

Angular

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering

Logistique



- Horaires
- Déjeuner & pauses
- Autres questions ?



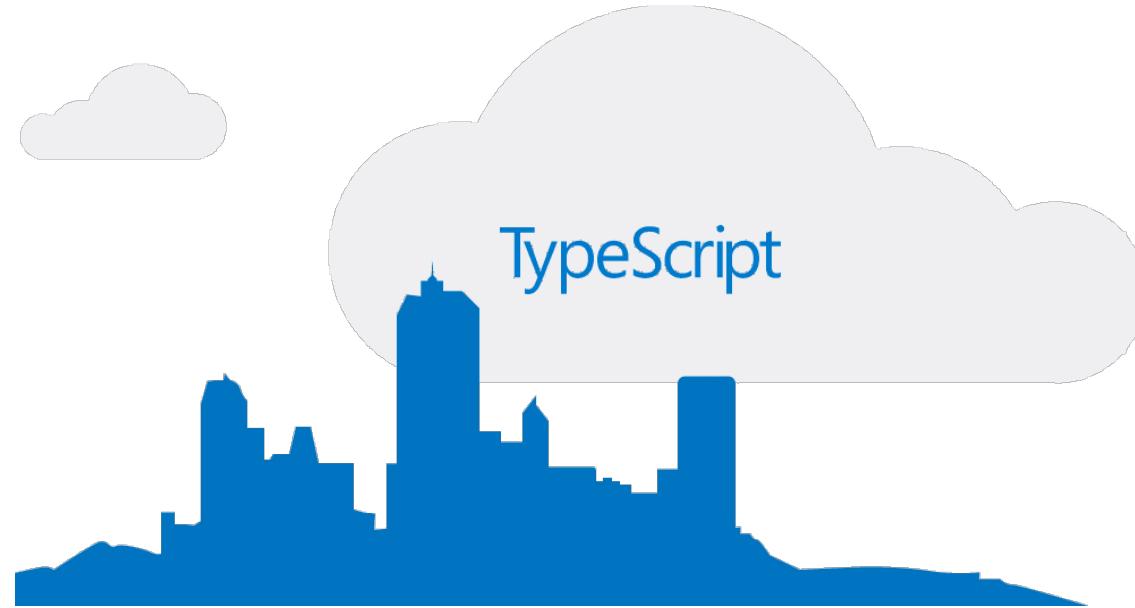
Rappels

Sommaire



- *Rappels*
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering

Introduction

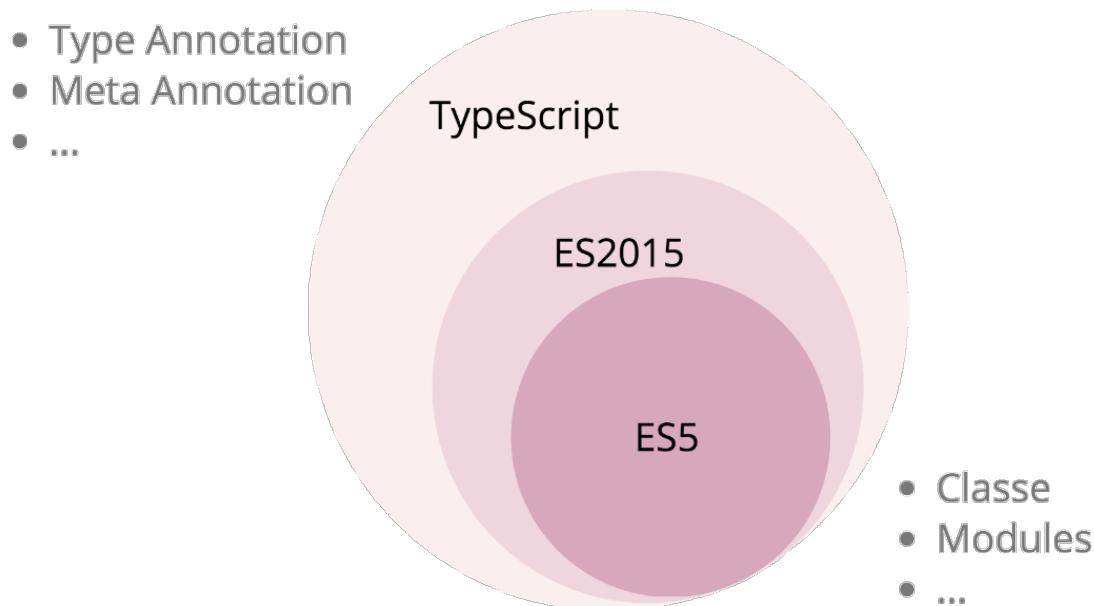


- Langage créé par **Anders Hejlsberg** en 2012
- Projet open-source maintenu par **Microsoft** (Version actuelle **2.1**)
- Influencé par **JavaScript**, **Java** et **C#**
- Alternatives : CoffeeScript, Dart, Haxe ou Flow

Introduction



- Phase de compilation nécessaire pour générer du **JavaScript**
- Ajout de nouvelles fonctionnalités au langage **JavaScript**
- Support d'ES3 / ES5 / ES2015
- Certaines fonctionnalités n'ont aucun impact sur le JavaScript généré
- Tout programme **JavaScript** est un programme **TypeScript**





TypeScript - Fonctionnalités

- Typage
- Génériques
- Classes / Interfaces / Héritage
- Développement modulaire
- Les fichiers de définitions
- Mixins
- Décorateurs
- Fonctionnalités **ES2015**





Types primitifs

- Pour déclarer une variable :

```
var variableName: variableType = value;  
let variableName2: variableType = value;  
const variableName3: variableType = value;
```

- boolean : `let isDone: boolean = false;`
- number : `let height: number = 6;`
- string : `let name: string = 'Carl';`
- array : `let names: string[] = ['Carl', 'Laurent'];`
- any : `let notSure: any = 4;`

Fonctions



- Comme en JavaScript, possibilité de créer des fonctions nommées ou anonymes

```
//Fonction nommée
function namedFunction():void { }

//Fonction anonyme
let variableAnonymousFunction = function(): void { }
```

- Peut retourner une valeur grâce au mot clé **return**
- Accès aux variables définies en dehors du scope de la fonction

```
let externalScope:number = 10;

function add(localArg: number): number { return localArg + externalScope; }
```





Fonctions - Paramètres (Optionels)

- Une fonction peut prendre des paramètres

```
function getFullName(name: string, forename: string) { }
```

- Un paramètre peut être optionnel
 - utilisation du caractère ?
 - ordre de définition très important
 - aucune implication dans le code JavaScript généré
 - si pas défini, le paramètre aura la valeur **undefined**

```
function getFullName(name: string, forename?: string) { }
```



Arrays

- Permet de manipuler un tableau d'objet
- 2 syntaxes pour créer des tableaux
 - Syntaxe Litérale

```
let list: number[] = [1, 2, 3];
```

- Syntaxe utilisant le constructeur **Array**

```
let list: Array<number> = [1, 2, 3];
```

- Ces 2 syntaxes aboutiront au même code JavaScript



Type Enum

- Possibilité de définir un type pour expliciter un ensemble de données numériques

```
enum Music { Rock, Jazz, Blues };  
  
let c: Music = Music.Jazz;
```

- La valeur numérique commence par défaut à 0
- Possibilité de surcharger les valeurs numériques

```
enum Music { Rock = 2, Jazz = 4, Blues = 8 };  
  
let c: Music = Music.Jazz;
```

- Récupération de la chaîne de caractères associée à la valeur numérique

```
let style: string = Music[4]; //Jazz
```



Classes



- Système de **classes** et **interfaces** similaire à la programmation orientée objet
- Le code javascript généré utilisera le système de **prototype**
- Possibilité de définir un constructeur, des méthodes et des propriétés
- Propriétés/méthodes accessibles via l'objet **this**

```
class Person {  
    constructor() {}  
}  
  
let person = new Person();
```

Classes - Méthodes



- Méthodes ajoutées au **prototype** de l'objet
- Version TypeScript

```
class Person {  
    constructor() {}  
  
    sayHello(message: string) { }  
}
```

- Version JavaScript

```
var Person = (function () {  
    function Person() { }  
    Person.prototype.sayHello = function (message) { };  
    return Person;  
}());
```

Classes - Propriétés



- Trois scopes disponibles : **public**, **private** et **protected**
- Utilise le scope **public** par défaut
- Scope **protected** apparu en TypeScript 1.3
- Propriétés ajoutées sur l'objet en cours d'instanciation (**this**)
- Possibilité de définir des propriétés statiques (**static**)
 - Tous les types supportés : types primitifs, fonctions, ...
 - Propriété ajoutée au constructeur de l'objet



Classes - Propriétés

- Version TypeScript

```
class Person {  
    firstName: string;  
    constructor(firstName: string){  
        this.firstName = firstName;  
    }  
}
```

- Version JavaScript

```
var Person = (function () {  
    function Person(firstName) {  
        this.firstName = firstName;  
    }  
    return Person;  
})();
```



Classes - Propriétés

- Seconde version pour initialiser des propriétés
- Version TypeScript

```
class Person {  
    constructor(public firstName: string) {}  
}
```

- Version JavaScript

```
var Person = (function () {  
    function Person(firstName) {  
        this.firstName = firstName;  
    }  
    return Person;  
})();
```

Classes - Accesseurs



- Possibilité de définir des accesseurs pour accéder à une propriété
- Utiliser les mots clé **get** et **set**
- Attention à l'espacement apres les mots clé
- Nécessité de générer du code JavaScript compatible ES5
- Le code JavaScript généré utilisera **Object.defineProperty**

```
class Person {  
    private _secret: string;  
    get secret(): string{  
        return this._secret.toLowerCase();  
    }  
    set secret(value: string) {  
        this._secret = value;  
    }  
}  
  
let person = new Person();  
person.secret = 'Test';  
console.log(person.secret); // => 'test'
```

Classes - Héritage



- Système d'héritage entre classes via le mot clé **extends**
- Si constructeur non défini, exécute celui de la classe parente
- Possibilité d'appeler l'implémentation de la classe parente via **super**
- Accès aux propriétés de la classe parente si **public** ou **protected**

```
class Person {  
    constructor() {}  
    speak() {}  
}  
  
class Child extends Person {  
    constructor() { super() }  
    speak() { super.speak(); }  
}
```

Interfaces



- Utilisées par le compilateur pour vérifier la cohérence des différents objets
- Aucun impact sur le JavaScript généré
- Système d'héritage entre interfaces
- Plusieurs cas d'utilisation possible
 - Vérification des paramètres d'une fonction
 - Vérification de la signature d'une fonction
 - Vérification de l'implémentation d'une classe



Interfaces - Implémentation d'une classe

- Cas d'utilisation le plus connu des interfaces
- Vérification de l'implémentation d'une classe
- Erreur de compilation tant que la classe ne respecte pas le contrat défini par l'interface

```
interface Musician {  
    play(): void;  
}  
  
class TrumpetPlay implements Musician {  
    play() {}  
}
```



Génériques

- Fonctionnalité permettant de créer des composants réutilisables
- Inspiration des génériques disponibles en Java ou C#
- Nécessité de définir un (ou plusieurs) paramètre(s) de type sur la fonction/variable/classe/interface générique

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
identity(5).toFixed(2); // Correct  
  
identity('hello').toFixed(2); // Incorrect  
  
identity(true);
```



Génériques

- Possibilité de définir une classe générique
- Définition d'une liste de paramètres de type de manière globale

```
class Log<T> {
    log(value: T) {
        console.log(value);
    }
}

let numericLog = new Log<number>();

numericLog.log(5); // Correct
numericLog.log('hello'); // Incorrect
```



- Node inclut un système de gestion des paquets : **npm**
- Il existe pratiquement depuis la création de Node.js
- C'est un canal important pour la diffusion des modules



npm install



- **npm** est un outil en ligne de commande (écrit avec Node.js)
- Il permet de télécharger les modules disponibles sur [npmjs.org](https://www.npmjs.org)
- Les commandes les plus courantes :
 - **install** : télécharge le module et le place dans le répertoire courant dans `./node_modules`
 - **install -g** : installation globale, le module est placé dans le répertoire d'installation de Node.js
Permet de rendre accessible des commandes
 - **update** : met à jour un module déjà installé
 - **remove** : supprime le module du projet

npm init



- **npm** gère également la description du projet
- Un module Node.js est un (ou plusieurs) script(s)
- Le fichier de configuration se nomme **package.json**
- **npm** permet également de manipuler le module courant
 - **init** : initialise un fichier **package.json**
 - **docs** : génère la documentation du module en cours
 - **install --save** ou **--save-dev** :
Comme install mais référence automatiquement la dépendance dans le **package.json**





package.json

- **npm** se base sur un fichier descripteur du projet
- **package.json** décrit précisément le module
- On y trouve différents types d'information
 - Identification
 - **name** : l'identifiant du module (unique, url safe)
 - **version** : doit respecter **node-semver**
 - Description : **description**, **authors**, ...
 - Dépendances : **dependencies**, **devDependencies**, ...
 - Cycle de vie : scripts **main**, **test**, ...



package.json : dépendances



- **dependencies**

La liste des dépendances nécessaires à l'exécution

- **devDependencies**

Les dépendances pour les développements (build, test...)

- **peerDependencies**

Les dépendances nécessaires au bon fonctionnement du module, mais pas installées lors d'un `npm install`



package.json : versions

- Les modules doivent suivre la norme **semver**
 - Structure : **MAJOR.MINOR.PATCH**
 - **MAJOR** : Changements d'API incompatibles
 - **MINOR** : Ajout de fonctionnalité rétro-compatible
 - **PATCH** : Correction de bugs
- Pour spécifier la version d'une dépendance
 - **version** : doit être exactement cette version
 - **~, ^** : approximativement, compatible
 - **major.minor.x** : **x** fait office de joker
 - Et bien d'autres : **>**, **<**, **>=**, **min-max...**



Publier un module npm

- Il est bien sûr conseillé de suivre toutes les bonnes pratiques
 - Utiliser la numérotation recommandée
 - Avoir des tests unitaires
 - Avoir un minimum d'informations dans le `package.json`
- Il n'y a pas d'autorité de validation
- Il faut par contre trouver un nom disponible
- La suite nécessite seulement la commande `npm`
 - `npm adduser` : enregistrer son compte
 - `npm publish` : uploader un module sur `npmjs.org`



Présentation

Sommaire



- Rappels
- *Présentation*
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering



Présentation



- Framework créé par **Google** et annoncé en 2014
- Réécriture total du framework
- Reprend certains concepts d'**AngularJS**
- 1e version **beta** annoncée le 23/10/2014
- Version officielle sortie en 2016
- Programmation orientée **Composant**
- Framework conçu pour être plus performant et optimisé pour les mobiles
- <http://angular.io/>



Points négatifs d'AngularJS

- Différences entre les directives et `ngController`
- Two-way data-binding source de problèmes de performance
- Hiérarchie des scopes
- Pas de server-side rendering
- Plusieurs syntaxes pour créer des services
- API des directives trop complexe
- API mal conçue nécessitant l'utilisant de fix (`ngModelOptions`)



Points négatifs d'AngularJS - directive



- API des directives trop complexe

```
app.directive('MyDirective', function(){
  return {
    restrict: 'AE',
    require: '?^ngModel',
    scope: { variable: '@' },
    controller: function(...) {},
    link: function(...) { ... }
  }
});
```

- Version **Angular** :

```
import { Component, Input} from '@angular/core'
@Component({
  selector: 'my-directive'
})
export class MyDirective {
  @Input() variable:string;
}
```



Points négatifs d'AngularJS - service

- API pour créer des services en **AngularJS**

```
//provider, factory, constant et value
app.service('Service', function(){
  let vm = this;
  vm.myMethod = function(){
    }
});
```

- Version Angular

```
@Injectable()
export class Service {

  myMethod(){
    }
}
```



Angular - Points Positifs

- Création d'application modulaire
- Utilisable avec plusieurs langages de programmation : **ES5**, **ES2015 (ES6)**, **TypeScript** et **Dart**
- API plus simple que **AngularJS**
- Seuls trois types d'éléments seront utilisés : **directive**, **pipe** et les **services**
- Basé sur des standards : **Web Component**, **Decorator**, **ES2015**, **ES7**
- Nouvelle syntaxe utilisée dans les templates
- Performance de l'API **Change Detection**
- Le Projet **Universal**
- Librairie pour commencer la migration : **ngUpgrade**
- Collaboration avec Microsoft et Ember



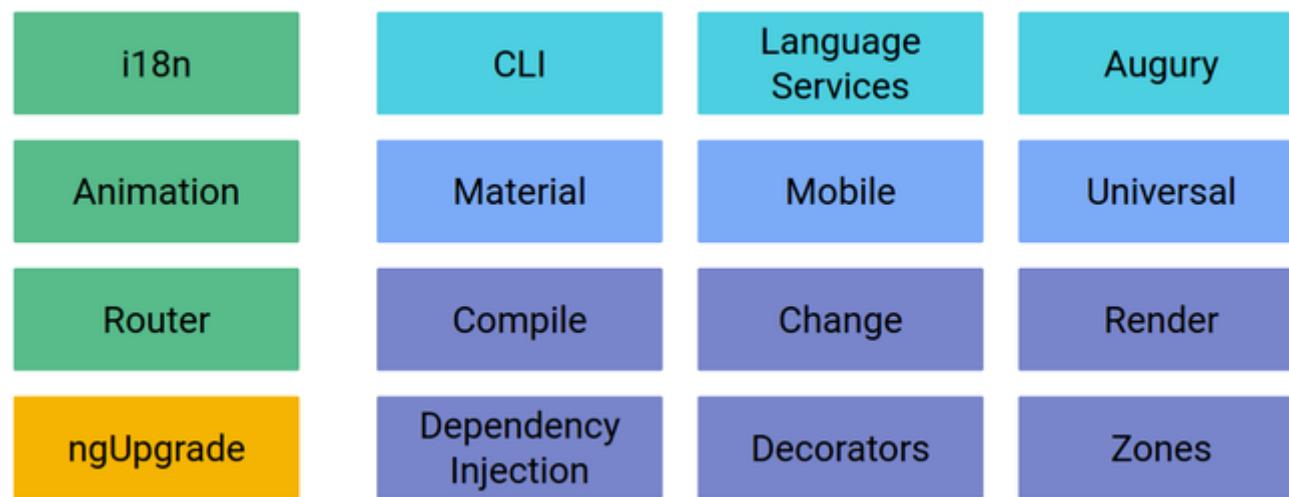
Angular - Points Négatifs

- Nouvelle phase d'apprentissage du framework
- Faible écosystème pour l'instant
- Application AngularJS incompatible avec cette nouvelle version
 - ngUpgrade permet de rendre compatible les directives, composants et services
- De nouveaux concepts à apprendre :
 - Zone
 - Observable
 - WebPack...



Angular = Une Plateforme

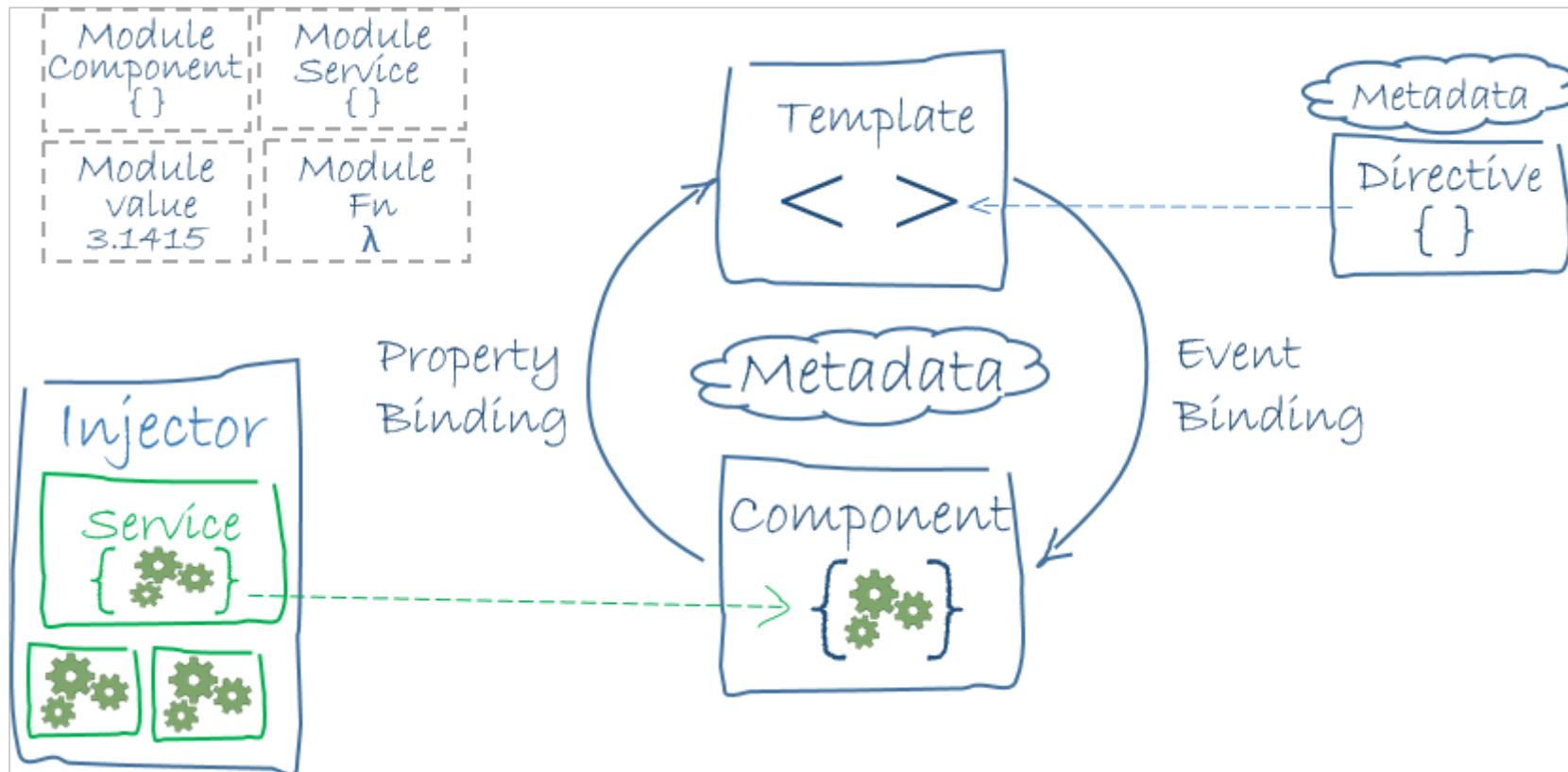
- Angular n'est pas qu'un simple framework
- Intégration Mobile
- Outilage pour faciliter la phase de développement





Architecture

- Architecture d'une application Angular





Architecture

- Modules : regroupement d'un ensemble de fonctionnalités sous un même namespace
- Library Modules (**barrels**): `@angular/core`, `@angular/http`...
- Les composants : Elément graphique composé d'un template et d'une classe
- Métdata : Moyen d'indiquer à Angular comment utiliser la classe
- Directives : composants sans template (**ngFor**, **ngIf**, ...)
- Services : Code métier implémenté dans des classes qui seront injectées dans les différents composants
- Pipe : Elément permettant de formatter une donnée (équivalent au **filter** d'AngularJS)



Architecture - Exemple complet

- Exemple complet utilisant les différentes briques d'une application Angular

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';

@Component({
  selector: 'app',
  template: '{{value | uppercase}}'
})
export class MyComponent{
  value:string;
  constructor(http:Http){
  }
}
```



Démarrer une application Angular

Sommaire



- Rappels
- Présentation
- *Démarrer une application Angular*
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering





Commencer un nouveau projet

- Gestion des dépendances via **NPM**
 - les différents modules **Angular** : `@angular/common`, `@angular/core`...
 - Webpack : gestion des modules
 - RxJS : gestion de l'asynchrone

```
npm init
npm install --save @angular/common @angular/core rxjs ...
```

- Initialisation et Configuration d'un projet **TypeScript**
- Configuration du système de gestion des modules (**Webpack**)
- Installation des fichiers de définition (**typings**, **npm** pour **TypeScript 2.0**)
- Nécessité d'utiliser un serveur Web
 - **Apache**, **serve**, **live-server**...



Commencer un nouveau projet

- Création du composant principal
 - définir le sélecteur nécessaire pour utiliser le composant
 - écrire le template
 - implémenter la classe **TypeScript**

```
import { Component } from '@angular/core'

@Component({
  selector: 'app',
  template: `<p>Hello</p>`
})
export class AppComponent { ... }
```

Commencer un nouveau projet



- Création d'un module Angular

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}

platformBrowserDynamic().bootstrapModule(AppModule);
```



Angular-CLI

- Projet en cours de développement
- Basé sur le projet **Ember CLI**
- Permet de créer le squelette d'une application
 - TypeScript, WebPack, Karma, Protractor, Préprocesseurs CSS ...
- Projet disponible sur **NPM**

```
npm install -g @angular/cli
```

- Plusieurs commandes disponibles

```
ng new Application
ng build (--dev / --prod)
ng serve

ng generate component Product (--inline-template) // => class ProductComponent{
...
}
ng generate pipe Uppercase
ng generate service User
ng generate directive myNgIf
```



Angular-CLI

- D'autres commandes disponibles :

- `ng test`
- `ng e2e`
- `ng lint`

WebPack



- Gestionnaire de modules
- Supporte les différents systèmes de modules (**CommonJS**, **AMD**, **ES2015**, ...)
- Disponible sur **NPM** : `npm install -g webpack`
- Construit un graphe de toutes les dépendances de votre application
- configuration via un fichier de configuration **JavaScript** (`webpack.config.js`)
 - loaders : **ES2015**, **TypeScript**, **CSS**, ...
 - preloaders: **JSHint**, ...
 - plugins: **Uglify**, ...



WebPack - Premier exemple

- Première utilisation de **WebPack**

```
//app.js
document.write('welcome to my app');
console.log('app loaded');
```

- Exécution de **WebPack** pour générer un fichier **bundle.js**

```
webpack ./app.js bundle.js
```

- Import de votre fichier **bundle.js** dans votre **index.html**

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



WebPack

- Version avec un fichier de configuration
 - solution à privilégier pour que tous les développeurs utilisent la même configuration

```
module.exports = {  
  entry: "./app.js",  
  output: {  
    filename: "bundle.js"  
  }  
}
```

webpack



WebPack - Configuration

- Possibilité de générer plusieurs fichiers
 - Utilisation du placeholder `[name]`

```
entry: {  
  app: 'src/app.ts',  
  vendor: 'src/vendor.ts'  
},  
output: {  
  filename: '[name].js'  
}
```

- Création d'un fichier `vendor.ts` important toutes librairies utilisées

```
// Angular  
import '@angular/core';  
import '@angular/common';  
import '@angular/http';  
import '@angular/router';  
// RxJS  
import 'rxjs';
```



WebPack - Configuration

- Possibilité de regénérer le `bundle.js` à chaque modification des sources (`watch`)
- Serveur web disponible (`webpack-dev-server`)
 - **Hot Reloading**
 - Mode **Watch** activée
 - Génération du fichier `bundle.js` en mémoire

WebPack - Les Loaders

- Permet d'indiquer à WebPack comment prendre en compte un fichier
- Plusieurs **loaders** existent : **ECMAScript2015**, **TypeScript**, **CoffeeScript**, **Style**, ...

```
entry: {  
  app: 'src/app.ts',  
  vendor: 'src/vendor.ts'  
},  
resolve: {  
  extensions: ['', '.js', '.ts']  
},  
module: {  
  loaders: [{  
    test: /\.ts$/,  
    loaders: ['ts']  
  }]  
},  
output: {  
  filename: '[name].js'  
}
```

WebPack - Les Plugins

- Permet d'ajouter des fonctionnalités à votre workflow de build

```
entry: {  
  app: 'src/app.ts',  
  vendor: 'src/vendor.ts'  
},  
resolve: {  
  extensions: ['', '.js', '.ts']  
},  
module: {  
  loaders: [{  
    test: /\.ts$/,  
    loaders: ['ts']  
  }]  
},  
plugins: [  
  new webpack.optimize.CommonsChunkPlugin({name: ['app', 'vendor']}),  
  
  new HtmlWebpackPlugin({template: 'src/index.html'})  
]  
output: {  
  filename: '[name].js'  
}
```

WebPack - Autres outils

- L'optimisation d'une application **Angular** peut être découpée en 4 phases:
 - Offline compilation : **ngc**
 - Inline modules : **WebPack**
 - Tree-Shaking : **Rollup**
 - Minification : **Uglify**





Les Tests

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- *Tests*
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering

Concepts



- Dans la documentation **Jasmine** est utilisé comme framework de tests
- **Angular** peut être également testé avec d'autres frameworks
- Karma propose d'exécuter facilement les tests
 - Il a été développé par l'équipe d'**AngularJS**, il est donc mis en avant
 - Il n'est pour autant ni indispensable ni lié à **Angular**
- **Jasmine** et **Karma** sont intégrés dans une application générée par **angular-cli**.

Tests - Utilisation de Jasmine



- Framework de Tests : <http://jasmine.github.io/>
- Aucune dépendance vers d'autres frameworks
- Ne nécessite pas d'élément du **DOM**
- Essayer **Jasmine** en ligne : <http://tryjasmine.com/>



Tests - Structure d'un test Jasmine

- Méthodes **describe** et **it** pour décrire la suite de tests
- Système de **matchers** : **toBe**, **toBeUndefined**, **toBeTruthy**, **toThrow**,
...
- Possibilité d'utiliser les librairies **Chai** ou **SinonJS**

```
describe('True value:', function() {
  it('true should be equal to true', function() {
    expect(true).toBe(true);

  });
});
```





Tests - Structure d'un test Jasmine

- Méthodes `beforeEach`, `afterEach`, `beforeAll`, `afterAll`
- Exécution d'une fonction avant ou après chaque test

```
describe('True value:', function() {
  let value;

  beforeEach(function(){
    value = true;
  });

  it('true should be equal to true', function() {
    expect(value).toBe(true);
  });
});
```





Tests - Structure d'un test Jasmine

- Possibilité de définir des **Spies** grâce à la méthode `spyOn`
- Vérifier l'exécution de la méthode **espionnée**
 - `toHaveBeenCalled`, `toHaveBeenCalledWith`
 - `and.callThrough`, `and.returnValue`, `and.callFake...`
 - `Spy.calls`

```
describe('Service objet:', function() {  
  
  it('checkout method should be called', function() {  
    spyOn(service, 'foo');  
    service.foo();  
    expect(service.foo).toHaveBeenCalled();  
  });  
});
```





Tests - Tests TypeScript

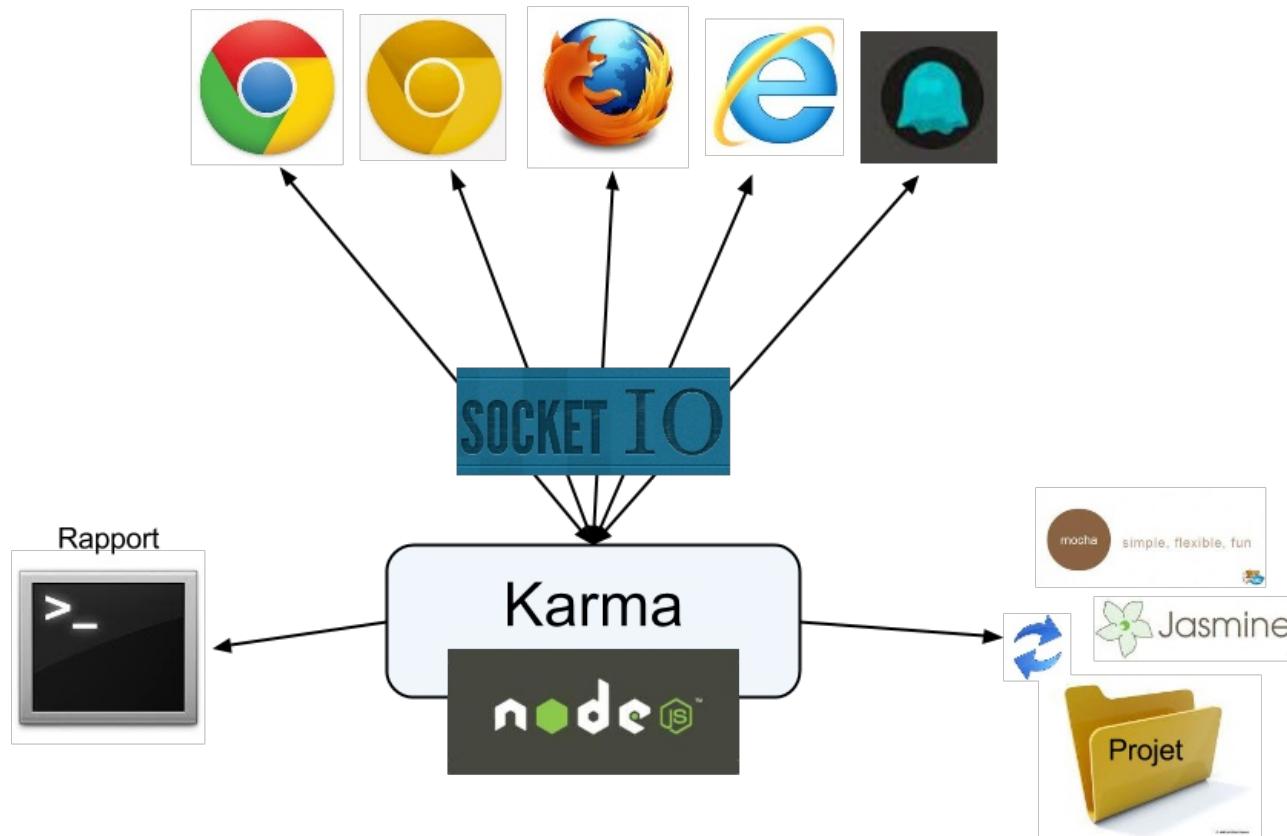
- Possibilité d'écrire des tests en **TypeScript**

```
class True {  
    returnTrue(){ return true; }  
}  
  
describe('True object:', () => {  
    describe('returnTrue method:', () => {  
        it('should return true', () => {  
            let trueObject:True = new True();  
            expect(trueObject.returnTrue()).toBe(true);  
        });  
    });  
});
```

Karma



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





Tests - Automatisation de l'exécution des tests

- Configuration automatiquement réalisée par `angular-cli`
- Les fichiers de test sont automatiquement créés lors de la création d'un `Composant/Service/Pipe` via `angular-cli`
- Ils se trouvent dans le même répertoire que l'élément à tester : `mon-service.spec.ts`
- Exécution des tests :

```
ng test
```







Template & Composants

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- *Template & Composants*
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering



Syntaxe des templates

- Système d'interpolation grâce à la syntaxe `{{ expression }}`
- L'expression peut être :
 - une chaîne de caractère
 - la valeur d'une variable
 - la valeur rentrée d'une fonction
- Cette expression sera convertie en `string` avant son affichage
- Une expression ne doit pas modifier l'état de l'application

Les propriétés



- Possibilité de définir une valeur pour une propriété
- Différent d'AngularJS, où nous utilisons les attributs **HTML**
- Attention à la différence entre attribut et propriété
- Un attribut est statique contrairement à une propriété
- Syntaxe identique pour les propriétés des éléments **HTML**, des composants et des directives
- Utilisation de la syntaxe `[property-name] = "expression"`

```
<button [disabled]="isUnchanged">Save</button>
<button bind-disabled="isUnchanged">Save</button>
<button data-bind-disabled="isUnchanged">Save</button>
<hero-detail [hero]="currentHero"></hero-detail>

<div [class.special]="isSpecial">Special</div>
<button [style.color] = "isSpecial ? 'red' : 'green'">
```



Les propriétés



- Pour les attributs n'ayant pas d'équivalence dans l'API DOM
 - utilisation du **Attribute Binding**
- A utiliser pour **aria**, **colspan**, **svg** par exemple
- Utilisation de la syntaxe `[attr.attribute-name] = "expression"`

```
<td [colspan]="dynamicColspan">help</td>

<!-- Template parse errors:
Can't bind to 'colspan' since it isn't a known native property-->

<td [attr.colspan]="dynamicColspan">help</td>
```

Les évènements



- Permet d'associer une expression **Angular** à un évènement
 - défini dans la spécification HTML : **click, blur, ...**
 - créé spécialement pour l'application (avec une sémantique précise)
- Les méthodes et propriétés utilisées doivent être définies dans la classe associée
- Utilisation de la syntaxe (**event-name**) = "expression"

```
<button (click)="myMethod()"></button>
<hero-detail (deleted)="onHeroDeleted()"></hero-detail>

<button on-click="myMethod()"></button>
<button data-on-click="myMethod()"></button>
```

Les évènements



- **Angular** va créer un handler pour chaque évènement
- Possibilité de récupérer le contexte de l'évènement, et de potentielles données via l'objet **\$event**
- Cet objet peut être utilisé dans l'expression **Angular**
- Tous les évènements natifs sont propagés vers les éléments parents
 - nous devons retourner une valeur **false** pour stopper cette propagation
- Les évènements **EventEmitter** ne se propagent pas.

```
<input [value]="currentHero.firstName"  
       (input)="currentHero.firstName=$event.target.value"/>
```





Syntaxe "Banana in the Box"

- Le **2-way data-binding** est désactivé par défaut
- Pour synchroniser un champ de formulaire avec une variable, nécessité d'utiliser cette syntaxe

```
<input [value]="currentHero.firstName"  
       (input)="currentHero.firstName=$event.target.value"/>
```

- **Angular** fournit du sucre syntaxique afin d'éviter cette redondance de code
- Première solution :

```
<input  
   [ngModel]="currentHero.firstName"  
   (ngModelChange)="currentHero.firstName=$event"/>
```

- Deuxième solution :

```
<input [(ngModel)]="currentHero.firstName"/>
```



Les Composants



- Les composants sont des éléments graphiques
- Utilisation de l'annotation `@Component`
- L'élément HTML est sélectionné par une sélecteur `CSS`
- Utiliser un préfixe pour les noms de vos directives pour éviter les conflits
- `@Component` fournit notamment les paramètres `template`, `templateUrl`, `styles`, `styleUrls` et `encapsulation`

```
import { Input, Component } from '@angular/core'
import { Product } from './model/Product'

@Component({
  selector: 'product-detail',
  template: `
    <article>
      <h1>Product 1</h1>
      <button>Add</button>
    </article>
  `
})
export class ProductComponent {}
```

Les Composants - Les paramètres



- Un composant pourra être paramétrable
- Déclaration d'une variable de classe annotée `@Input`
- Le nom de la variable de classe qui sera utilisée dans le template

```
import { Input, Component } from '@angular/core'
import { Product } from './model/Product'

@Component({
  selector: 'product-detail',
  template: `
    <article>
      <h1>{{ product.title }}</h1>
      <button>Add</button>
    </article>
  `
})
export class ProductComponent {
  @Input() product:Product;
}
```



Les Composants - Les évènements



- De la même façon, une directive pourra émettre un évènement
- Déclaration d'une variable de classe annotée `@Output` de type `EventEmitter`
- Le nom de la variable correspond au nom de l'évènement qui sera utilisé dans l'HTML

```
@Component({
  selector: 'product-detail',
  template: `
    <article>
      <h1>{{ product.title }}</h1>
      <button (click)="clickHandler()">Add</button>
    </article>`
})
export class ProductComponent {
  @Input() product:Product;
  @Output() addToBasket = new EventEmitter<Product>();

  clickHandler(){ this.addToBasket.emit(this.product); }
}
```

Les Composants - Enregistrement



- Les composants externes nécessaires à votre applications doivent :
 - être définis dans un module importé par votre application (`ngModule`)
 - être définis dans la propriété `declarations` du décorateur `ngModule` de votre application

```
import { NgModule, ApplicationRef } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    CommonModule,
    FormsModule
  ]
})
export class AppModule {}
```

Les Composants - Projection



- Permet d'insérer le contenu enfant défini lors de l'utilisation du composant
- Correspond à la directive **ngTransclude** en **AngularJS**
- Possibilité d'avoir plusieurs points d'insertion (utilisation de la propriété **select**)
- La propriété **select** accepte comme valeur un sélecteur **CSS** : classe, élément et attribut

```
//<post><h2>Title</h2><p>Content</p></post>
import {Component} from '@angular/core';
@Component({
  selector: 'post',
  template: `<article>
    <header><ng-content select="h2"></ng-content></header>
    <section><ng-content select="p"></ng-content></section>
  </article>`
})
export class PostComponent { }
```



Les Composants - Tests

- Nécessité de configurer l'objet **TestBed** via sa méthode **configureTestingModule** :

```
TestBed.configureTestingModule({
  declarations: [
    TitleComponent
  ],
  imports: [
    // HttpModule, FormsModule, etc.
  ],
  providers: [
    // TitleService,
    // { provide: TitleService, useClass: TitleServiceMock }
  ]
});
```

- La méthode **createComponent** de l'objet **TestBed** retourne un objet de type **ComponentFixture** qui est une représentation du composant

Les Composants - Tests



- Un objet de type **ComponentFixture** propose deux propriétés intéressantes :
 - **componentInstance** : l'instance **JavaScript** du composant
 - **nativeElement** : l'élément **HTML**
- Pour exécuter l'API **Change Detection**, utilisation de la méthode **detectChanges**

```
@Component({
  selector: 'title', template: '<h1>{{title}}</h1>'
})
export class TitleCmp {
  @Input() title: string;
}
```

Les Composants - Tests



- Test Unitaire :

```
import { TestBed, async } from '@angular/core/testing';

import { TitleComponent } from './title.component';
describe('TitleComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TitleComponent]
    });
  });

  it('should have a title', async(() => {
    let fixture = TestBed.createComponent(TitleComponent);
    let instance = fixture.componentInstance;
    instance.title = 'Hello World';
    fixture.detectChanges();

    let element = fixture.nativeElement;
    expect(element.querySelector('h1').textContent).toBe('Hello World');
  }));
});
```





Directives

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- *Directives*
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering

Les Directives



- Portion de code permettant de définir l'apparence ou le fonctionnement d'un élément HTML
- Création de directive personnalisée avec l'annotation `@Directive`
- Pour faire de la manipulation de DOM, toujours utiliser le service `Renderer`
- Peuvent accepter des paramètres (`Input`) et émettre des évènements (`Output`)

```
//<p myHighlight>Highlight me!</p>
import { Directive, ElementRef, Renderer } from '@angular/core';
@Directive({
  selector: '[myHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef, renderer: Renderer) {
    //el.nativeElement.style.backgroundColor = 'yellow';
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'yellow');
  }
}
```

Les Directives - Action utilisateur



- Possibilité d'écouter les évènements de l'élément sur lequel est placé la directive
- Utilisation de la propriété **host** de l'annotation **@Directive**
- L'ajout d'handler programmatiquement est à éviter pour des problèmes de mémoire
- Possibilité d'utiliser les décorateurs **HostListener** et **HostBinding**

```
import { Directive, HostListener, HostBinding } from '@angular/core';

@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
    @HostBinding('style.backgroundColor') color = 'red';

    constructor() { ... }

    @HostListener('mouseenter') onMouseEnter() { this.color = 'blue'; }

    @HostListener('mouseleave') onMouseLeave() { this.color = 'red'; }
}
```

Les directives Angular



- **Angular** fournit une trentaine de directives :
 - Manipulation de DOM
 - Gestion des formulaires
 - Routeur
- Importer le module correspondant pour les utiliser :
 - **CommonModule**
 - **FormsModule**
 - **RouterModule**

Les directives Angular - ngStyle



- Directive permettant d'ajouter des définitions CSS
- Nécessité de spécifier un objet JSON en tant que paramètre

```
import {Component} from '@angular/core';

@Component({
  selector: 'ngStyle-example',
  template: `
    <h1 [ngStyle]="'font-size': size">
      Title
    </h1>

    <label>Size:
      <input type="text" [value]="size" (change)="size =
$event.target.value">
    </label>
  `
})
export class NgStyleExample {
  size = '20px';
}
```

Les directives Angular - ngClass



- La directive **ngClass** ajoute ou enlève des classes CSS.
- Trois syntaxes coexistent
 - `[ngClass]="'class class1'"`
 - `[ngClass]=["class", "class1"]"`
 - `[ngClass]="{'class': isClass, 'class1': isClass1}"`
- La 3e syntaxe est la plus courante :
 - permet de garder la déclaration CSS dans les templates

Les directives Angular - ngClass



- Exemple d'utilisation de la directive `ngClass`

```
import {Component} from '@angular/core';

@Component({
  selector: 'toggle-button',
  template: `
    <div class="button" [ngClass]="{'disabled': isEnabled}"></div>
    <button (click)="toggle(!isEnabled)">Click me!</button>`,
  styles: [
    '.disabled {
      ...
    }
  ]
})
class ToggleButton {
  isEnabled = false;

  toggle(newState) { this.isEnabled = newState; }
}
```

Les directives Angular - ngFor



- Permet de dupliquer un template pour chaque élément d'une collection
- Correspond à la directive **ngRepeat** en **AngularJS**
- Définition des éléments HTML à dupliquer dans un élément **<template>**
- Utilisation de la propriété **ngForOf** pour définir l'expression permettant l'itération
- Sauvegarde de la valeur en cours dans des variables de rendu (prefixées par **let-**)
- Angular met à disposition cinq données supplémentaires : **index**, **first**, **last**, **even** et **odd**

```
<template ngFor let-item [ngForOf]="items" let-i="index"><li>...</li></template>
```

- Seconde syntaxe disponible (également pour **ngIf** et **ngSwitch**)

```
<li *ngFor="let item of items; let i = index"> {{ item.label }} </li>
```





Les directives Angular - `ngIf`

- Ajout / Suppression d'elements HTML en fonction d'une condition
- Si l'expression retourne `true` le template sera inséré

```
<div *ngIf="condition">...</div>
<template [ngIf]="condition"><div>...</div></template>
```

- Pas de directives `ngShow` et `ngHide`
- Utilisation de la propriété `hidden` (nécessite des polyfills)

```
<div [hidden]="condition">...</div>
```





Les directives Angular - ngSwitch

- Ajout / Suppression d'elements HTML en fonction d'une condition
- Trois directives disponibles :
 - **ngSwitch** : élément container
 - **ngSwitchCase** : élément à utiliser pour chaque valeur possible
 - **ngSwitchDefault** : pour définir un template pour une valeur par défaut

```
<div [ngSwitch]="value">
  <p *ngSwitchCase="'init'">increment to start</p>
  <p *ngSwitchCase="0">0, increment again</p>
  <p *ngSwitchCase="1">1, stop incrementing</p>
  <p *ngSwitchDefault>&gt; 1, STOP!</p>
</div>
```





Injection de Dépendances

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- *Injection de Dépendances*
- Pipes
- Service HTTP
- Router
- Formulaires
- Server-side Rendering

Injecteurs



- Élément utilisé pour injecter les services
- Possibilité d'un injecteur par composant contrairement à **AngularJS** (un unique injecteur global)
- Les composants héritent de l'injecteur de leur parent
- Nécessité de configurer les injecteurs
 - de manière globale via le module principal **@NgModule**
 - de manière locale via **@Component**
- Les services sont injectés via la constructeur du parent et sont des singulaires ***au sein du même injecteur***



Configuration globale de l'Injecteur

- `@NgModule` a un paramètre **providers**: un tableau de **providers**
- Les **providers** définis dans un `NgModule` sont injectables partout dans l'application

```
// fichier application.component.ts
import { UserService } from './user.service'

@Component({ ... })
export class AppComponent {
    constructor(userService: UserService){
        console.log(userService.getUser());
    }
}

// fichier app.module.ts
import { UserService } from './services/user.service';

@NgModule({
    providers: [ UserService ],
})
export class AppModule { }
```



Configuration locale de l'Injecteur

- Possibilité de définir une propriété **providers** dans l'annotation **@Component**
- Même syntaxe que la configuration globale
- Les **providers** définis dans un **Component** sont injectables dans ce component et ses fils

```
// fichier application.component.ts
import { UserService } from './user.service'

@Component({
  providers: [ UserService ]
})
export class AppComponent {
  constructor(userService: UserService){
    console.log(userService.getUser());
  }
}
```



Dépendances des services



- Nécessité d'ajouter l'annotation `@Injectable`
- Utilisée pour que **Angular** puisse générer les métadatas nécessaires pour l'injection de dépendances
- Inutile pour les composants, car nous utilisons déjà `@Component`

```
import { Injectable } from '@angular/core';
import { Logger } from './logger-service';

@Injectable()
export class UserService {

    constructor(public logger:Logger){

    }
    myMethod(){ ... }

}
```



Configurer les providers

- Plusieurs syntaxes existent pour définir les providers
- L'identifiant du provider peut être un objet, une chaîne de caractères ou un **InjectionToken**

```
export function serverConfigFactory(appService: AppService){  
    return appService.getConfig();  
}  
  
@NgModule({  
    providers: [  
        UserService,  
        { provide: LoginService, useClass: LoginService },  
        {  
            provide: ServerConfig,  
            useFactory: serverConfigFactory  
            deps: [AppService]  
        }  
    ]  
})  
export class AppModule { }
```



Configurer les providers

- Lorsque nous avons des objets à injecter, et non des classes
- Possibilité de définir une chaîne de caractère comme identifiant
- Utilisation de l'objet **InjectionToken** de préférence
- Nécessité d'utiliser l'annotation **Inject** pour injecter ce genre de service

```
let apiUrl: string = 'api.heroes.com';
let env: string = 'dev';

@NgModule({
  providers: [{provide: 'apiUrl', useValue:apiUrl},{provide: 'env',
useValue:env}],
})
export class AppModule { }

class AppComponent {
  constructor(@Inject('apiUrl') api:string) { ... }
}
```

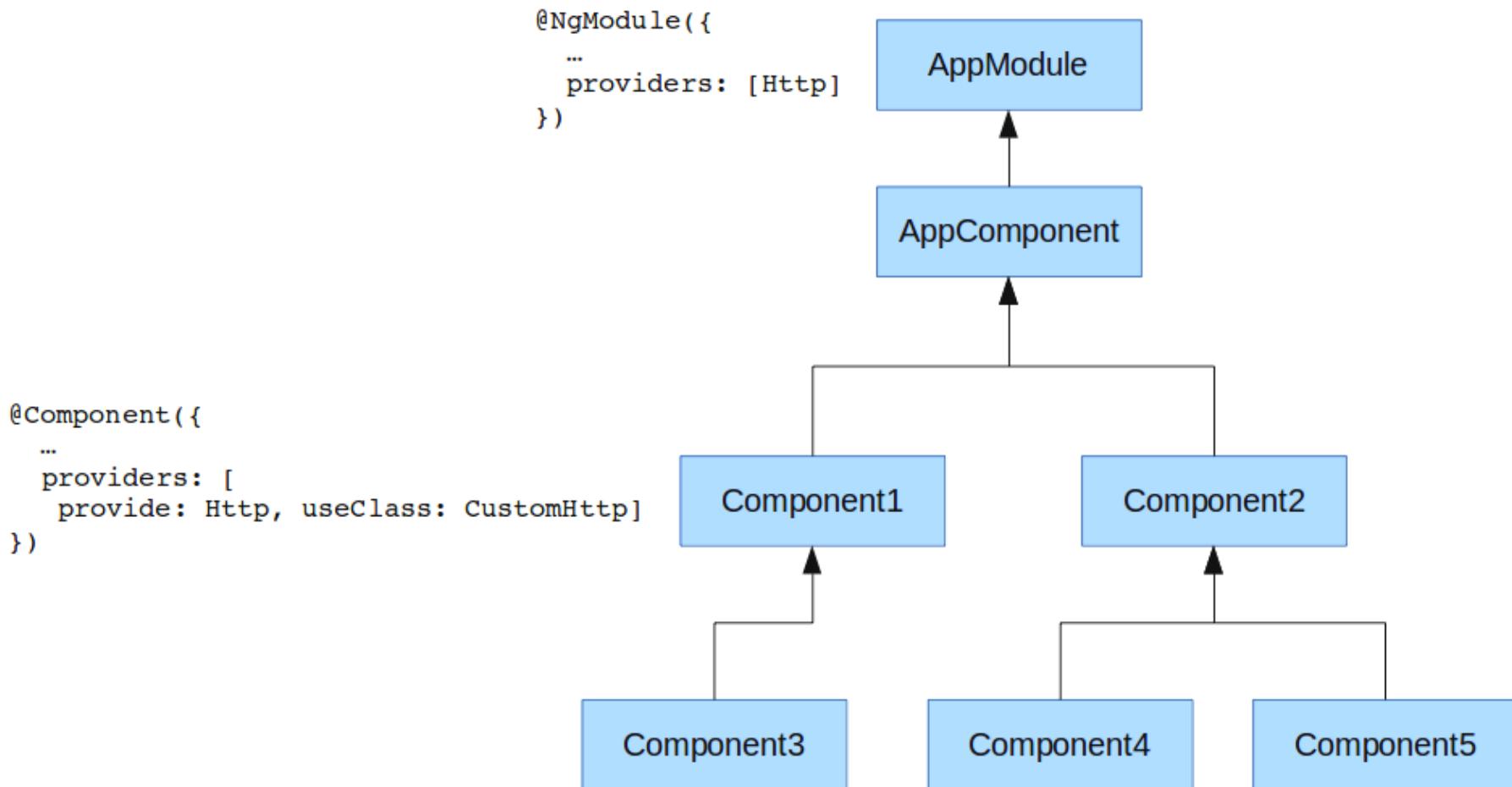


Hiérarchie d'injecteurs



- Chaque composant peut définir un injecteur avec un certain nombre de providers
- Chaque provider fournit un singleton d'un service
- Si un composant a besoin d'un service mais que son injecteur n'a pas de provider correspondant, il demande à l'injecteur de son parent

Hiérarchie d'injecteurs



Injection de Dépendances - Tests



- Possibilité de bénéficier de l'injection de dépendance grâce à la méthode `inject`
- Définition des services injectés dans les tests via la méthode `configureTestingModule` de l'objet `TestBed` (propriété `providers`)
- Méthode `async` utilisée pour tester les services asynchrones (utilise le mécanisme de `Zone`)

```
import { TestBed, async, inject } from '@angular/core/testing';

describe('UserService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({ providers: [UserService] });
  });

  it('should return 1 user', async(inject([UserService], service => {
    service.getUsers().then(users => {
      expect(users.length).toBe(1);
    });
  ))});
});
```





Pipes

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- *Pipes*
- Service HTTP
- Router
- Formulaires
- Server-side Rendering

Les Pipes



- Mécanisme permettant la manipulation d'une donnée avant son utilisation
- Similaire aux filtres dans **AngularJS**
- Utilisation dans les templates HTML similaires à l'ancienne version
- Possibilité de définir des **Pipes** pure ou impure
- Pipes disponibles par défaut dans le framework (`@angular/common`):
 - `LowerCasePipe` , `UpperCasePipe`
 - `CurrencyPipe`, `DecimalPipe`, `PercentPipe`
 - `DatePipe`, `JSONPipe`, `SlicePipe`
 - `I18nPluralPipe`, `I18nSelectPipe`
 - `AsyncPipe`

Les Pipes - Utilisation dans les Templates



- Les **Pipes** disponibles par défaut sont directement utilisables dans les templates
- Les Templates Angular supportent le caractère `|` pour appliquer un **Pipe**
- Possibilité de chaîner les pipes les uns à la suite des autres

```
{{ myVar | date | uppercase }}  
//FRIDAY, APRIL 15, 1988
```

- Certains pipes sont configurables
 - Séparation des paramètres par le caractère `:`

```
 {{ price | currency:'EUR':true }}
```

Les Pipes - Création



- Définir une classe implémentant l'interface `PipeTransform`
- Implémenter la méthode `transform`
- Annoter la classe avec le décorateur `@Pipe`
- Exporter cette classe via `export`

```
import {isString, isBlank} from '@angular/core/src/facade/lang';
import {InvalidPipeArgumentError} from
  '@angular/common/src/pipes/invalid_pipe_argument_error';
import {PipeTransform, Pipe} from '@angular/core';

@Pipe({name: 'mylowercase'})
export class MyLowerCasePipe implements PipeTransform {
  transform(value: string, param1:string, param2:string): string {
    if (isBlank(value)) return value;
    if (!isString(value)) {
      throw new InvalidPipeArgumentError(MyLowerCasePipe, value);
    }
    return value.toLowerCase();
  }
}
```



Les Pipes - Utilisation

- Les pipes externes nécessaires à votre applications doivent :
 - être définis dans un module importé par votre application (`ngModule`)
 - être définis dans la propriété `declarations` du décorateur `ngModule` de votre application

```
import {Component} from '@angular/core';

@Component({
  selector: 'app',
  template: '<h2>{{ 'Hello World' | mylowercase }}</h2>'
})
export class App {}
```



Les Pipes - Utilisation



- Utilisation de l'injection de dépendances pour utiliser un **Pipe**
- Pas nécessaire d'utiliser un service **\$filter** ou une règle de nommage (**dateFilter**) comme en **AngularJS**

```
import {Component} from '@angular/core';
import {MyLowerCasePipe} from './mylowercase';

@Component({
  selector: 'app',
  providers: [MyLowerCasePipe]
})
class App {
  name:string;

  constructor(public lower:MyLowerCasePipe){
    this.string = lower.transform('Hello Angular');
  }
}
```

Les Pipes - pures et impures



- Deux catégories de **Pipes** : pure et impure
- **Pipes** pures par défaut: exécuté seulement quand l'input du pipe subit un changement "pure"
 - changement de référence
 - changement d'une valeur primitive (boolean, number, string...)
- Ceci optimise les performances du mécanisme de détection de changement
- Mais, pas toujours le comportement souhaité :
 - ajout/suppression d'un objet dans un tableau (la référence du tableau ne change pas)
 - modification d'une propriété d'un objet



Les Pipes - impure

- Exécuté à chaque cycle du système de **Change Detection**
- Pour définir un **Pipe** impure, mettre la propriété **pure** à **false**

```
@Pipe({  
  name: 'myImpurePipe',  
  pure: false  
})  
class MyImpurePipe {  
  transform(value){ ... }  
}
```

Les Pipes - AsyncPipe



- Exemple de pipe impure
- **Pipe** recevant une **Promise** ou un **Observable** en entrée
- Retournera la donnée émise

```
@Component({
  selector: 'pipes',
  template: '{{ promise | async }}'
})
class PipesAppComponent {
  promise: Promise;

  constructor() {
    this.promise = new Promise(function(resolve, reject) {
      setTimeout(function() {
        resolve("Hey, this is the result of the promise");
      }, 2000);
    });
  }
}
```

Les Pipes - Tests



- Instanciation du **Pipe** dans une méthode **BeforeEach**
- Appel de la méthode **transform** pour tester tous les cas possibles

```
import {MyLowerCasePipe} from './app/mylowercase';

describe('MyLowerCasePipe', () => {
  let pipe;

  beforeEach(() => { pipe = new MyLowerCasePipe(); });

  describe('transform', () => {
    it('should return lowercase', () => {
      var val = pipe.transform('SOMETHING');
      expect(val).toEqual('something');
    });
  });
});
```





Service HTTP

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- *Service HTTP*
- Router
- Formulaires
- Server-side Rendering

Les Observables



- Le pattern **Observable** se base sur la librairie **RxJS**
- Documentation ici : <https://github.com/Reactive-Extensions/RxJS>
- Traitement de tâches asynchrones similaires à des tableaux
- Permet d'avoir des traitements asynchrones retournant plusieurs données
- Un Observable peut être **cancelable**
- Utilisation de méthodes dérivées de la programmation fonctionnelle
 - `map`, `forEach`, `filter`, ...
- Utilisable pour les traitements asynchrones : WebSocket, gestion des événements JavaScript

Les Observables



- Tout observable peut être, comme un tableau, utilisé par des fonctions classiques :
 - `take(n)` va piocher les n premiers éléments.
 - `map(fn)` va appliquer la fonction fn sur chaque événement et retourner le résultat.
 - `filter(predicate)` laissera passer les seuls événements qui répondent positivement au prédictat.
 - `reduce(fn)` appliquera la fonction fn à chaque événement pour réduire le flux à une seule valeur unique.
 - `merge(s1, s2)` fusionnera les deux flux.
 - `subscribe(fn)` appliquera la fonction fn à chaque évènement qu'elle reçoit.
 - `debounce(ms)` retardera l'exécution d'un observable



Les Observables - Exemple simple

- Exemple complet d'utilisation des Observables
 - `getDataFromNetwork` et `getDateFromAnotherRequest` sont des traitements asynchrones

```
getDataFromNetwork()
  .debounce(300)
  .filter (rep1 => rep1 != null)
  .flatMap(rep1 => getDateFromAnotherRequest(rep1))
  .map(rep2 => ` ${rep2} transformed`)
  .subscribe(value => console.log(`next => ${value}`))
```



Les Observables - Création



- Conversion de `setInterval` en `Observable`

```
import {Observable} from 'rxjs/Observable';
import {Subscriber} from 'rxjs/Subscriber';

@Component({})
export class AppComponent {

    private subscriber:Subscriber;

    constructor() {
        let source = new Observable(observer => {
            let interval = setInterval(_ => observer.next('TICK')), 1000);
            return function () {
                observer.complete();
                clearInterval(interval);
            };
        });
        this.subscriber = source.subscribe(v => console.log(v));
    }

    reset() { this.subscriber.unsubscribe(); }
}
```



Les Observables dans Angular

- Angular utilise ce système d'Observables à plusieurs endroits :
 - requêtes HTTP
 - intéraction avec un formulaire
 - affichage des vues par le **router**
 - **ngrx** : **ngrx/store**, **ngrx/devtools**, **ngrx/router**, ...



In Memory API

- L'équipe d'Angular propose le module **angular2-in-memory-api** pour commencer à intégrer une API sans serveur
 - se base sur une implémentation **in-memory** du service **XHRBackend**
 - idéal pour commencer les développements

```
npm install --save angular2-in-memory-web-api
```

- Nécessité d'implémenter l'interface **InMemoryDbService**
- Surcharger dans le système d'Injection de Dépendances l'implémentation de **XHRBackend** à utiliser

In Memory API



- Exemple d'utilisation :
 - Implémentation de l'interface `InMemoryDbService`

```
import { InMemoryDbService } from 'angular2-in-memory-web-api'

//API accessible via app/heroes
export class HeroData implements InMemoryDbService{
  createDb() {
    let heroes = [
      { id: '1', name: 'Windstorm' },
      { id: '2', name: 'Tornado' }
    ];
    //return {heroes: heroes};
    return {heroes};
  }
}
```

In Memory API



- Exemple d'utilisation :
 - Enregistrement de notre `InMemoryDbService`
 - Utilisation de l'implémentation `InMemoryBackendService` pour l'interface `XHRBackend`

```
import { XHRBackend } from '@angular/http';
import { InMemoryBackendService, SEED_DATA } from 'angular2-in-memory-web-api';
import { HeroData } from './hero-data';

@NgModule({
  providers: [
    { provide: XHRBackend, useClass: InMemoryBackendService }, // in-mem server
    { provide: SEED_DATA, useClass: HeroData } // in-mem server
  ],
  data: []
})
export class AppModule { }
```



- **Angular** fournit un ensemble de services pour pouvoir communiquer via des requêtes AJAX
- Services sont disponibles via le module **HttpModule** que nous devons importer dans notre module applicatif.
- Se base sur le pattern **Observable** contrairement à AngularJS et ses **Promises**
 - Plus grande flexibilité grâce aux différents opérateurs de **RxJS** : **retry**, ...
- Nous injecterons le service **Http** pour envoyer nos requêtes HTTP
- D'autres providers disponibles : **RequestOptions** similaire à **transformRequest** d'AngularJS
- Bonne pratique : implémenter les appels REST dans des services



- Exemple simple d'un service utilisant `Http`
 - Import d'`Http` depuis le module `@angular/http`
 - Injection du service via le constructeur
 - La méthode du service retournera le résultat de la requête `HTTP`

```
import { Http } from '@angular/http';
import { Injectable } from '@angular/core';

@Injectable()
export class ContactService {
  constructor(private http:Http){ }

  getContacts() {
    return this.http.get('people.json');
  }
}
```



HTTP - Configuration

- La requête HTTP pourra être configurée via un objet `RequestOptionsArgs`
- Possibilité de définir la méthode HTTP utilisée, les `headers`, le corps de la requête ...
- Structure de l'interface `RequestOptionsArgs` :

```
url : string
method : string | RequestMethod
search : string | URLSearchParams
headers : Headers
body : string
```

- `RequestMethod` : enum avec les différents méthodes HTTP possibles
- `Headers` : correspond à la spécification `fetch`

HTTP - Exemple



- Requête HTTP de type **PUT** avec surcharge des **Headers**

```
import {Http, Headers} from '@angular/http';

export class ContactService {
  constructor(private http:Http){ }

  save(contact){
    let headers = new Headers();
    headers.set('Authorization', 'xxxxxxx');

    return this.http.put('rest/contacts/' + contact.id, contact, {headers});
  }
}
```



HTTP - Exemple

- Exemple simple d'une requête HTTP

```
import {Http, Response} from '@angular/http';
import {Component} from '@angular/core';
import 'rxjs/add/operator/map';

@Component({selector: 'app', template: '{{displayedData}}'})
export class AppComponent {
    private displayedData;

    constructor(private http:Http) {
        http.get('people.json')
            .map((result:Response) => result.json())
            .subscribe(jsonObject => {
                this.displayedData = jsonObject;
            });
    }
}
```



HTTP - Exemple

- Exemple d'un Observable utilisant la méthode **filter**

```
import 'rxjs/add/operator/map';
import {MyObject} from './MyObject';
import {Http, Response} from '@angular/http';
import {Component} from '@angular/core';

@Component({selector: 'app', template: '{{displayedData | json}}'})
export class AppComponent {
    private displayedData = [];

    constructor(private http:Http) {
        http.get('people.json')
            .map((result:Response) => result.json())
            .filter(data => data.hasToBeDisplayed)
            .map(data => new MyObject(data.id, data.name))
            .subscribe((jsonObject:MyObject) => {
                this.displayedData.push(jsonObject);
            });
    }
}
```



HTTP - Surcharger les en-têtes

- Possibilité de surcharger les paramètres **HTTP** par défaut
 - grâce à l'injection de dépendances, utilisation du token **RequestOptions**
 - token utilisé dans le constructeur du service **Http**
 - utilisation de la classe **BaseRequestOptions** pour bénéficier des paramètres par défaut définis par **Angular**

```
import {provide} from '@angular/core';
import {Http, BaseRequestOptions, RequestOptions} from '@angular/http';

class MyOptions extends BaseRequestOptions {
  search: string = 'coreTeam=true';
}

@NgModule({
  providers: [{provide: RequestOptions, useClass: MyOptions}],
})
export class AppModule { }
```

HTTP - Tests



- Possibilité de définir une implémentation bouchonnée du service `Http`

```
import {TestBed, inject} from '@angular/core/testing';
import {Http, RequestOptions, Response, ResponseOptions} from '@angular/http';
import {MockBackend} from '@angular/http/testing';
import 'rxjs/add/operator/map';

describe('UserService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        UserService,
        MockBackend,
        BaseRequestOptions,
        {
          provide: Http,
          useFactory: (backend: MockBackend, defaultsOptions: RequestOptions) =>
            new Http(backend, defaultsOptions),
          deps: [MockBackend, RequestOptions]
        }
      ]
    });
  });
});
```

HTTP - Tests



- Création d'un test avec cette implémentation bouchonnée

```
it('return return 1 user', inject([UserService, MockBackend],  
  (service, mockBackend) => {  
    let mockedUsers = [new User()];  
  
    let response = new Response(new ResponseOptions({body: mockedUsers}));  
  
    mockBackend.connections.subscribe(connection=>connection.mockRespond(response));  
  
    service.getUsers().subscribe(users => {  
      expect(users.length).toBe(1);  
    });  
  }));
```





Router

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- *Router*
- Formulaires
- Server-side Rendering

Router



- Module totalement différent du module **ngRoute**
- **ngRoute** était un module trop simpliste
 - Un seul **ngView** dans l'application
 - Pas de vue imbriquée
- Développement d'un nouveau Router
 - Prise en compte des différents cas d'utilisation : authentification, login, permission, ...
 - Etude des autres solutions : **ui-router**, **route-recognizer** et **durandal**
 - 3e implémentation depuis le début de **Angular**
- Compatible avec **AngularJS** et **Angular**
- Permet de faciliter la migration d'une application **AngularJS** vers **Angular**

Router



- Router orienté **composant**
- Association d'un composant principal avec une URL de votre application
- Utilisation d'une méthode **RouterModule.forRoot** pour définir la configuration
- Utilisation de la directive **RouterOutlet** pour définir le point d'insertion
- Navigation entre les pages via la directive **RouterLink**
- Installation via **NPM** : `npm install --save @angular/router`

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [RouterModule],
})
export class AppModule { }
```



Router - forRoot

- Méthode permettant d'enregistrer de nouvelles routes
- Elle prend en paramètre un tableau de **RouterConfig**, qui correspond à un tableau de **Route**

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent, ContactsComponent, ContactComponent } from './pages';

export const routes: Routes = [
  { path: '', component: HomeComponent }, // path: '/'
  { path: 'contacts', component: ContactsComponent },
  { path: 'contact/:id', component: ContactComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
})
export class AppModule { }
```

Router - RouterOutlet



- Directive à utiliser via l'attribut **router-outlet**
- Permet de définir le point d'insertion dans le composant principal
- Le composant sera inséré en tant qu'enfant de l'élément sur lequel la directive **RouterOutlet** est utilisée
- Possibilité de définir la vue via un attribut **name** (utilisé pour définir plusieurs vues au même niveau)
- Possibilité de créer des vues enfant grâce à l'utilisation de **RouterOutlet** imbriquées

```
@Directive({selector: 'router-outlet'})
export class RouterOutlet {
  constructor(
    private elementRef: ElementRef,
    private loader: DynamicComponentLoader,
    private parentRouter: routerMod.Router,
    @Attribute('name') nameAttr: string) {...}
}
```



Router - RouterOutlet

- Exemple simple de la directive **RouterOutlet**

```
import { Component } from '@angular/core';

@Component({
  template: `
    <header><h1>Title</h1></header>
    <router-outlet></router-outlet>
  `,
})
class AppComponent { }
```

Router - RouterLink



- Permet de naviguer d'une route à une autre
- Utilisation de la méthode `navigate` du service `Router`
- La directive `RouterLink` s'attend à un tableau de noms de routes, suivi par d'éventuels paramètres

```
@Component({
  template: `
    <div>
      <h1>Hello {{message}}!</h1>
      <a [routerLink]="'contacts'">Link 1</a>
      <a [routerLink]=[ 'contact', 1]>Link 2</a>
      <a [routerLink]=[ 'contact', id]>Link 3</a>
      <router-outlet></router-outlet>
    </div>`
})
class AppComponent {
  id: number = 2;
}
```



Router - RouterOutlet imbriquées

- Imbrication de plusieurs **RouterOutlet** pour définir une hiérarchie de vues

```
import { RouterModule, Routes } from '@angular/router';
import { ContactComponent, EditComponent, ViewComponent } from './pages';

export const routes: Routes = [
{
  path: 'contact/:id',  component: ContactComponent, children: [
    {path: 'edit', component: EditCmp},
    {path: 'view', component: ViewCmp}
  ]
};
];

const routing = RouterModule.forRoot(routes);
```



Router - RouterLink en détails



- Utilisation via un attribut `routerLink`
- Configuration de la route désirée via ce même attribut `routerLink`
- Attribut `href` généré par le service `Location`
- Ajout de classes CSS si la route est active (directive `routerLinkActive`)

```
@Directive({selector: '[routerLink]'})
export class RouterLink implements OnChanges {
    @Input() routerLink: any[]|string;
    @HostBinding() href: string;

    ngOnChanges(changes: {}): any { this.updateTargetUrlAndHref(); }

    @HostListener('click', [])
    onClick(): boolean {
        ...
        this.router.navigateByUrl(this.urlTree);
        return false;
    }
}
```

Router - Stratégies pour le génération des URLs

- **PathLocationStrategy** (stratégie par défaut)
 - Nécessite la définition de l'URL de base de votre application (`APP_BASE_HREF` ou `<base>`)

```
router.navigate(['contacts']); //example.com/my/app/contacts
```

- **HashLocationStrategy**

```
router.navigate(['contacts']); //example.com#/contacts
```

- Possible de configurer l'implémentation à utiliser

```
import {HashLocationStrategy, LocationStrategy } from '@angular/common';

@NgModule({
  providers: [{ provide: LocationStrategy, useClass: HashLocationStrategy }],
})
export class AppModule { }
```

Router - Configuration de l'URL de base de l'application

- Définition d'un nouveau **provider** pour la constante `APP_BASE_HREF`
- Sera utilisé lors de la génération des différentes URLs

```
import {Component} from '@angular/core';
import {APP_BASE_HREF} from '@angular/common';

@NgModule({
  providers: [{ provide: APP_BASE_HREF, useValue: '/my/app' }],
})
export class AppModule { }
```

Router - Récupération des paramètres d'URL

- Utilisation du service `ActivatedRoute` et `params (Observable)`

```
@Component({
  template: `
    <a [routerLink]=["contact", 1]"></a>
    <router-outlet></router-outlet>
  `})
class AppComponent { }
```

```
import { ActivatedRoute } from '@angular/router';

@Component({
  template: '<main><router-outlet></router-outlet></main>'
})
export class ProductComponent {
  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.params.subscribe(params => {
      let id = +params['id']; // (+) conversion 'id' string en number
      ...
    });
  }
}
```

Router - Récupération des paramètres d'URL

- Utilisation du service `ActivatedRoute` et `snapshot`

```
@Component({
  template: `
    <a [routerLink]=["contact", 1]"></a>
    <router-outlet></router-outlet>
  `})
class AppComponent { }
```

```
import { ActivatedRoute, ActivatedRouteSnapshot } from '@angular/router';

@Component({
  template: '<main><router-outlet></router-outlet></main>'
})
export class ProductComponent {
  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    const s: ActivatedRouteSnapshot = this.route.snapshot;
    // Valeur initiale des paramètres
    let id = +s.params.id;
    ...
  }
}
```

Router - Cycle de Vie

- Possibilité d'intéragir avec le cycle de vie de la navigation (**Lifecycle Hooks**)
- Interface **CanActivate** : interdire / autoriser l'accès à une route

```
import { Injectable } from '@angular/core';
import { CanActivate, Router, ActivatedRouteSnapshot } from '@angular/router';
import { AuthService } from '../shared/auth.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}
  canActivate(route: ActivatedRouteSnapshot) {
    if(this.authService.isLoggedIn()) return true;
    this.router.navigate(['/login']);
    return false;
  }
}

// fichier app/application.routes.ts
import { AdminComponent, AuthGuard } from './admin/';
export const AppRoutes: RouterConfig = [
  { path: 'admin', component: AdminComponent, canActivate: [AuthGuard] }
```





Gestion des Formulaires

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- *Formulaires*
- Server-side Rendering

Formulaires et Angular



- Se base sur les mécanismes standards des formulaires HTML
- Supporte les types de champs de saisie habituels et les validations natives
 - `input[text]`, `input[radio]`, `input[checkbox]`, `input[email]`,
`input[number]`, `input[url]`
 - `select`
 - `textarea`
- Il est possible de créer ses propres composants
- Deux syntaxes disponibles :
 - Syntaxe réactive
 - Syntaxe HTML



Formulaires : Principe général

- Associer des champs de saisie à des propriétés du composant grâce à `ngModel`
- Nommer les champs grâce à l'attribut `name`
- Ajouter des validateurs
- Appeler une méthode du composant pour traiter le formulaire en JavaScript





Formulaires : Principe général

- **ngModel** : Gère le binding entre la variable du contrôleur et le champ HTML

```
<input type="text" [(ngModel)]="contact.name" name="name">
```

- **submit** : Associe une méthode à la soumission du formulaire

```
<form (submit)="saveForm()">
  <button type="submit">Save</button>
</form>
```



Validation : Désactiver la gestion native

- Par défaut, les navigateurs effectuent les validations nativement
 - Manque de cohérence visuelle avec l'application et entre navigateurs
 - Interfère avec le mécanisme d'AngularJs
- Solution : Désactiver la validation native et l'effectuer par Angular
- Attribut **novalidate** sur le formulaire
 - Attribut standard HTML5
 - Attribut ajouté automatiquement par **Angular**

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





Formulaires : FormControl

- Un **FormControl** est une classe représentant un **input** contenant :
 - La valeur
 - L'état (dirty, valid, ...)
 - Les erreurs de validation
- Angular crée un **FormControl** dès l'utilisation de la directive **ngModel**
- Il utilise la valeur de la propriété **name** comme libellé
- On peut l'associer à une variable pour l'utiliser dans le template avec la syntaxe **#inputName="ngModel"**





Formulaires : FormControl

Exemple complet:

```
<form novalidate (submit)="saveForm()">
  <input type="text" name="myName" [(ngModel)]="contact.name"
#nameInput="ngModel" required>
  <span [hidden]="nameInput.valid">Error</span>

  <button type="submit">
    Validate
  </button>
</form>
```





Validation : Concept

- Un champ peut posséder un ou plusieurs validateurs
 - Standards ou personnalisés
 - Support des validateurs HTML5 : **required**, **min**, **max**, ...
- L'état de la validation est stocké par l'objet **FormControl** dans la propriété **errors**

```
<input type="text" [(ngModel)]="contact.name" #name="ngModel"  
name="name" required>  
<span [hidden]="!name.errors?.required">Name is not valid</span>
```



Validation : État du formulaire et des champs

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



Validation : Crédit d'un validateur

Il est possible de créer ses propres validateurs avec une classe implémentant l'interface **Validator**

```
@Directive({
  selector: '[pattern][ngModel]',
  providers: [
    { provide: NG_VALIDATORS, useExisting: PatternValidator, multi: true }
  ]
})
export class PatternValidator implements Validator {
  @Input('pattern') pattern: string;

  validate(c: AbstractControl): { [key: string]: any } {
    if (c.value && c.value.match(new RegExp(this.pattern))) {
      return null;
    }
    return { pattern: true };
  }
}
```

```
<input type="text" name="name" [(ngModel)]="contact.name"
pattern="[a-z]{10}">
```



NgForm



La directive **NgForm** est automatiquement associée à chaque balise `<form>`

- Autorise l'utilisation de l'évènement **ngSubmit**
- Crée un **FormGroup** pour gérer les inputs contenus dans le formulaire
- Instance de la directive utilisable dans le template : `#myForm="ngForm"`

```
<form #myForm="ngForm" novalidate (submit)="onSubmit()">
  <input type="text" name="myName" [(ngModel)]="contact.name"
#nameInput="ngModel" required>

  <span [hidden]="nameInput.valid">Error</span>

  <button type="submit" [disabled]="myForm.invalid">
    Validate
  </button>
</form>
```





Server-side Rendering

Sommaire



- Rappels
- Présentation
- Démarrer une application Angular
- Tests
- Template & Composants
- Directives
- Injection de Dépendances
- Pipes
- Service HTTP
- Router
- Formulaires
- *Server-side Rendering*

Besoin



- Indexation par les moteurs de recherche (SEO)
- Prévisualisation (comme dans le partage facebook)
- Amélioration progressive
 - Proposer une version simple pour tous
 - Enrichir l'expérience en fonction du client
- **Accélérer le chargement de l'application**

Angular Universal



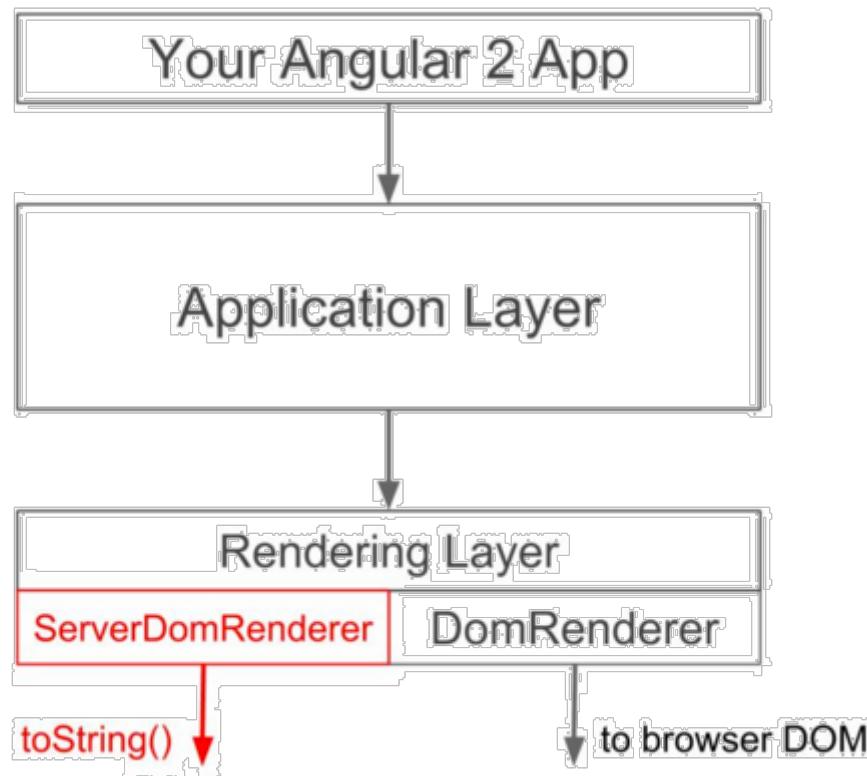
- Projet Angular officiel. Contrairement aux projets de Server Side Rendering pour [AngularJS](#)
- Contient deux modules
 - Le premier rend le code côté serveur
 - Le deuxième enregistre les actions de l'utilisateur pour les rejouer une fois l'interface complètement chargée
- Le terme Universal vient de l'idée de pouvoir proposer l'application dans d'autres environnements que celui du navigateur
- Pas encore assez stable pour la mise en production



Méchanisme



- **AngularJS** fortement lié au DOM
- **Angular** introduit une séparation du mécanisme de rendu



Procédure de rendu



- Le moteur de rendu (Express en NodeJS) va construire le HTML
- Le plugin **Angular Universal** va réaliser le **bootstrap** de l'application
- La réponse des appels REST est attendue
- La page complètement construite est retourné à l'utilisateur
- La librairie **Preboot** de **Angular Universal** enregistre les actions de l'utilisateur
- Le navigateur client termine de charger le code javascript
- La librairie **Preboot** rejoue les actions de l'utilisateur

Mise en place



- Le plus simple est de reprendre le starter
<https://github.com/angular/universal-starter>
- Créer deux points d'entrées pour l'application
 - Classique pour le client avec la fonction **bootstrap**
 - Pour le serveur avec la mise en place de **Express** et de **Angular Universal**

Rendu serveur



Elements notables du script de lancement du serveur

```
import 'angular2-universal-polyfills';
import { createEngine } from 'angular2-express-engine';
import { NgModule } from './main.node';

app.engine('.html', createEngine({
  ngModule: NgModule,
  providers: [
  ]
}));

app.set('views', __dirname);
app.set('view engine', 'html');

app.get('/*', (req, res) => {

  res.render('index', {
    req,
    res,
    preboot: true
  });
});
```

