

- Google préconise l'utilisation de TypeScript pour Angular 2
- TypeScript ajoute de nouvelles fonctionnalités au javascript standard
- TypeScript suit et adapte les nouvelles normes JS

Installer un environnement

Objectifs du chapitre

- Utiliser NPM : Node Package Manager
- Configurer Webpack
- Installer les typings

Node et NPM

1. Installer Node.js

https://nodejs.org

2. Utiliser npm en ligne de commande

```
> npm init                        // Crée un fichier package.json
> npm install             // Installe le fichier package.json
> npm install library             // Installe la librairie dans le répertoire courant
> npm install library -g          // Installe library globalement (nécessite d'être admin)
> npm install lib  --save         // Installe lib dans les dependencies du package.json
> npm install lib --save-dev      // Installe lib dans les devDependencies du package.json
> npm start                       // run du script start
> npm test                        // run du script test
> npm run deploy          // run d'un autre script (mot clé run)
```

Fichier Package.json

```
{
  "name": "exemple",
  "version": "1.0.0",
  "description": "exemple",
  "main": "index.ts",
  "author": "ls",
  "license": "ISC",
  "dependencies": {
    "angular": "^1.5.7",
    "angular-ui-router": "^0.3.1",
    "mdi": "^1.5.54"
    ...
  },
  "devDependencies": {
    "css-loader": "^0.23.1",
    "es6-shim": "^0.35.0"
    ...
  },
  "scripts": {
    "start": "webpack-dev-server --progress --inline --content-base www/ --colors --port 9000 --watch",
    "deploy": "npm run cleandist && npm run webpack",
    "webpack": "webpack --config webpack.production.config.js",
    "cleandist": "rimraf dist/",
    "test": "karma start"
  }
}
```

Webpack - Configuration

Entry

- Import des **CSS**
- Import des **JS d'environnement** front (angular etc)

Output

- Génération d'un **bundle de dev**
- Ou génération d'un **bundle minifié de prod**

Sourcemaps

- Sources non compilées visibles sous **debug** dans les navigateurs

Webpack - Loaders

- Concaténation des CSS
- Compilation des fichiers LESS / SASS
- Compilation des fichiers Typescript
- Load des fichiers media (fontes, images…)
- postLoaders : fichiers spec et loader istanbul / reporting tests

Webpack - Plugins

HtmlWebpackPlugin

- Génération dynamique du fichier index.html

ProvidePlugin

- Déclaration de variables globales (ex jQuery)

DefinePlugin

- Utile pour définir des variables d’environnement (dev / prod)

Webpack - Scripts npm

- webpack-dev-server : Lance un server de test
- webpack : Commande de déploiement / build

Les fichiers typings

Installer typings

```
> npm install -g typings
```

Initialiser typings.json

```
> typings init
```

Installer des fichiers de définition

```
> typings install
```

typings.json

```
{
  "globalDependencies": {
    "angular": "registry:dt/angular#1.5.0+20160709055139",
    "angular-ui-router": "registry:dt/angular-ui-router#1.1.5+20160707113237",
    "es6-shim": "registry:dt/es6-shim#0.31.2+20160602141504",
    "jquery": "registry:dt/jquery#1.10.0+20160704162008",
    "redux": "registry:dt/redux#3.5.2+20160703092728",
    "redux-logger": "registry:dt/redux-logger#2.6.0+20160619033847",
    "redux-thunk": "registry:dt/redux-thunk#2.1.0+20160703120921"
  },
  "globalDevDependencies": {
    "angular-mocks": "registry:dt/angular-mocks#1.5.0+20160608104721",
    "jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
    "node": "registry:dt/node#6.0.0+20160709114037"
  }
}
```

En conclusion

- NPM nous permet de gérer l'installation des dépendances de l'application
- NPM nous permet de lancer des scripts d'exécution, test et déploiement
- Webpack est un module loader (compilateur typescript, less…)
- Webpack gère un serveur de développement
- Webpack déploie un bundle applicatif

Documentation webpack : https://webpack.github.io/docs/

Bootstrap Angular

Objectifs du chapitre

- Importer les fichiers principaux
- Définir un premier composant pour l'application
- Démarrer Angular

Import des librairies et styles

./index.ts

```
/**
 * Import CSS
 */
import './node_modules/mdi/css/materialdesignicons.min.css'
import './www/less/style.less'

/**
 *  Import Javascript libraries
 */
import './node_modules/es6-shim/es6-shim.js'
import './node_modules/angular/angular.js'
import './node_modules/angular-ui-router/release/angular-ui-router.js'
import './node_modules/angular-sanitize/angular-sanitize.js'
import './node_modules/redux/dist/redux.js'
import './node_modules/redux-thunk/dist/redux-thunk.js'
import './node_modules/redux-logger/dist/index.js'
import './node_modules/ng-redux/dist/ng-redux.js'

/**
 *  Main App
 */
import './www/js/app'
```

Premier composant angular

./www/js/hello-world.ts

```
export default class HelloWorldComponent {
  public template
  constructor() {
     this.template = `
       <div>Hello</div>
     `
}
```

./www/js/app.ts

```
import HelloWorldComponent from './hello-world'

angular.module('ToyStore', [])
  .component('helloWorld', new HelloWorldComponent())

angular.bootstrap(document.documentElement, ['ToyStore'])
```

Dans ./www/index-base.html

```
<body>
    <hello-world></hello-world>
</body>
```

Conclusion

- Les composants et fonctions angular sont importés dans l'app
- L'application est un module angular
- Bootstrap lance le module angular

- Note Angular <=1.4
- L'app était lancée avec la directive ng-app

```
<body ng-app="ToyStore">
    <hello-world></hello-world>
</body>
```

Premiers pas avec AngularJS

Objectifs du chapitre

- L'environnement AngularJS
- Evolution du framework

Un Framework complet

Double data binding / Flux unidirectionnel

Directives / Components

Routing

Controller / Components

SPA

Tests

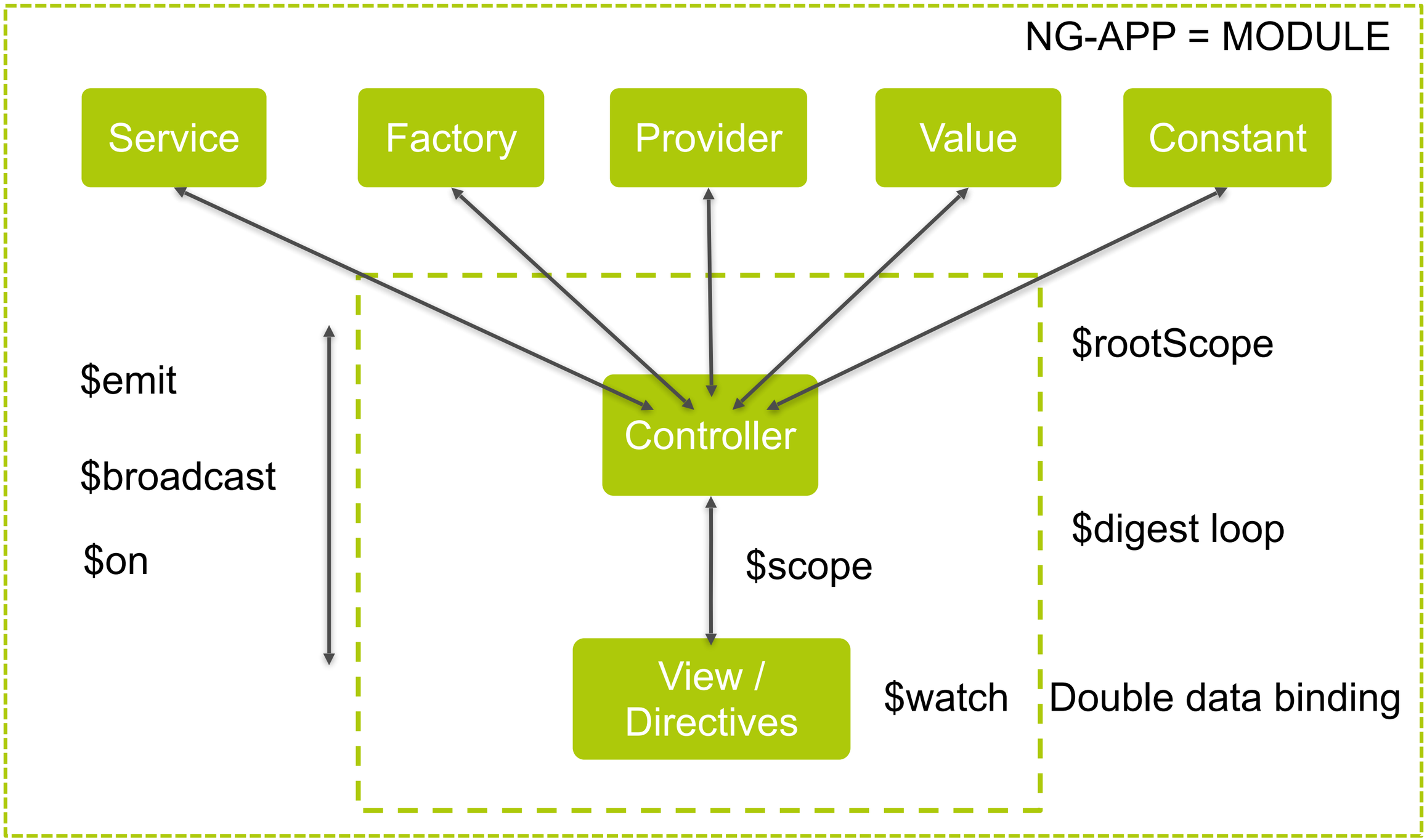
Templates

Services

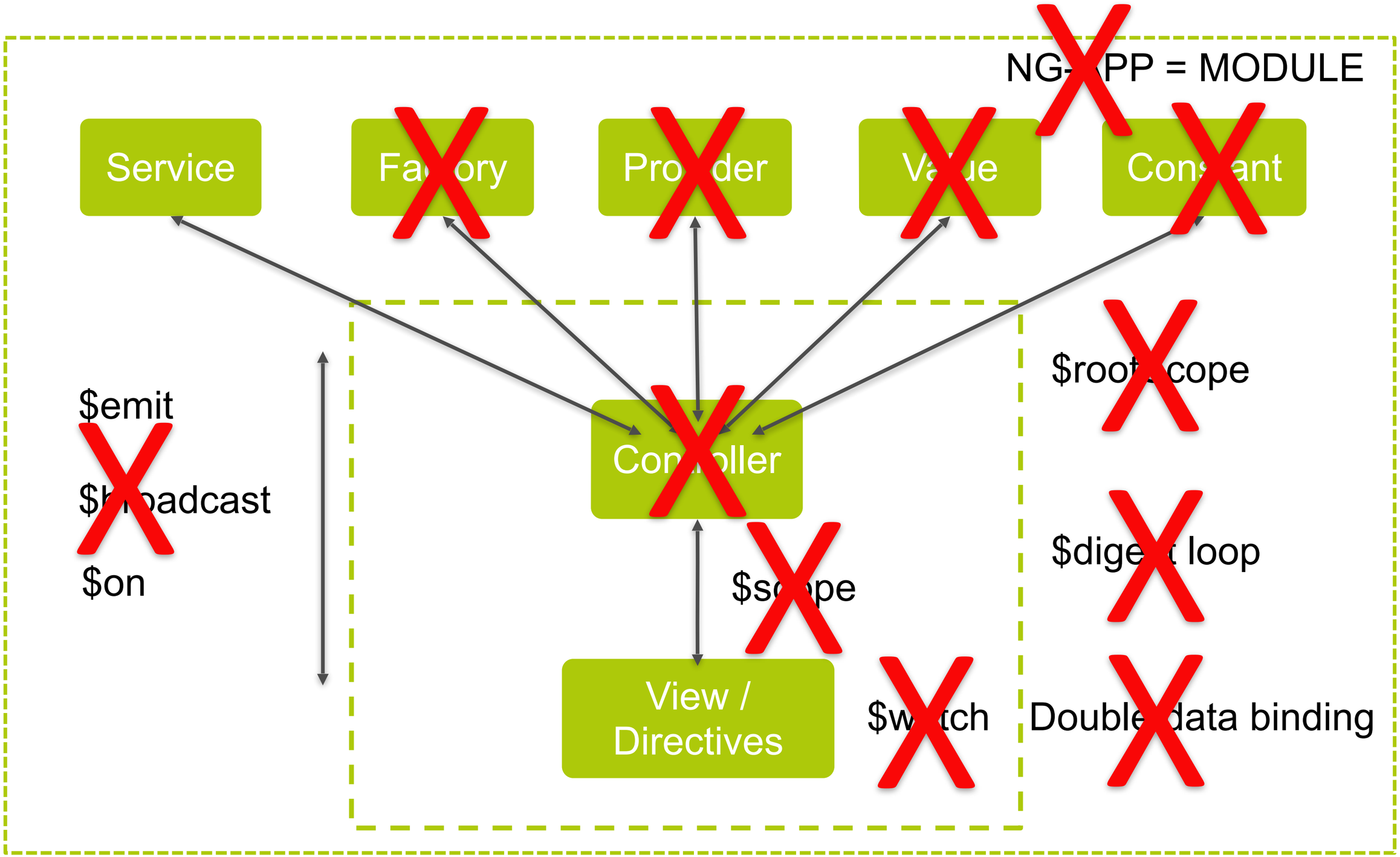
Injection de dépendance

HTTP / Ajax

Angular 1 - Concepts



Angular 1 - Evolution vers la v2



Angular 1.5

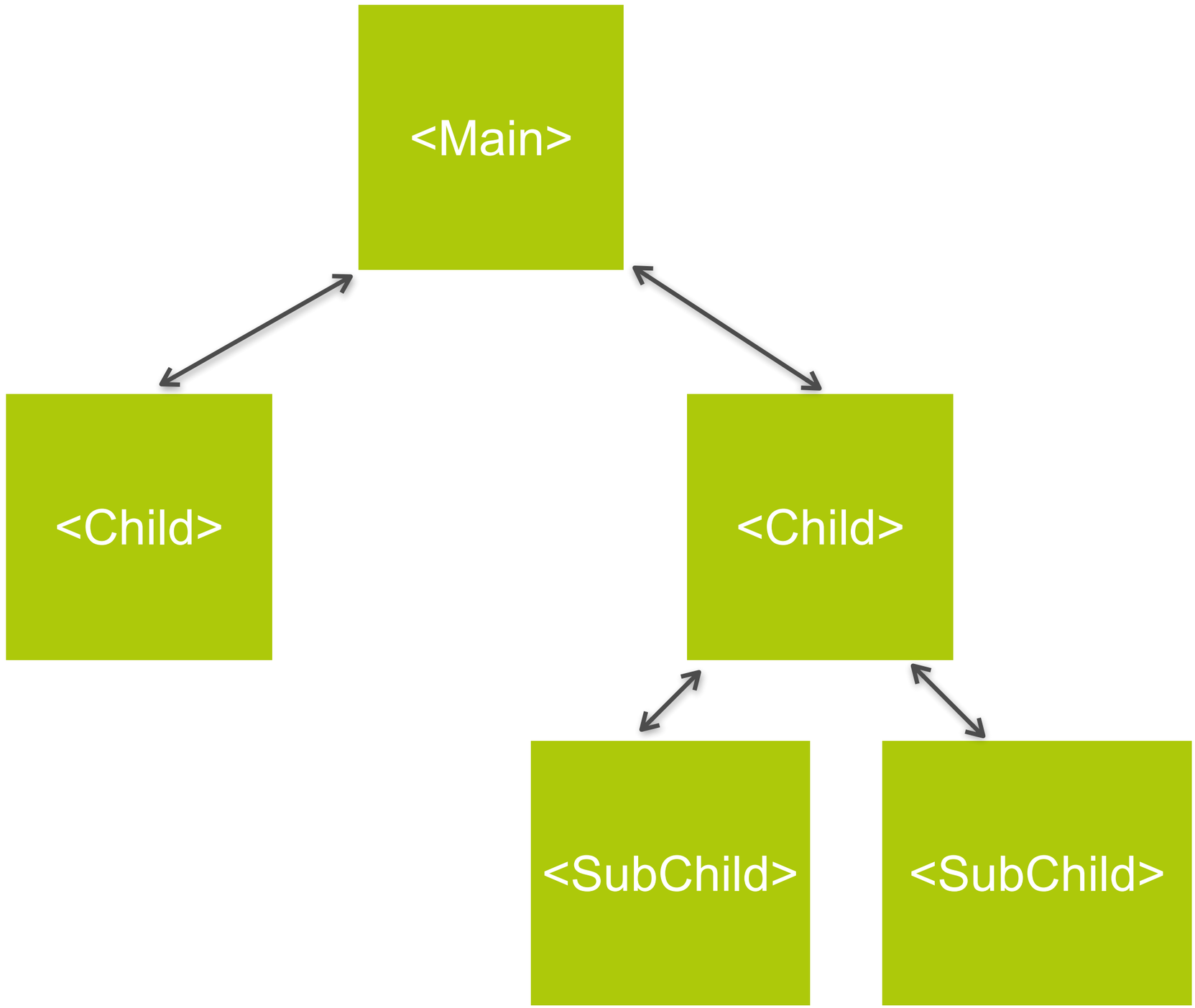
- Une hiérarchie de **components**
- Utiliser **Typescript**
- Gérer l'état de l'application à l'aide de **services**
- Adopter un **flux unidirectionnel** et supprimer le double data binding
- Utiliser les directives de type **attribut** uniquement

Les Components

Objectifs du chapitre

- Une hiérarchie de components
- Définir un component
- Manipulation des templates
- Directives angular

Hiérarchie de components



Component

```
export default class MonComponent {
  public controller             // association à un controller
  public templateUrl            // template du composant
  public bindings               // paramètres d'entrée sortir du composant

  constructor() {               // déclaration des paramètres
       this.controller = MonController
       this.templateUrl = './chemin/vers/template.html'
       this.bindings = {
          entree: '<',          // input par référence
          label: '@',           // input par valeur
          methode: '&'          // appel de fonction / évènement
       }
  }
}
```

```
import MonComponent from './mon-component'
angular.module('monModule', [])                 // déclaration du composant
  .component('monComponent', new MonComponent()
```

```
<mon-component
   entree="objet"
   label="litteral"
   methode="appelFonction()"
></mon-component>               // utilisation dans Html
```

Controller

```
export default class MonController {
  public variable              // variable exposée au template
  private data

  constructor() {}

  public methode = () => {
        ... do something
  }

  public "hook" () => {}       // cycle de vie du composant
}
```

$onChanges	Exécutée lors de changements d'input
$onInit	Exécutée à l'initialisation d'un composant, après le premier OnChanges
$onDestroy	Exécutée lors de la désinstantiation d'un composant
$postLink	exécuté après binding des data du template

Template

- Template = HTML
- Les variables et méthodes publiques sont utilisées avec le préfixe $ctrl
- Manipulation des templates = directives angular

```
<div>
        {{$ctrl.expression}}

        {{$ctrl.execMethode()}}

        <span ng-click="$ctrl.autreMethode()">click</span>
</div>
```

Directives Angular

Directive ng	Description
ng-click	Evènement click
ng-if / ng-switch	Affichage de la balise en fonction d'un booléen
ng-class	Affectation dynamique d'une classe CSS
ng-src	Affectation dynamique d'une url
ng-href	Affectation dynamique d'un lien
ng-keydown / keyup / keypress	Evènements clavier
ng-focus / blur	Evènements focus / blur
ng-show / hide	Affichage ou masquage en fonction d'un booléen
ng-repeat	Répète le contenu d'un array
ng-mouseover / mouseup / mousedown / mouseenter / mouseleave / mousemove	Evènements souris

ng-show / ng-hide : exemple

```
// JS
class MonController {

    public show
    public hidden

    public showBloc = () => {
       this.show = !this.show
    }

    public hideBloc = () => {
       this.hidden = !this.hidden
    }

}

// HTML
<div>
       <div ng-show="$ctrl.show">Vu</div>
       <div ng-hide="$ctrl.hidden">Pas vu</div>

       <button ng-click="$ctrl.showBloc()">Show</button>
       <button ng-click="$ctrl.hideBloc()">Hide</button>
</div>
```

ng-if + ng-repeat : exemple

```
// JS
class MonController {
    public datas = [1, 2, 3, 4, 5]
    public shown

    public showHideBloc = () => {
        this.shown = !this.shown
    }
}
</script>

// HTML
<div>
        <ul ng-if="$ctrl.shown">
                <li ng-repeat="data in $ctrl.datas">{{data}}</li>
        </ul>
        <button ng-click="$ctrl.showHideBloc()">Show</button>
</div>
```

➢ Différence avec ng-show / ng-hide : Les directives incluses dans le bloc ng-if ne sont pas exécutées si le bloc n'est pas affiché.

ng-switch : exemple

```
// JS
class MonController {
  public affiche
  public shown
}

<div>
        <input type="text" ng-model="$ctrl.affiche">

        <section ng-switch="$ctrl.affiche">
                <article ng-switch-when="1">Article 1</article>
                <article ng-switch-when="2">Article 2</article>
                <article ng-switch-when="3">Article 3</article>
                <article ng-switch-when="4">Article 4</article>
                <article ng-switch-when="5">Article 5</article>
                <article ng-switch-default>Article par défaut</article>
        </section>
</div>
```

ng-class : exemple

```
<style>
        .bleu   { color: blue; }
        .rouge  { color: red; }
        .vert   { color: green; }
</style>

<p ng-class="{rouge: !$ctrl.valide, vert: $ctrl.valide}">Mon texte</p>
<button ng-click="$ctrl.valide = !$ctrl.valide">click</button>

<p ng-class="$ctrl.couleur">Mon autre texte</p>
<input type="text" placeholder="bleu ou rouge" ng-model="$ctrl.couleur">
```

ng-repeat : les options (1)

Variable	Type	Description
$index	number	Itérateur de l'élement répété (de 0 à length-1)
$first	boolean	true si premier élément de la liste
$middle	boolean	true si ni premier ni dernier élément
$last	boolean	true si dernier élément de la liste
$even	boolean	true si valeur paire
$odd	boolean	true si valeur impaire

ng-repeat : les options (2)

```
<style>
        .bleu   { color: blue; }
        .rouge  { color: red; }
        .vert   { color: green; }
</style>

// JS
class MonController
  public datas = ['un', 'deux', 'trois', 'quatre', 'cinq']
}


<ul>
        <li ng-repeat="data in $ctrl.datas"
            ng-class="{'rouge': $first, 'bleu': $last, 'vert': $middle}">
                {{data}}
        </li>
</ul>

<ul>
        <li ng-repeat="data in $ctrl.datas"
            ng-class="{'bleu': $even, 'vert': $odd}">
                {{data}}
        </li>
</ul>
```

Angular <=1.4 - Le controller (1)

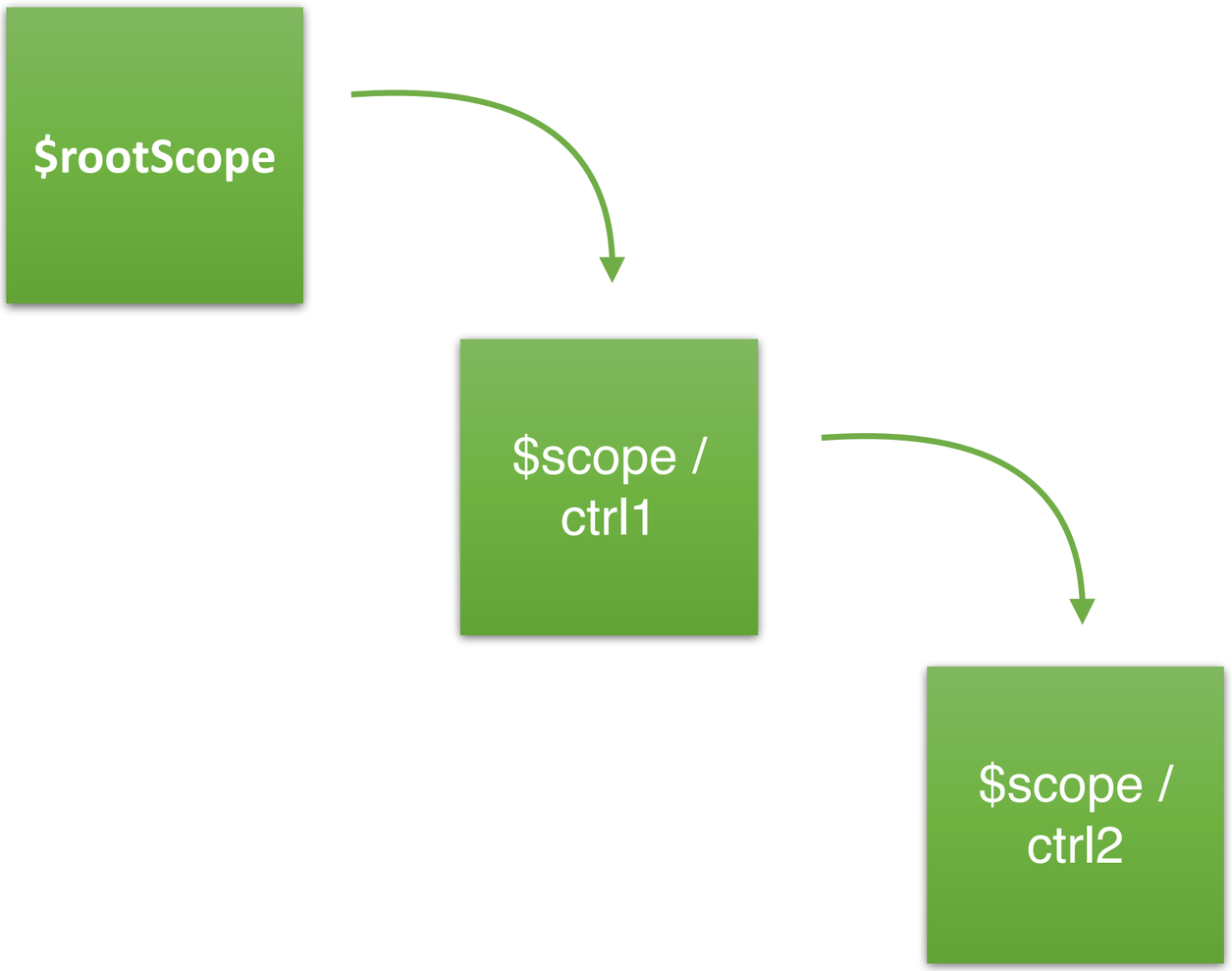
- Le controller permet la manipulation des données d'une application AngularJS
- Les controllers sont des objets Javascript
- Le $scope est le lien entre le controller et la view

Angular <=1.4 - Le controller (2)

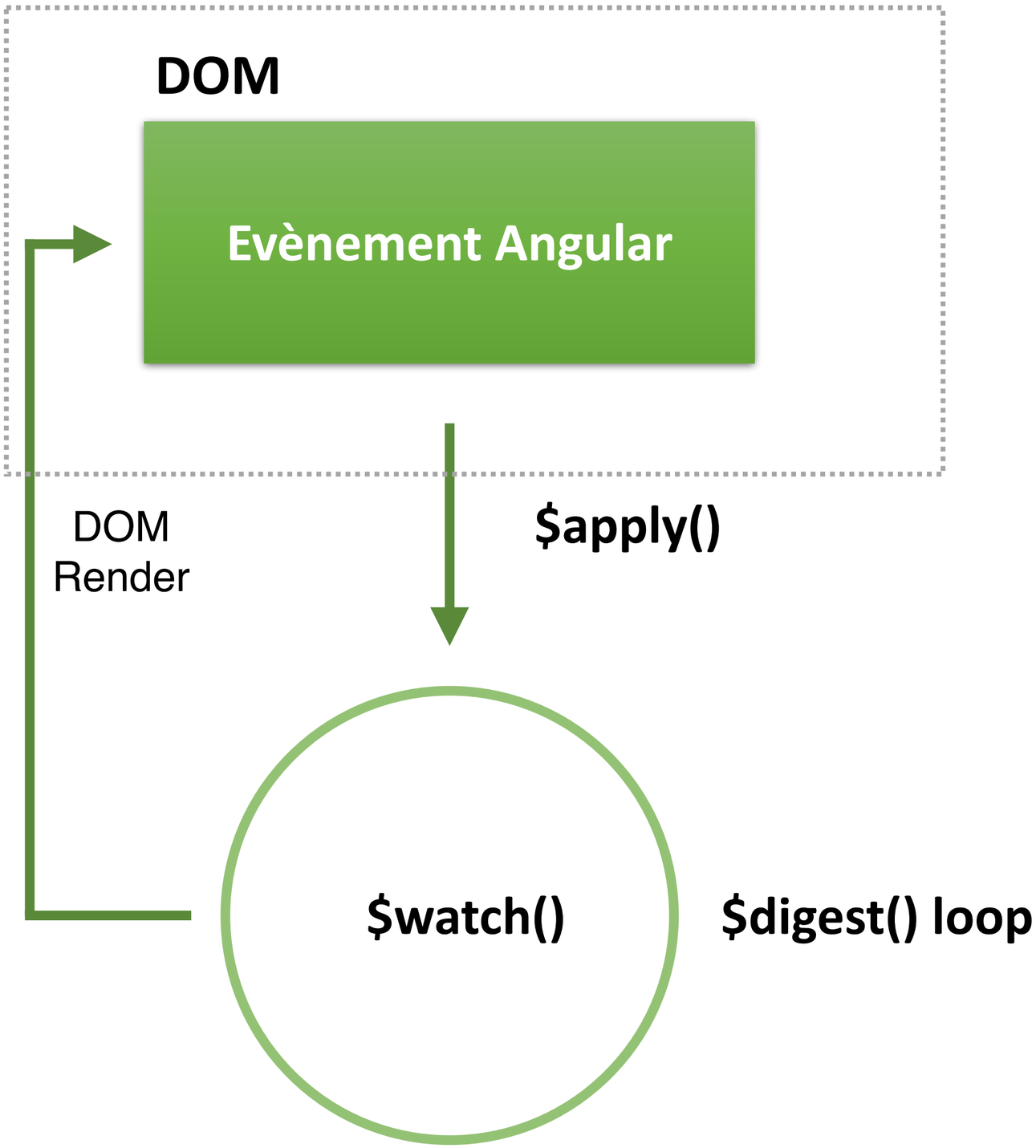
```
// JS
angular.module('App')
  .controller(($scope) => {

     $scope.datas = ['un', 'deux', 'trois', 'quatre', 'cinq']

  })

// HTML
<div ng-controller="Ctrl">

  {{binding}}

</div>
```

Hiérarchie de scopes et $rootScope

- Des controllers peuvent contenir d'autres controllers
- Le scope enfant d'un controller hérite du scope de son controller parent
- Il est possible d'injecter le rootScope à un controller : le rootScope est le scope parent de tous les $scope de l'application



Dirty-Checking, comment ça marche ?



- Pour chaque élément du $scope, Angular crée automatiquement un $watch

- A chaque évènement Angular, le $digest va contrôler les $watch et appliquer les modifications à la vue

Les méthode de $scope

$watch, $watchCollection	Surveille le changement de $scope d'angular suite à un évènement
$digest, $apply	Provoque l'éxécution de la $digest loop
$on, $emit, $broadcast	Gestion des évènements de $scope

Conclusion et bonnes pratiques

- Angular évolue vers une hiérarchie de composants
- Ne pas utiliser ng-controller
- Ne pas utiliser $scope, $rootScope et les méthodes associées

Les services

Objectifs du chapitre

- Créer un service
- Injecter un service
- Organiser son application en modules

Que sont les services ?

- Les services permettent d'organiser, structurer et partager le code de l'application
- Les services sont des objets liés entre eux par injection de dépendance
- Ils sont instanciés au lancement de l'application
- Ce sont des singleton
- AngularJS fournit en fournit un certain nombre, ainsi que la possibilité de créer ses propres services

L'injection de dépendance

➢ Permet de découpler la dépendance entre objets d'une application

- Code modulaire réutilisable
- Objets plus faciles à lire
- Objets plus faciles à tester

```
class MonController {

        constructor(
            private MonService1,
            private MonService2
        ) {}
        $onInit = () => {
                this.MonService1.initialize()
        }
}

MonController.$inject = ['MonService1', 'MonService2']
```

Les services $timeout et $interval

➢ Wrappers autours des timers javascripts dans Angular

```
class MonController {
    private temps = 1000 // millisecondes

    constructor(
        private $timeout,
        private $interval
    ) {}

    $onInit() => {
        const monTimer = this.$timeout(() => {
                ...
                this.$timeout.cancel(monTimer)
        }, temps)

        const monInterval = this.$interval(() => {
                ...
                this.$interval.cancel(monInterval);
        }, temps)
    }

}
```

➢ A utiliser plutôt qu'en JS natif, car ils lancent le $digest()

Le service $http (1)

- $http permet d'effectuer des requêtes Ajax
- Success et Error lancent une fonction callback en fonction du résultat

```
this.$http.get('/url')
        .success((data, status) => {
                this.data = data;
        })
        .error((data, status) => {
                console.log(status)
        })
```

```
const data = {'value': 'value'}

$http.post('/url', data)
        .success(() => { ... })
        .error(function(data, status) {
                console.log(status)
        })
```

Le service $http (2)

➢ Liste des méthodes du service $http :

Méthode	Description
$http.get(url, options)	Requête GET
$http.post(url, data, options)	Requête POST
$http.put(url, data, options)	Requête PUT
$http.delete(url, options)	Requête DELETE
$http.head(url, options)	Requête HEAD
$http.jsonp(url, options)	JSONP / Cross domain

Le service $http (3)

➢ $http(options) peut être également appelé comme une fonction

Options	Description
method	GET, PUT, POST, DELETE, HEAD, JSONP
url	url de la requête
params	Objet de propriétés supplémentaires (format Json)
headers	Envoi de propriétés de header au serveur
timeout	Valeur de timeout de la reqûete en millisecondes
cache	Mise en cache true / false
transformRequest	Fonction qui transforme la requête avant envoi
transformResponse	Fonction qui transforme la réponse du serveur

Le service $q : les promises (1)

- Javascript fait des appels asynchrones
- Une promise est un mécanisme qui permet de résoudre ces appels asynchrones
- La promise exécute une fonction callback quand la réponse est reçue

➢ Les requêtes $http retournent une promise

Le service $q : les promises (2)

- **then(successCallback, errorCallback)** reçoit la promise en réponse de l'appel
- **resolve** valide la promise et renvoie un résultat
- **reject** invalide la promise en cas d'erreur et renvoir une erreur

```
public test = () => {
        const deferred = this.$q.defer()

        this.$http.get('/url')
                .success(results => {
                        deferred.resolve(results.data);
                })
                .error((res, errors) => {
                        deferred.reject(errors.status)
                })
        return deferred.promise;
}

$onInit = () => {
    this.test().then(returnedData => {
                this.data = returnedData
        }, erreur => {
                console.log(erreur)
        })
}
```

Le service $q : les promises (3)

- $q.all permet de résoudre plusieurs promises en parallèle
- reject est exécuté si une des promises est en erreur

```
public test = urls => {
        return this.$q.all(urlGets)
            .success(function(results) {
                deferred.resolve(results);
            })
            .error(function(errors){ deferred.reject(errors) })
};

this.test([promise1, promise2, promise3])
        .then(returnedData => {
                const data = []
                for (var i=0, lng = returnedData.length; i < lng; i++) {
                        data.push(returnedData[i].data)
                }
        }, erreur => { // si une des promises est rejetée
                console.log(erreur.status)
        }
```

Créer son service : Service

➢Un service est une classe

```
class MonService {

  public variable
  private autreVar    // non utilisable à l'extérieur du service

  constructor(
        private ServiceInjecte
  ) {}

  public methode = () => {
        this.variable = 'hello'
  }

}

MonService.$inject = ['ServiceInjecte']

app.module('myServices', [])
   .service('MonService', MonService)
```

Configuration

➢ Au lancement d'angular, deux étapes optionnelles :

1. **Phase config** : les services ne sont pas instanciés, mais on peut injecter des providers. Les providers sont configurés avant instance.

2. **Phase run** : initialisation après instanciation des services. Les services sont injectables

```
class Config {
       constructor($httpProvider) {}
}
class Run {
       constructor($http) {}
}

Config.$inject = ['$httpProvider']
Run.$inject = ['$http']

app.module('myApp', [])
  .config(Config)
  .run(Run)
```

Créer son service : Provider

- Un provider doit retourner une méthode $get
- Pour être utilisé en Config, il doit être suffixé par "Provider"

```
class NomProvider {
    public methodeConfig = () => {}
    public $get = () => {
       return {
           methodeRun: () => {}
       }
    }
}

class Config {
    constructor(nomProvider) { nomProvider.methodeConfig() }
}
class Run {
    constructor(nomProvider) { nomProvider.methodeRun() }
}

Config.$inject = ['NomProviderProvider']
Run.$inject = ['NomProvider']

app.module('myProviders', [])
       .provider('NomProvider', NomProvider)
       .config(Config)
       .run(Run)
```

$httpProvider - Interceptors (1)

- **$httpProvider** contient un Array d'interceptors
- Un Interceptor est une **factory** que l'on peut ajouter à cet Array
- Ils permettent **d'intercepter** les requêtes et réponses $http

Fonction	Description
request	Appelée avant l'envoi de la requête $http
response	Appelée après la reception de la réponse $http
requestError	Appelée en cas d'erreur de requête
responseError	Appelée en cas d'erreur de réponse

$httpProvider - Interceptors (2)

```
class Config {
    constructor($httpProvider) {
        $httpProvider.interceptors.push('MonInterceptor')
    }
}

class MonInterceptor {
    constructor( private TokenSrv ) {}

    public request: config => {
        if (!this.TokenSrv.isAnonymus) {
            config.headers['x-session-token'] = this.TokenSrv.token
        }
        return config
    }

    public responseError: response => {
        // gestion d'erreur
        return response
    }

}
```

Angular <= 1.4 - Autres services

Service	Description
value	Simple objet Json clé / valeur. Non injectable en config
constant	Simple objet Json clé / valeur. Injectable en config
factory	Retourne un objet Json

Les modules, dépendance entre modules

- Il est conseillé de découper son application en modules
- Le module de notre app peut dépendre d'autres modules injectés
- Les modules peuvent être organisés par fonctionnalité de l'application

```
angular.module('monModule1', [])
      .component(...)
      .service(...)

angular.module('monModule2', [])
      .service(...)

angular.module('app', ['monModule1', 'monModule2'])
```

En conclusion

- ✓ Les services sont des Singleton
- ✓ Ils sont utilisés pour la logique métier
- ✓ Les Services sont instanciés au run
- ✓ Les Providers sont utilisables en phase de config

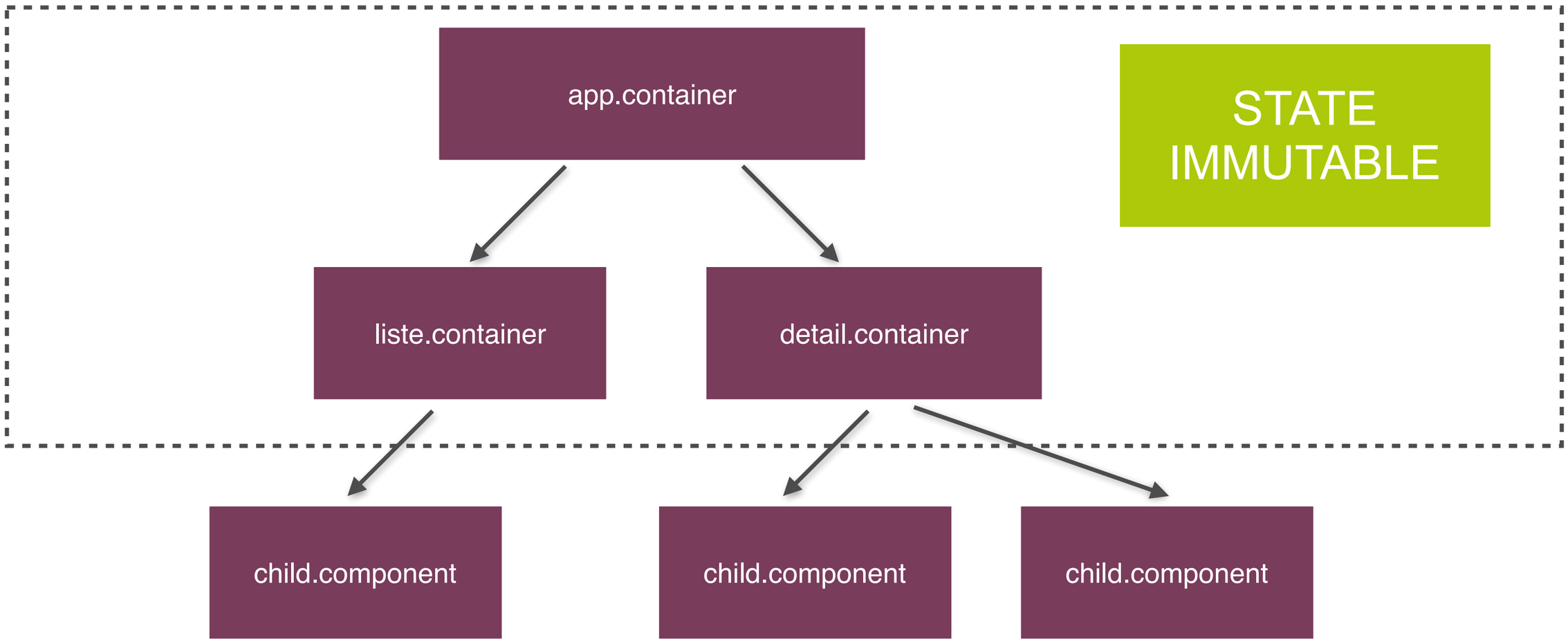
Redux

Objectifs du chapitre

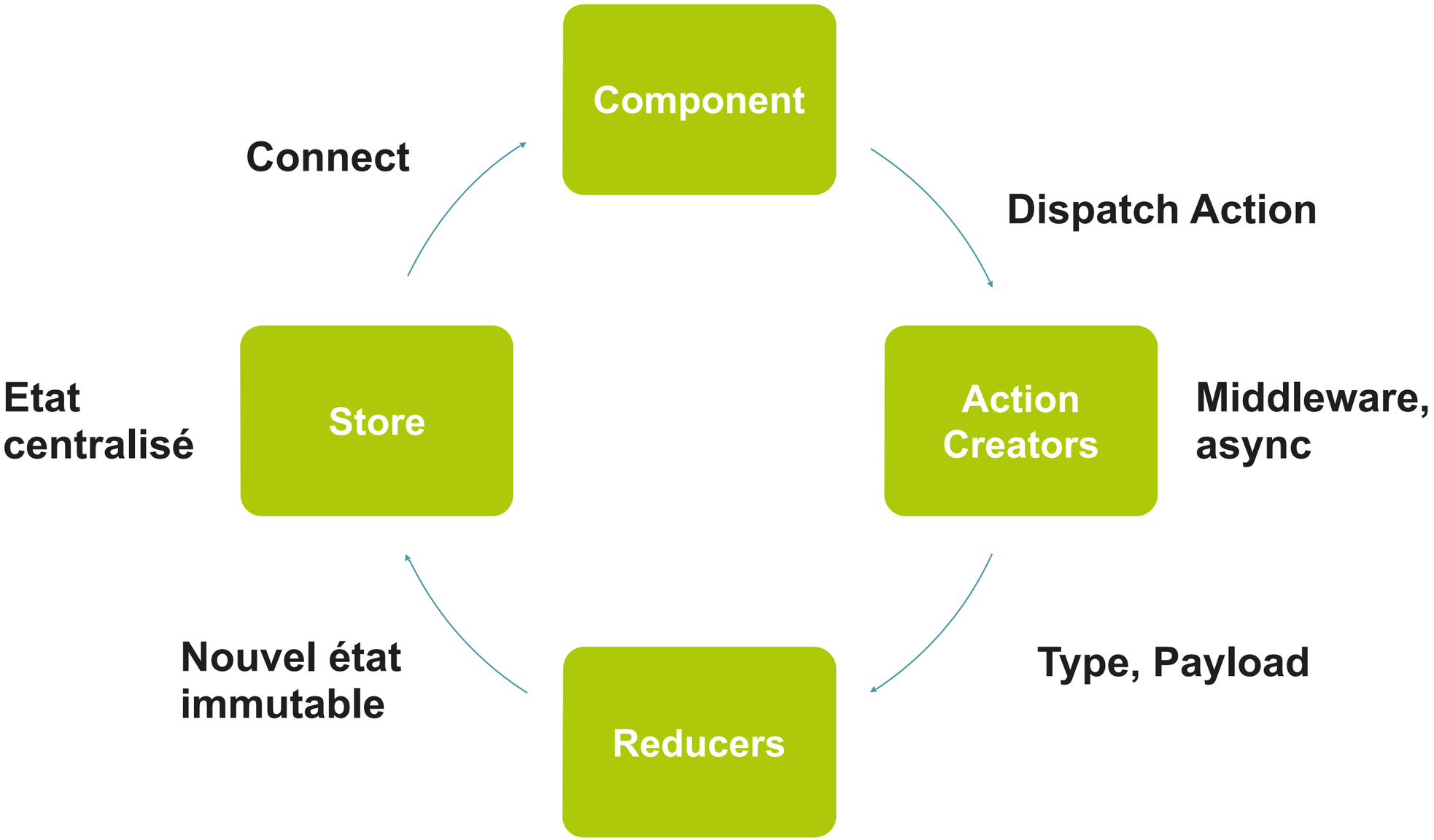
- Découvrir les principes de flux unidirectionnel
- Découvrir Redux
- Utiliser ng-redux

Detection du changement (1)

- binding '<' = binding unidirectionnel de parent à enfant
- => favoriser un changement de données **immutable** (ou primitif)
- => **Containers** = composants logiques, manipulant les données, propres à l'application
- => **Components** = onPush, destinés à l'UI



Redux - flux unidirectionnel



Redux - Caractéristiques

- **Store** : Etat de toute l'application, centralisé, immutable
- **Actions** : la vue dispatche les actions utilisateur
- **Action creators** : centralisation des actions
- **Middleware** : Gestion asynchrone, lancement de fonctions avant ou après reduce
- **Reducers** : Retournent un nouvel état suite aux actions
- **Connect** : Les components souscrivent au store et se mettent à jour à chaque nouvel état

ng-redux : implémentation de redux pour Angular 1

ngRedux - rootReducer

- Combine : combiner les reducers dans un rootReducer

```
import { combineReducers } from 'redux'

import userReducer from './user.reducer'
import autreReducer from './autre.reducer'

const rootReducer = combineReducers({
  userReducer,
  autreReducer
})

export default rootReducer
```

ngRedux - instancier le store

```
import ngRedux from 'ng-redux'
import Config from './config'

angular.module('App', [
   ...,
   ngRedux
])
.config(Config)

angular.bootstrap(document.documentElement, ['App'])
```

```
import rootReducer from './reducers/index'
import * as createLogger from 'redux-logger'
import thunk from 'redux-thunk'

class Config {

  constructor(
    $ngReduxProvider
  ) {

    $ngReduxProvider.createStoreWith(rootReducer, [thunk, createLogger()])

  }
}

Config.$inject = ['$ngReduxProvider']
export default Config
```

ngRedux - Reducers

- Les reducers sont des fonction pures

```
import { USER } from '../actions/user.actions'

function userReducer(state: any = {}, action: any) {

  switch (action.type) {
    case USER.LOAD:
      return action.user || {}

    case USER.UPDATE:
      return action.user

    default:
      return state
  }

}

export default userReducer
```

ngRedux - Action Creators

```
export const USER = {                                  // constante représentant chaque action
  LOAD_REQUEST: 'USER_LOAD_REQUEST',
  LOAD_RESPONSE: 'USER_LOAD_RESPONSE',
  LOGOUT: 'USER_LOGOUT'
}

class UserActions {
    constructor(private userService) {}                        // inject service

    login = credentials => {
        return (dispatch, getState) => {               // thunk : retourne une fonction

            dispatch({                                 // dispatch synchone
              type: USER.LOAD_REQUEST
            })
            return this.UserService.getUser(credentials).then(user => {
               dispatch({                                        // dispatch synchrone
                 type: USER.LOAD_RESPONSE,
                 user
               })
            }
        }
    }

    logout = () => {                                   // retour simple objet
      return {
        type: USER.LOGOUT
      }
    }
}
UserActions.$inject = ['UserService']
export default UserActions
```

ngRedux - connect

```
class LoginController {
  private disconnect

  constructor(
    private $ngRedux,
    private userActions
  ) {}

  $onInit = () => {
    this.disconnect = this.$ngRedux.connect(this.mapStateToThis, () => {})(this)
  }

  $onDestroy = () => {
    this.disconnect()
  }

  public login = credentials => {
    this.$ngRedux.dispatch(this.userActions.login(credentials))
  }

  private mapStateToThis(state) {
    return {
      error: state.userReducer.error
    }
  }
}

LoginController.$inject = ['$ngRedux', 'UserActions']
```

En conclusion

- Optimisation de la **détection** de changement dans l'application
- redux adopte un flux unidirectionnel de données **immutables**
- L'état de l'application est **centralisé** en un seul endroit
- La séparation container / component permet le **partage** et la **réutilisation** de components UI

UI Router

Objectifs du chapitre

- Configurer un routeur
- Naviguer dans son application

Principes de routing

- Exploitation du # de l'url :

  http://www.monsite.com/#url-route

- Associer la route à une URL (partir variable), un état
- Associer la route à un template / container
- Précharger des données avant accès à a route : resolve
- Définir des sous-routes

Configuration

- $stateProvider : configuration des routes
- $urlRouterProvider : route par défaut

```
class Config {

  constructor(
   $stateProvider,
    $urlRouterProvider
  ) {

    $stateProvider
      .state('toys', {
        url: '/toys',
        template: '<toy-container/>'    // ou templateUrl="path/to/template"
      })

    $urlRouterProvider.otherwise(($injector) => {    // route par défaut
      const $state = $injector.get('$state')
      $state.go('toys')                 // redirection vers 'toys'
    })

  }
}

Config.$inject = ['$stateProvider', '$urlRouterProvider']
export default Config
```

Vue et navigation

- ui-view : directive cible du template
- $state.go et ui-sref : navigation

```
<section>
    <header></header>
    <div ui-view></div>                    // conteneur template
</section>
```

```
public goToTarget = () => {
    this.$state.go('toys')
}
```

```
<div>
    <a ui-sref="state.name">Aller vers toys</a>
</div>
```

Routes et paramètres dynamiques

- paramètre d'url
- gestion des paramètres avec $stateParams

```
$stateProvider
  .state('toy', {
    url: '/toy/:idToy',          // idToy : paramètre dynamique
    template: '<toy-container/>'
  })
```

- Routing html

```
<a ui-sref="toy({idToy: $ctrl.id})">View Toy</a>
```

- Routing avec service $state

```
this.$state.go('toy', {idToy: this.idToy})
```

- Le service $stateParams contient les paramètres d'url passés

```
this.$stateParams.idtoy
```

Resolve

- appel de méthodes avant route active
- paramètre resolve

```
$stateProvider
  .state('toy', {
    url: '/toys',
    template: '<toy-container/>',

    resolve: {
       toys: ['$ngRedux', 'ToyActions', ($ngRedux, ObjectDetailActions) => {
            return $ngRedux.dispatch(
              ToyActions.getToys()
            )
       }]
    }
  })
```

- Route activée quand le resolve retourne une promise résolue
- Route non activée en cas de promise.reject

Sous-routes et routes abstraites

- Route abstract : non accessible par Url
- Accès à la sous-route : les paramètres de la route abstraite sont pris en compte

```
$stateProvider
  .state('main', {
    abstract: true,
    template: `
       <header>...</header>
   <div ui-view></div>
    `,
    resolve: ...
 })

  $stateProvider
      .state('main.toys', {
               url: '/toys',
               template: '<toy-container/>'
       })

  $stateProvider
      .state('main.selection', {
               url: '/selection',
               template: '<selection-container/>'
       })
```

Routes nommées

- La paramètre view invalide les paramètres template et templateUrl
- On peut avoir plusieurs routes nommées dans un template

```
$stateProvider
  .state('main', {
    abstract: true,
    template: `
       <header ui-view="header"></header>
        <div ui-view="body"></div>
    `
})

  $stateProvider
      .state('main.toys', {
               views: {
               header: { . . . }, // paramètres de route classiques
                            body: { . . . }
                  }
       })
```

En conclusion

- UI router permet de simuler une pagination dans l'app
- Angular fournit un router par défaut : ngRoute, trop limité

Les filtres

Objectifs du chapitre

- Découvrir ce que sont les filtres
- Utiliser les filtres fournis par Angular
- Créer ses filtres personnalisés

Les filtres standard

- Les filtres permettent la transformation des données **avant affichage** du template
- Les filtres sont utilisés dans les **directives et expressions**

Syntaxe : {{ expression | filter:param1:param2:param*n* }}

Filtre	Description
currency	Formate un nombre au format devise
date	Formate une date
limitTo	Filtre ***n*** caractères d'un string ou ***n*** éléments d'un array
uppercase	Convertit une chaîne en majuscules
lowercase	Convertit une chaîne en minuscules
number	Convertit un nombre en string
orderBy	Tri d'un tableau selon une expression
json	Convertit un objet javascript au format json
filter	Filtre un tableau en fonction d'un critère de recherche

Les filtres : currency

{{currency_expression | currency : 'sigle monnaie'}}

```
<div>
        <p>{{$ctrl.pognon | currency}}</p>                  // 1 230,00 €

        <p>{{$ctrl.pognon | currency:'$'}}</p>             // 1 230,00 $
</div>
```

➢ Par défaut, le sigle monnaie est celui de la locale. Le sigle est optionnel

Les filtres : date

{{date_expression | date : 'format'}}

```
<div>

        <p>
                {{$ctrl.myDate | date:'d MMM yyyy'}} -
                {{$ctrl.myDate | date:'hh : mm : ss'}}
        </p>

</div>
```

Les filtres : limitTo

{{limitTo_expression | limitTo : valeur}}

```
class MyComponentController {
    public limiter = 'Septembre'
    public datas = [1, 2, 3, 4, 5]
}

<div>
        <p>{{$ctrl.limiter | limitTo:3}}</p> <!-- affiche 'Sep' -->

        <p>
                <span ng-repeat="data in $ctrl.datas | limitTo:3">{{data}}</span>
        </p> <!-- affiche 123 -->
</div>
```

Les filtres : uppercase / lowercase

{{upper_expression | uppercase}}

{{lower_expression | lowercase}}

```
class MyComponentController {
    public upper = 'septembre'
    public lower = 'OCTOBRE'
}


<div>
        <p>{{$ctrl.upper | uppercase}}</p> <!-- affiche 'SEPTEMBRE' -->

        <p>{{$ctrl.lower | lowercase}}</p> <!-- affiche 'octobre' -->
</div>
```

Les filtres : number

{{ number_expression | number : fraction}}

```
class MyComponentController {

        public nb = 1000
        public bigNb = 11230450.12
        public virgule = 1234.56789

}

<div>
        <p>{{$ctrl.nb | number}}</p>            <!-- affiche '1 000' –>

        <p>{{$ctrl.bigNb | number}}</p>         <!-- affiche '11 230 450,12' -->

        <p>{{$ctrl.virgule | number:3}}</p>     <!-- affiche '1234,568' -->

</div>
```

➢ Le chiffre est arrondi en fonction de la valeur de la fraction après la virgule.

Les filtres : orderBy

{{ orderBy_expression | orderBy : expression : reverse}}

```
class MyComponentController {
        public objets = [
                {'prenom': 'laurent'},
                {'prenom': 'anne'},
                {'prenom': 'julien'}
        ]
}

<div>

        <ul>
           <li ng-repeat="objet in $ctrl.objets | orderBy:'-prenom':true">
                {{objet.prenom}}
           </li>
        </ul>

</div>
```

- L'expression du tri peut être préfixée par '-' (tri descendant) ou '+' (tri ascendant)
- reverse implique un tri descendant si true, ascendant si false. Optionnel

Les filtres : json

{{json_object | json}}

```
<pre>{{ {'nom':'valeur'} | json }}</pre>

Affiche :

{
  "nom": "valeur"
}
```

➢ Utile pour débugger

Les filtres : filter (1)

{{ filter_expression | filter : expression : comparator}}

- expression peut être soit un string, soit un objet, soit une fonction
- comparator détermine si l'élement recherché doit être contenu dans le résultat ou égal au résultat

```
class MyComponentController {
        public saisie
        public objets = [
           {'prenom': 'laurent'},
           {'prenom': 'laure'},
           {'prenom': 'anne'},
           {'prenom': 'julien'}
        ]
}

<div>
   <input type="text" ng-model="$ctrl.saisie">
   <ul>
      <li ng-repeat="objet in $ctrl.objets | filter: $ctrl.saisie : true">
        {{objet.prenom}}
      </li>
   </ul>
</div>
```

Les filtres : filter (2)

```
class MyComponentController {

        public objets = [
           {'nom': 'durand', 'prenom': 'laurent'},
           {'nom': 'martin', 'prenom': 'laurent'},
           {'nom': 'dupont', 'prenom': 'laure'}
        ]

}

<div>
   <ul>
      <li ng-repeat="objet in $ctrl.objets | filter: {'nom': 'd', 'prenom':
'laurent'}">
        {{objet.nom}}
        {{objet.prenom}}
      </li>
   </ul>
</div>


Affiche :

'durand laurent'
```

Les filtres : filter (3)

```
class MyComponentController {
          public saisie
          public objets = [
             {'prenom': 'laurent'},
             {'prenom': 'laure'},
             {'prenom': 'anne'},
             {'prenom': 'julien'}
          ];

          public saisieFct =(val, i) => {
             if(!this.saisie) { return false }

             const cond1 = val.prenom >= this.saisie;
             const cond2 = val.prenom.substr(0, this.saisie.length) === this.saisie
             return cond1 && cond2
          }
}

<div>
   <input type="text" ng-model="$ctrl.saisie">
   <ul>
      <li ng-repeat="objet in $ctrl.objets | orderBy:'prenom' | filter: $ctrl.saisieFct">
         {{objet.prenom}}
      </li>
   </ul>
</div>
```

Les filtres custom

```
class myComponentController {
            public countries = [
              'france',
              'belgique',
              'allemagne',
              'luxembourg'
            ]
}

angular.module('app')
    .filter('capitalize', () => {
        return (input, condition) => {

              return condition ?
                 input.charAt(0).toUpperCase() + input.slice(1) :
                 input

        }
    })
```

```
<ul>
    <li ng-repeat="country in $ctrl.countries">{{ country | capitalize:true }}</li>
</ul>
```

Le service $filter

➢ Le service $filter permet d'appliquer un filtre à une variable javascript

Syntaxe : $filter('nomFilter')(Array, params)

```
class myComponentController {
  public country = 'Berlin'

  constructor (private $filter) {}

  $onInit = () => {
        this.$filter('uppercase')(this.country)
  }

}


<div>{{$ctrl.country}}</div>    // affiche 'BERLIN'
```

En conclusion

- ✓ Les filtres permettent de **manipuler** les données de la vue
- ✓ $filter peut être utilisé dans les **services** afin de filtrer ses données

Les formulaires

Objectifs du chapitre

- Créer des formulaire sous Angular
- Valider ses formulaires
- Soumettre ses formulaires

Forms et saisie

- Form groupe un lot de contrôles de saisie
- Input, Textarea et Select permettent la saisie d'informations
- ng-model lie chaque élément du Form au controller
- Angular fournit des directives permettant la validation de formulaires
- Il est possible d'imbriquer plusieurs balises form

ng-model, input text et textarea

```
class myComponentController {
        public title
        public text

        public update = () => {
                console.log(this.text)
        }
}

<form novalidate>

    <input type="text" ng-model="$ctrl.title" ng-change="$ctrl.update()">

    <textarea ng-model="$ctrl.text" ng-change="$ctrl.update()"></textarea>

</form>
```

➢ la directive ng-change est executée au moment de la modification du champ

Binding de checkboxes

- Par défaut les checkboxes ont pour valeur true / false

- Les directives ng-true-value / ng-false-value permettent d'affecter une autre valeur au check

```
class myComponentController {
          public newsletter
          public partenaires
          public update = () => {

                    // this.newsletter
                    // this.partenaires

          }
}


<form novalidate>
    <p>
          <input type="checkbox" ng-model="$ctrl.newsletter" ng-change="$ctrl.update()">
          S'inscrire à la newsletter
    </p>
    <p>
          <input type="checkbox" ng-model="$ctrl.partenaires"
                    ng-true-value="yes" ng-false-value="no"
                    ng-change="$ctrl.update()">
          S'inscrire aux offres partenaires
    </p>

</form>
```

Binding de radio buttons

➢ Angular lie les input[radio] entre eux en spécifiant un ng-model commun

```
class myComponentController {
          public newsletter

          public update = () => {

                    // this.newsletter

          }
}
```

```
<form novalidate>
    <div>S'inscrire à la newsletter</div>

    <p>
        <input type="radio" ng-model="$ctrl.newsletter" ng-change="$ctrl.update()"
                        value="oui"> Oui

        <input type="radio" ng-model="$ctrl.newsletter" ng-change="$ctrl.update()"
                        value="non"> Non

    </p>
</form>
```

Binding de select boxes (1)

➢ Deux options possibles :

- **Utilisation des balises html <option>**

  Dans tous les cas <option> peut être utilisé pour afficher une valeur par défaut

  ng-value permet une affectation dynamique de valeur aux options via ng-repeat

- **Utilisation de la directive ng-options, avec les possibilités suivantes :**

  Pour les tableaux : *label for value in array*

  pour les objets : *label for (key , value) in object*

Binding de select boxes (2)

```
class myComponentController {
        public myForm = {
                bg: null,
                color: null,
                options: [
                        {value: 'blue',  name: 'Bleu'},
                        {value: 'red',   name: 'Rouge'},
                        {value: 'green', name: 'Vert'}
                ],
        }
        public setColor: () => {
                this.myForm.bg = {
                        background: this.myForm.color
                }
        }

        $onInit = () => {
                this.setColor()
        }
}

<form ng-style="$ctrl.myForm.bg" novalidate>
       <select ng-model="$ctrl.myForm.color"
                ng-options="obj.value as obj.name for obj in $ctrl.myForm.options"
                ng-change="$ctrl.setColor()"
        >
                 <option value="">Votre couleur</option>
       </select>
</form>
```

Validation des formulaires

- Angular fournit des directives de manipulation et validation des formulaires
- Si un champ est invalide, le ng-model associé n'est pas mis à jour : il aura pour valeur 'undefined'

Directive	Description
ng-minlenght / ng-maxlength	Longueur min/max d'un champ
ng-pattern	RegExp sur saisie
ng-required	Champ obligatoire
ng-disabled	Champ désactivé (true/false)
ng-readonly	Champ en lecture seule
ng-trim	Suppression des blancs non significatifs
ng-change	Action de modification d'un champ

Validation : minlength , maxlength

```
class myComponentController {

        public  name = 'Laure Dupont'
        public  update = () => {
                        // this.name
        }

}

<form novalidate>

    <input type="text" ng-model="$ctrl.name" ng-minlength="5"
        ng-maxlength="12" ng-change="$ctrl.update()">

</form>
```

Validation : ng-pattern

➢ Ng-pattern valide le champ en fonction d'une expression régulière

```
class myComponentController {
        public name
        public update = () => {
                // this.name
        }
}


<form novalidate>
      <input type="text" ng-model="$ctrl.name" ng-pattern="/^[0-9]{1,128}$/"
              ng-change="$ctrl.update()">
</form>
```

Validation : ng-required

➢ Si ng-required est true, la saisie du champ est obligatoire

```
class myComponentController {
        public name
        public update = () => {
                // this.name
        }
}

<form novalidate>
        <input type="text" ng-model="$ctrl.name" ng-required="true"
                ng-change="$ctrl.update()">
</form>
```

Validation : ng-trim

- Par défaut, les blancs non significatifs sont supprimés de la saisie.
- Si ng-trim est false, les blancs sont pris en compte
- ng-trim n'est pas compatible avec input[password]

```
class myComponentController {
        public modele1
        public modele2

        public update = () => {
                // si saisie de "   valeur   "
                console.log(this.modele1) // => 'valeur'
                console.log(this.modele2) // => '  valeur  '
        }

}


<input type="text" ng-model="$ctrl.modele1" ng-change="$ctrl.update()">
<input type="text" ng-model="$ctrl.modele2" ng-trim="false"
        ng-change="$ctrl.update()">
```

Etats de validation des formulaires (1)

- En spécifiant un attribut name à la balise form, celui-ci devient une propriété du scope
- En spécifiant des attributs name aux balises du formulaire, celles-ci deviennent des propriétés du scope formulaire

```
class myComponentController {
        public nom
        public prenom
        public update = () => {
            // this.nom
            // this.prenom
        }
}

<form name="myForm" novalidate>
        <input type="text" ng-model="$ctrl.nom" name="nom"
               ng-change="$ctrl.update()">
        <input type="text"
               ng-model="$ctrl.prenom"
               name="prenom" ng-change="$ctrl.update()">
</form>
```

Etats de validation des formulaires (2)

➢ Ces éléments vont permettre de tester l'état de la saisie en fonction de 4 paramètres :

Propriété	Description
$pristine	A pour valeur true si le champ ou le formulaire n'ont pas été modifiés, sinon false
$dirty	A pour valeur false si le champ ou le formulaire n'ont pas été modifiés, sinon true
$valid	A pour valeur true si l'élement est valide, sinon false
$invalid	A pour valeur true si l'élément est invalide, sinon true

Etats de validation des formulaires (3)

```
<style>
        .invalide { color: red; background: yellow; }
</style>

<form name="myForm">
        <input type="text"
                ng-model="nom"
                name="nom"
                ng-required="true" ng-minlength="3"
                ng-change="update()"
                ng-class="{invalide: myForm.nom.$invalid}"
        >
        <button ng-disabled="myForm.$invalid">Valider</button>
</form>
```

Formulaires et styles

Classe	Classe
ng-valid	ng-valid-[key]
ng-invalid	ng-invalid-[key]
ng-pristine	ng-touched
ng-dirty	ng-untouched

```
<style>
input.ng-invalid  { color: red; background: yellow; }
input.ng-pristine { color: black; backgrouns: white; }
input.ng-valid-required { color: black; border: green; background: white;}
</style>

<form name="myForm">
        <input type="text"
                ng-model="nom"
                name="nom"
                ng-required="true" ng-minlength="3"
                ng-change="update()"
        >
</form>
```

Soumission des formulaires (1)

➢ Deux méthodes de soumission des formulaires :

- Utiliser un **ng-submit** sur la balise <form>
- Utiliser un **ng-click** sur une balise <button>

➢ Dans les deux cas, exécution d'une méthode de controller

Soumission des formulaires (2)

```
class myFormController {
        public formData = {
                nom: null,
                prenom: null
        }
        public envoi = () => {

                this.$http.put('/url', this.formData)

        }
}

<form name="myForm" ng-submit="$ctrl.envoi()">
        <input type="text"
                ng-model="$ctrl.formData.nom"
                name="nom"
                ng-required="true" ng-minlength="3"
                ng-class="{invalide: myForm.nom.$invalid}"
        >
        <input type="text"
                ng-model="$ctrl.formData.prenom"
                name="prenom"
                ng-required="true" ng-minlength="3"
                ng-class="{invalide: myForm.prenom.$invalid}"
        >

        <input type="submit" value="Envoi" ng-disabled="myForm.$invalid">
</form>
```

En conclusion

- ✓ Pour AngularJS, les éléments de formulaire sont des directives
- ✓ Les contrôles de formulaire peuvent être réalisés directement dans le DOM

Les directives

Objectifs du chapitre

- Manipuler le DOM avec les directives Angular
- Créer ses propres directives

Introduction

- Les directives gèrent la manipulation du DOM

- Angular en fournit un certain nombre, comme ng-if, ng-repeat, ng-view

- On peut créer ses directives

- Avec l'ajout des components, seules les directives de type attribut sont intéressantes

Ma première directive

```
<script>
        angular.module('app')
           .directive('tagName', [function() {
                var mesParams = {
                        restrict: 'E',
                        template: '<p>Ma directive</p>'
                };
                return mesParams
        }])
</script>

<tag-name></tag-name>
```

- Création d'une directive nommée <tag-name>, de type élément (restrict = 'E')
- tag-name est converti au format camel case côté javascript => tagName
- La directive retourne un objet qui définit ses caractéristiques
- A l'exécution, le contenu de template est inséré dans la balise <tag-name>

➢ Comme pour les controllers et les services, **on peut injecter des services**

Paramètre restrict

- Les directives peuvent être de 4 types, définis par restrict
- Chaque directive peut comporter une combinaison de ces 4 options

```
angular.module('app')
      .directive('tagName', [function() {
            const mesParams = {
                  restrict: 'EACM',
                  template: '<p>Ma directive</p>'
            }
            return mesParams
      }])
</script>

<tag-name></tag-name>                    Type E = Element
<div tag-name></div>                     Type A = Attribute
<div class="tag-name"></div>             Type C = Class
<!-- directive: tag-name -->             Type M = Comment
```

Paramètres template et templateUrl

- template contient le code html à afficher pour la directive
- templateUrl permet de spécifier le chemin d'accès à un fichier contenant le html

```
app.directive('nom', [function() {
        return {
                restrict: 'AE',
                template: '<div>Ma directive {{expression}}</div>'
        }
}])

app.directive('adresse', [function() {
        return {
                restrict: 'AE',
                templateUrl: '/dossier/templates/adresse.html'
        }
}])
```

Paramètre replace

- replace = true : remplace la balise de la directive
- replace = false (default) : la balise de la directive est conservée

```
app.directive('nom', [function() {
        return {
                restrict: 'E',
                replace: true,
                template: '<p>Mon contenu</p>'
        }
}])


<nom></nom>

<!-- HTML affiché : -->
<!-- <p>Mon contenu</p> -->
```

Scope et scope isolé (1)

➢ Par défaut, le scope de la directive hérite du scope parent

```
<script>
	app.controller ('myCtrl', [function() {
		$scope.content = 'interactif'
	}])
	app.directive('nom', [() => {
		return {
			restrict: 'E',
			replace: true,
			template: '<p>Mon contenu {{content}}</p>'
		}
	}])
</script>

<div ng-controller="myCtrl">
	<nom></nom>
</div>
```

Scope et scope isolé (2)

➢ Le paramètre **scope** permet de définir un scope isolé

- **false** : valeur par défaut, scope non isolé qui hérite du scope parent par référence. Equivalent à ne pas déclarer le paramètre scope.
- **true** : scope qui hérite du scope parent, mais par valeur
- **{}** : scope totalement isolé, ignore le scope parent

```
app.directive('nom', [() => {
        return {
                restrict: 'E',
                replace: true,
                template: '<p>Mon contenu {{content}}</p>',
                scope: {} // scope isolé
        }
}])
```

Scope isolé : Passer des paramètres (1)

➢ On peut passer 3 types de paramètres à une directive, en utilisant ces préfixes :

- **@** : passage par valeur (binding simple)
- **=** : passage par référence (double binding)
- **&** : binding de méthode

Scope isolé : Passer des paramètres (2)

```
class component{
        public monNom = 'toto'
        public monPrenom = 'titi'
        public getAge = () => {
                return 100
        }
}
app.directive('personne', [() => {
        return {
                restrict: 'E',
                replace: true,
                template: `<p>
                        {{$ctrl.nom}} {{$ctrl.prenom}} {{$ctrl.methode()}}
                </p>`,
                scope: {
                        nom: '@',
                        prenom: '=lastname',
                        methode: '&'
                }
        }
}])

<component>
  <personne nom="{{$ctrl.monNom}}" lastname="$ctrl.monPrenom"
methode="$ctrl.getAge()"></personne>
</component>
```

Paramètre compile

- compile permet de définir comment la directive va modifier le HTML
- compile initialise la directive. Le scope n'est pas encore défini
- il retourne une fonction link qui permet d'agir sur le scope

```
app.directive('personne', ['$filter',
        $filter => {
                return {
                        restrict: 'E',
                        replace: true,
                        template: '<p>{{$ctrl.nom}}</p>',
                        scope: {
                                nom: '='
                        },
                        compile: function(element, attrs) {
                           element.css({color: 'red'});
                           return function(scope, element, attrs) {
                              scope.nom = $filter('uppercase')(attrs.nom);
                           };
                        }
                }
        }])
```

➢ Dans un ng-repeat, compile est exécuté une fois, link à chaque itération

Paramètre link

➢ link peut être directement déclaré sans compile

```
app.directive('personne', ['$filter',
        $filter => {
                return {
                        restrict: 'E',
                        replace: true,
                        template: '<p>{{nom}}</p>',
                        scope: {
                                nom: '='
                        },
                        link:(scope, element, attrs) => {

                           scope.nom = $filter('uppercase')(attrs.nom)
                           element.css({color: 'red'})

                        }
                }
        }])
```

Angular.element et jqLite

- element permet de manipuler le DOM
- Angular fournit jqLite pour cette manipulation, une version simplifiée de jQuery
- Si jQuery est déclaré avant Angular, ce dernier utilisera jQuery
- La manipulation du DOM peut être aussi faite avec angular.element()

```
app.directive('personne', [() => {
        return {
                restrict: 'E',
                template: '<p>{{$ctrl.nom}}</p>',
                link: (scope, elm, attrs) => {
                   var domElements = document.querySelectorAll('p');
                   angular.element(domElements).css({color: 'red'});
                }
        }
}])
```

jqLite

addClass()
after()
append()
attr()
bind()
children()
clone()
contents()
css()
data()
detach()
empty()
eq()
find()
hasClass()
html()
next()
on()
off()
one()
parent()
prepend()
prop()
ready()
remove()
removeAttr()
removeClass()
removeData()
replaceWith()
text()
toggleClass()
triggerHandler()
unbind()
val()
wrap()

La transclusion

➢ La transclusion est un mécanisme qui permet de conserver le contenu d'une directive et de l'injecter dans le template

```
app.directive('personne', [() => {
      return {
              restrict: 'E',
              replace: true,
              scope: {
                      nom: '@'
              },
              transclude: true,
              template: '<p>{{nom}} <span ng-transclude></span></p>'
      }
   }])

<personne nom="Dupont">Alfred</personne>

<!-- résultat : —>
<!-- Dupont Alfred -->
```

En conclusion

- ➢ Seules les directives attribut sont utiles avec ng > 1.5
- ➢ Eviter l'utilisation de jqLite, qui ne sera pas inclus dans ng2
- ➢ Avec les components, préférer l'injection du service **$element**

Les tests unitaires

Objectifs du chapitre

- Configurer Karma
- Jasmine
- Stratégies de test des objets angular

Tests unitaires - Les outils

- Karma : moteur de lancement des tests et reports
- Lancement dans un browser, ou dans PhantomJS (headless browser)
- Langage d'assertion : Jasmine, Behaviour Driven Development (BDD)
- Librairie helper : angular.mocks

Karma

package.json

```
"scripts": {
   "test": "karma start"
}

>> npm test
```

karma.conf.js

```
module.exports = function (config) {
  config.set({

    basePath: './',              // répertoire de base des fichiers utilisés
    frameworks: ['jasmine'],     // framework de test : jasmine, mocha etc.
    files: [
      'spec.ts',
      'www/**/*.spec.ts'
    ],
    exclude: [],         // liste des fichiers à exclure
    preprocessors: {             // liste des preprocesseurs : compilation, coverage etc.
      'spec.ts': ['webpack'],
      'www/js/**/!(*.spec)+(.ts)': ['coverage', 'webpack']
    },
    webpack: webpackConfig,      // fichier de config webpack à importer
    reporters: ['spec', 'coverage'],   // report en console
    coverageReporter: {          // reporting en sortie
      reporters: []
    },
   autoWatch: true,              // surveille les modifications de fichier
   browsers: ['PhantomJS'],              // browsers de lancement de test. ie Chrome, PhantomJS...
   singleRun: false              // tests en une passe ou en continu
 })
}
```

Jasmine - Assertions

- **Describe** : Bloc d'un élément à tester
- **BeforeEach** / **AfterEach** : code exécuté avant/après chaque spec.
- **BeforeAll / AfterAll** : code exécuté avant / après la suite de tests
- **It** : Bloc de spécification à tester
- **Expect** : Condition de vérification du résultat

```
describe('MonObjet', () => {

    beforeEach(() => {

    })

    it('Should do something',() => {
       expect(true).toBe(true)
    })

    it('Should add some value',() => {
       expect(false).not.toBe(true)
    })

    afterEach(() => {

    })

})
```

Jasmine : Les matchers

toBe()	Egalité stricte (ie ===)
toEqual()	Egalité (ie ==)
toMatch()	RegExp
toBeDefined() / toBeUndefined()	Défini ou undefined
toBeNull()	est null
toBeTruthy() / toBeFalsy()	truthy / falsy
toContain()	contient un élément dans un tableau
toBeLessThan() / toBeGreaterThen()	Inférieur ou supérieur
toThrow()	A lancé une exception
not	Négation (ex : not.toBe(true))

Jasmine : Les spies

- **Spy** : mock d'une méthode d'objet

spyOn	spyOn(MonObjet, 'maMethode')
.and.returnValue()	spyOn(MO, 'ma').and.returnValue('resultat')
.and.callFake()	spyOn(MO, 'ma').and.callFake(() => { return true })
.and.callThrough()	spyOn(MO, 'ma').and.callThrough()
.toHaveBeenCalled()	expect(MO.ma).toHaveBeenCalled()
.tohavebeenCalledWith()	expect(MO.ma).toHaveBeenCalledWith('une valeur')
jasmine.createSpyObj()	jasmine.createSpyObj(MO, ['methode1', 'methode2', 'methode3'])

Jasmine : Asynchrone

- beforeEach, beforeAll, afterEach, afterAll, it retournent une fonction à exécuter en fin d'appel asynchone
- done() et done.fail() stoppent l'appel asynchrone

```
describe('MonObjet', () => {

    it('Should do something', done => {

        setTimeout(() => {

            expect(doSomething).toBe(true)

            done()                                          // exécuté en fin d'async

        }, 1000)

        if (uneErreur) {
            done.fail()
        }

    })

})
```

Angular.mock

- **angular.mocks.module : Lancer un module**
- **angular.mocks.inject : injection des services nécessaires au test**

```
describe('MonObjet', () => {

    var $rootScope

    beforeEach(angular.mock.module('ToyStore'))

    beforeEach(angular.mock.inject((_$rootScope_) => {
        $rootScope = _$rootScope_
    }))

    it('Should be defined', () => {

         expect($rootScope).toBeDefined()

    })
})
```

Tester un Service

```
class MonService {
    constructor(private ExternalService) {}
    public initialize = () => {
        return ExternalService.uneMethode()
    }
}
```

```
describe('MonService', () => {

    let MonService, ExternalService

    beforeEach(angular.mock.module('MonApp'))

    beforeEach(angular.mock.inject((_MonService_, _ExternalService_) => {
        MonService = _MonService_

        ExternalService = _ ExternalService_
        spyOn(ExternalService, 'uneMethode').and.returnValue('hello')
    }))

    it('Should do something', () => {
        const result = MonService.initialize()

        expect(ExternalService.uneMethode).toHaveBeenCalled()
        expect(result).toBe('hello')
    })
})
```

$httpBackend

- Permet de mocker un appel $http
- .flush() exécute les appels asynchrones
- Utilise la méthodes when et leurs raccourcis : whenGET, whenPOST etc.
- .respond() retourne une réponse fake
- $q : résolu après un $rootScope.$apply()

```
describe('MonService', () => {

    let MonService, $httpBackend

    beforeEach(angular.mock.module('MonApp'))

    beforeEach(angular.mock.inject((_MonService_, _$httpBackend_) => {
        MonService = _MonService_
        $httpBackend = _$httpBackend_
        $httpBackend.whenGET('http://url').respond(200, 'data')
   }))

    it('Should do something', () => {
        const result = MonService.getData()
        $httpBackend.flush()

        expect(result).toBe('data')
    })
})
```

Tester un Filter

- $injector : service qui permet d'instancier un service

```
describe('MonFilter', () => {
    const MonFilter
    beforeEach(angular.mock.module('monApp'))

    beforeEach(angular.mock.inject(($filter) => {
        MonFilter = $filter('MonFilter')
    }))

    it('Should be ok', function() {
        expect( MonFilter(data, param) ).toBe('ok')
    })
})
```

Tester une action

```
import { USER } from './user.actions'

describe('User Actions', () => {

  let userActions

  beforeEach(angular.mock.module('MonApp'))

  beforeEach(angular.mock.inject((_UserActions_) => {
    userActions = _UserActions_
  }))

  it('Should pay', () => {
    const result = userActions.pay()
    expect(result).toEqual({
      type: USER.PAY
    })
  })

})
```

Tester un Reducer

- Appel de la fonction reducer pour le test

```
import { USER } from '../actions/user.actions'
import userReducer from './user.reducer'

describe('User Reducer', () => {

  let result
  const initState = {}

  it('Should pass', () => {
    result = userReducer(initState, {
      type: 'OTHER'
    })
    expect(result).toEqual(initState)
  })

  it('Should pay', () => {
    result = userReducer(initState, {
      type: USER.PAY
    })
    expect(result.payed).toBe(true)
  })

})
```

Tester un Component

- $componentController : instancie le component
- $rootScope.$new() : crée le scope du component
- $scope.$apply() : résout la digest loop (promises, mise à jour de données du scope etc.)

```
describe('UnContainer', () => {

    let scope, component, $ngRedux, unService

    beforeEach(angular.mock.module('monApp'))

    beforeEach(angular.mock.inject(($rootScope, $componentController, _$ngRedux_, _unService_) => {
       $ngRedux = _$ngRedux_
       unService = _unService_
       scope = $rootScope.$new()
       const bindings = null
       component = $componentController('monContainer', { $scope: scope }, bindings)
    }))

    it('Should init, () => {
        spyOn($ngRedux, 'dispatch')
        spyOn(unService, 'methode')

        expect(component).toBeDefined()

        component.$onInit()
        scope.$apply()
        expect($ngRedux.dispatch).toHaveBeenCalled()
        expect(unService.methode).toHaveBeenCalled()
    })
}
```

En conclusion

- Les tests assurent un code de qualité et sans erreur
- Les tests permettent le contrôle des non regressions
- Les tests sont une documentation de spécifications

FIN