Angular

Sommaire

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

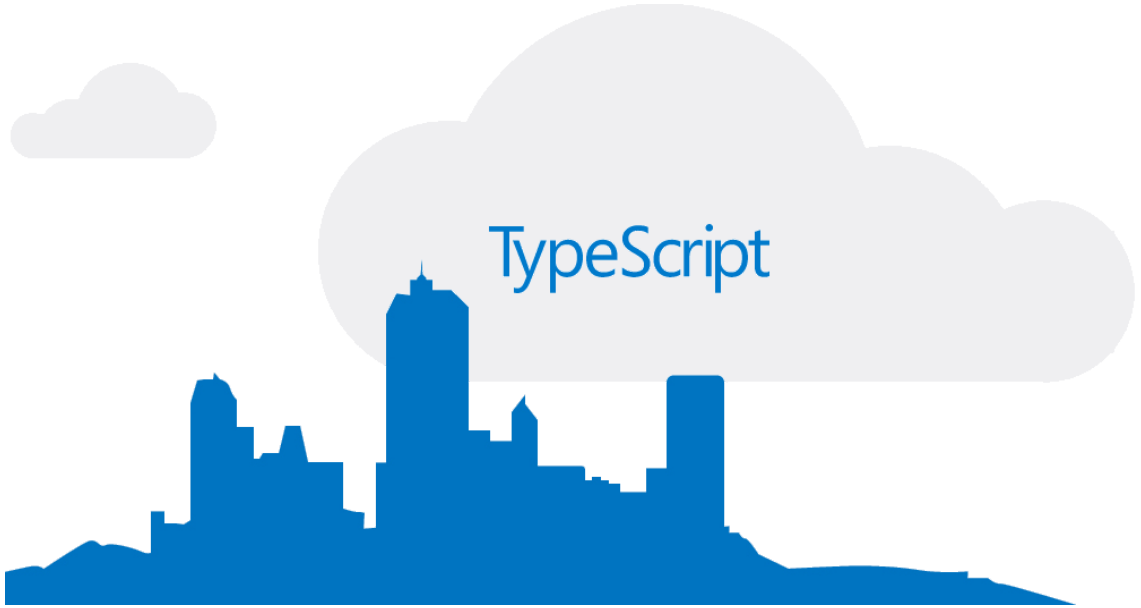
Logistique

- Horaires
- Déjeuner & pauses
- Autres questions ?

Rappels

Sommaire

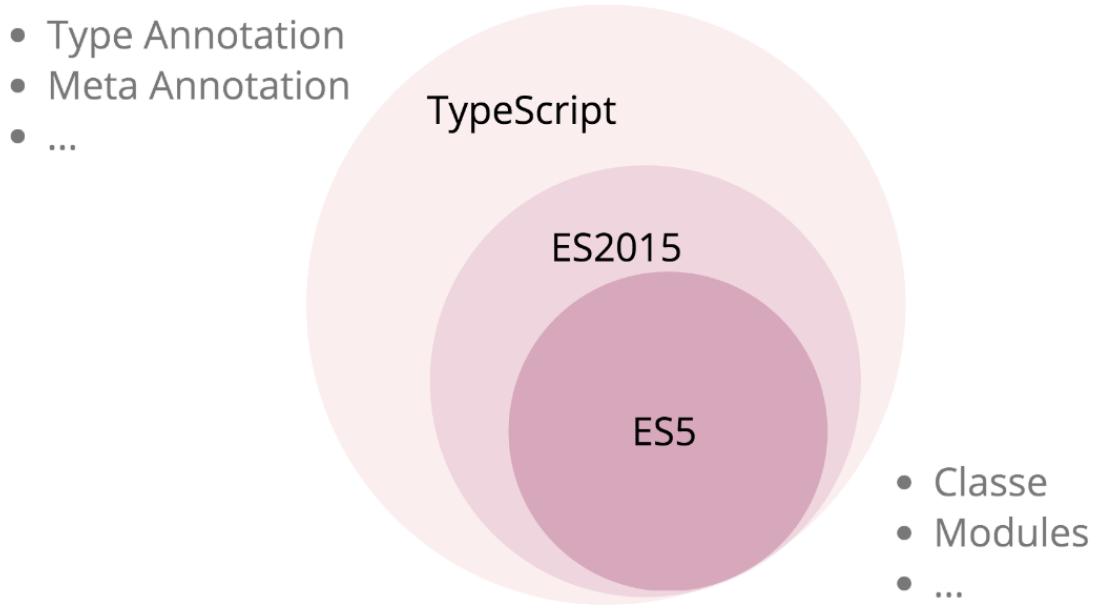
Introduction



- Langage créé par **Anders Hejlsberg** en 2012
- Projet open-source maintenu par **Microsoft** (Version actuelle **2.5**)
- 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

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

Types primitifs

- Pour déclarer une variable :

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

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

Fonctions

- Comme en JavaScript : fonctions nommées, anonymes et arrow functions
- Ajout du typage des arguments et de la valeur de retour

```
// Fonction nommée
function namedFunction(arg1: number, arg2: string): void { }

// Fonction anonyme
const variableAnonymousFunction = function(arg: boolean): void { };

// Arrow function
const variableArrowFunction = (arg: any): void => {  };
```

- Peut retourner une valeur grâce au mot clé `return`
- Possibilité d'avoir des paramètres optionnels ou avec une valeur par défaut

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

Arrays

- Permet de manipuler un tableau d'objets
- 2 syntaxes pour définir les tableaux : littérale ou par le constructeur

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

// Syntaxe utilisant le constructeur `Array`
let list: Array<number> = new Array<number>(1, 2, 3);
```

- Ces 2 syntaxes aboutiront au même code JavaScript

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` (toujours explicité)

```
class Person {
  firstName: string;
  lastName: string;

  constructor() {}

  sayHello() {
    console.log(`Hello, I'm ${this.firstName} ${this.lastName}`);
  }
}

let person = new Person();
```

Classes

- Trois scopes disponibles : `public`, `private` et `protected`
- Utilise le scope `public` par défaut
- Scope `protected` apparu en TypeScript 1.3
- Possibilité de définir des propriétés et méthodes statiques : `static`
- Possibilité de définir des propriétés en lecture seule : `readonly`
- Raccourcis pour déclarer et initialiser des propriétés

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

// ===

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

Classes - Accesseurs

- Possibilité de définir des accesseurs pour accéder à une propriété
- Utiliser les mots clé `get` et `set`
- Attention à l'espacement après 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 possibles
  - Vérification des paramètres d'une fonction
  - Vérification de la signature d'une fonction
  - Vérification de l'implémentation d'une classe

```
// Les interfaces pour typer facilement
interface Config {
  someProperty: string
}

const config: Config = {
  someProperty: 'myValue'
};
```

Interfaces

- Utilisation la plus connue : implémentation d'une classe
- 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(): void {
    console.log('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
```

Génériques

- Possibilité de définir une classe générique
- Définition d'une liste de paramètres de types 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
```

NPM

- 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
- 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 accessibles des commandes dans la console

    Attention : Ne rend pas une librairie accessible pour tous les projets
  - `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 <moduleName>` ou `install <moduleName> --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'informations
  - 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` (depuis NPM3)

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 totale du framework
- Reprend certains concepts d'**AngularJS**
- Première version **beta** annoncée en octobre 2014
- Version **finale** `2.0.0` officielle sortie en septembre 2016
- Dernière version majeure `5.0.0` sortie en novembre 2017
- Programmation orientée **Composant**
- Framework conçu pour être plus performant et optimisé pour les mobiles
- http://angular.io/

Présentation - Numérotation

- Numérotation à partir de 2.0.0 pour se démarquer d'AngularJS
- Respect à partir de là de la norme **semver**
- Les versions majeurs ne seront plus des réécritures comme de la 1 à la 2
- Saut de la version 3.0.0 après le merge du projet **Router** déjà en 3.x
- Planification d'une version majeure tous les 6 mois dans le futur
- Sortie de la version 4.0.0 en mars 2017
  - Pas de grands bouleversements
  - Nouveau moteur de compilation des templates
  - Modularisation du système d'animations
  - Intégration du projet Universal
  - Passage à TypeScript 2.1+

Points négatifs d'AngularJS

- Différences entre les directives et `ngController`
- Two-way data-binding source de problèmes de performances
- 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('UserService', function (){
  const vm = this;
  vm.getUsers = function (){

  }
});
```

- Version Angular

```
@Injectable()
export class UserService {
  getUsers(): User[] {
    return [];
  }
}
```

Angular - Points Positifs

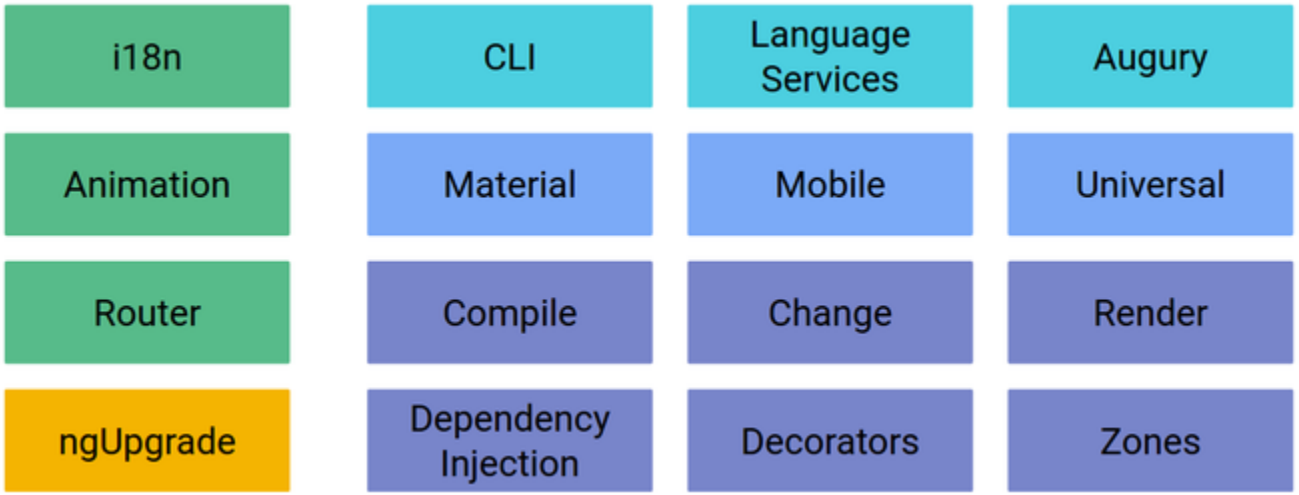
- Création d'application modulaire
- Utilisable avec plusieurs langages de programmation : `ES5`, `ES2015(ES6)`, `TypeScript` et `Dart`
- API plus simple que **AngularJS**
- Seuls trois types d'éléments seront utilisés : `directive`, `pipe` et les `services`
- Basé sur des standards : `Web Components`,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

Angular - Points Négatifs

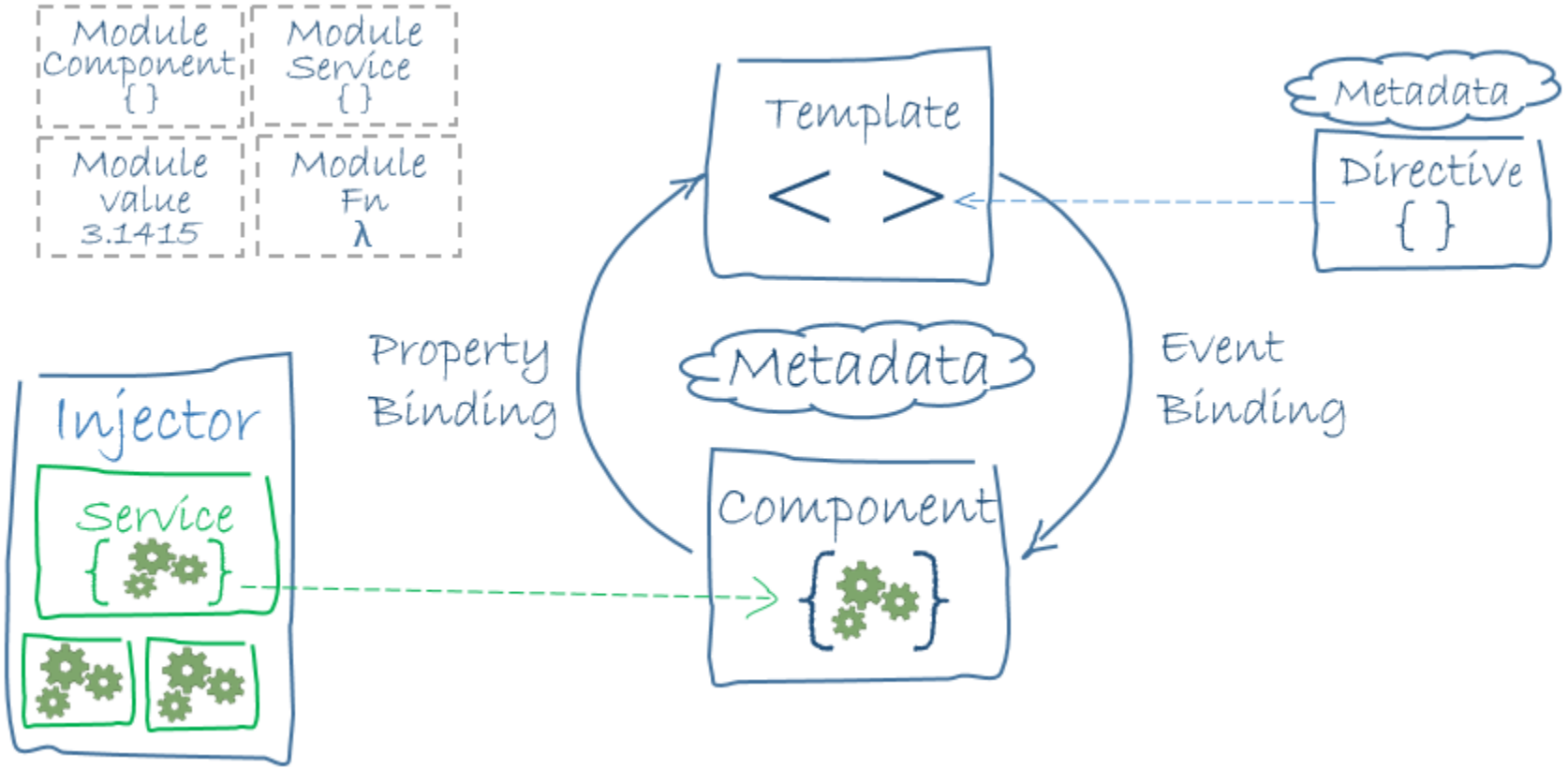
- Nouvelle phase d'apprentissage du framework si habitué à AngularJS
- Écosystème encore jeune
- Applications AngularJS incompatibles avec cette nouvelle version
- ngUpgrade permet de réutiliser du code AngularJS mais pas de migrer
- De nouveaux concepts à apprendre :
  - `Zone`
  - `Observable`
  - `Webpack`
  - ...

Angular = Une Plateforme

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



Architecture



Architecture

- Metadata : Configuration pour décrire le fonctionnement d'un composant
- Component : Classe TypeScript qui décrit son comportement
- Template : Code HTML réalisant le rendu à l'aide du component
- Modules : regroupement d'un ensemble de fonctionnalités
- Injector : système d'injection de dépendances d'Angular
- Directive : composant sans template (**ngFor**, **ngIf**, ...)
- Service : Code métier implémenté dans des classes qui seront injectées dans les différents composants

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

Commencer un nouveau projet

- Gestion des dépendances via **NPM**
  - les différents modules **Angular** : `@angular/common`, `@angular/core`...
  - Webpack : gestion des modules
  - RxJS : programmation réactive, dépendance forte d'Angular

```
npm init

npm install @angular/common @angular/core rxjs ...
```

- Initialisation et Configuration d'un projet **TypeScript**
- Configuration du système de gestion des modules (**Webpack**)

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

- En retard sur la sortie d'Angular, il est maintenant en `1.0.0`
- Basé sur le projet **Ember CLI**
- Permet de créer le squelette d'une application
- Embarque automatiquement les technologies suivantes :
  TypeScript, Webpack, Karma, Protractor, Préprocesseurs CSS ...
- Projet disponible sur **NPM**

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

- Propose des commandes pour le cycle de vie de l'application

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

Angular CLI

- Nombreuses commandes disponibles
- `ng generate` : Génère du code pour différents éléments d'Angular
  - `ng generate component Product` :
    Génère un nouveau composant avec template, style et test
  - `ng generate pipe UpperCase` : Génère un nouveau pipe
  - `ng generate service User` : Génère un nouveau service
  - `ng generate directive myNgIf` : Génère une nouvelle directive
- `ng test` : Lance les tests avec Karma
- `ng e2e` : Lance les tests end-2-end avec Protractor
- `ng lint` : Lance TSLint

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>
```

- L'ajout de la balise script peut également être réalisé avec un plugin

Webpack

- Version avec un fichier de configuration

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

- Webpack va lire le fichier de configuration automatiquement

```
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'
}
```

- Permet d'utiliser 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

- Système de recompilation automatique très performant
  - Utilisation de l'option `webpack --watch`
  - Webpack conserve le graphe des modules en mémoire
  - Regénère le `bundle` pour n'importe quel changement sur un des fichiers
- 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-loader']
  }]
},
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'
}
```

Lab 1

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
- Pour exécuter facilement les tests, on propose d'utiliser `Karma`
  - Il a été développé par l'équipe d'`AngularJS`
  - Il n'est pour autant ni indispensable ni lié à `Angular`
- `Jasmine` et `Karma` sont les outils utilisés dans une application générée avec Angular CLI

Jasmine

- Framework de tests : http://jasmine.github.io/
- Aucune dépendance vers d'autres frameworks
- Ne nécessite pas d'éléments du **DOM**

Jasmine - Structure

- Fonctions `describe` et `it` pour décrire la suite de tests
- Système de **matchers** : `toBe`, `toBeUndefined`, `toBeTruthy`, `toThrow`, ...
- Possibilité d'utiliser une bibliothèque externe comme `Chai`

```
describe('True value', () => {

  it('should be equal to true', () => {

     expect(true).toBe(true);

  });

});
```

Jasmine - Hooks

- Fonctions `beforeEach`, `afterEach`, `beforeAll`, `afterAll`
- Exécution d'une fonction avant ou après chaque ou tous les tests

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

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

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

Jasmine - Spies

- Jasmine propose un système de **Spies** inclus
- Il est également possible d'utiliser une librairie externe comme **Sinon**
- Création d'un spy : `jasmine.createSpy()` ou `spyOn(someObj)`
- Matchers sur un spy : `toHaveBeenCalled`, `toHaveBeenCalledWith`, `and.callThrough`, `and.returnValue`, `and.callFake`, `mySpy.calls`...

```
describe('Service objet:', function() {

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

});
```

Jasmine - TypeScript

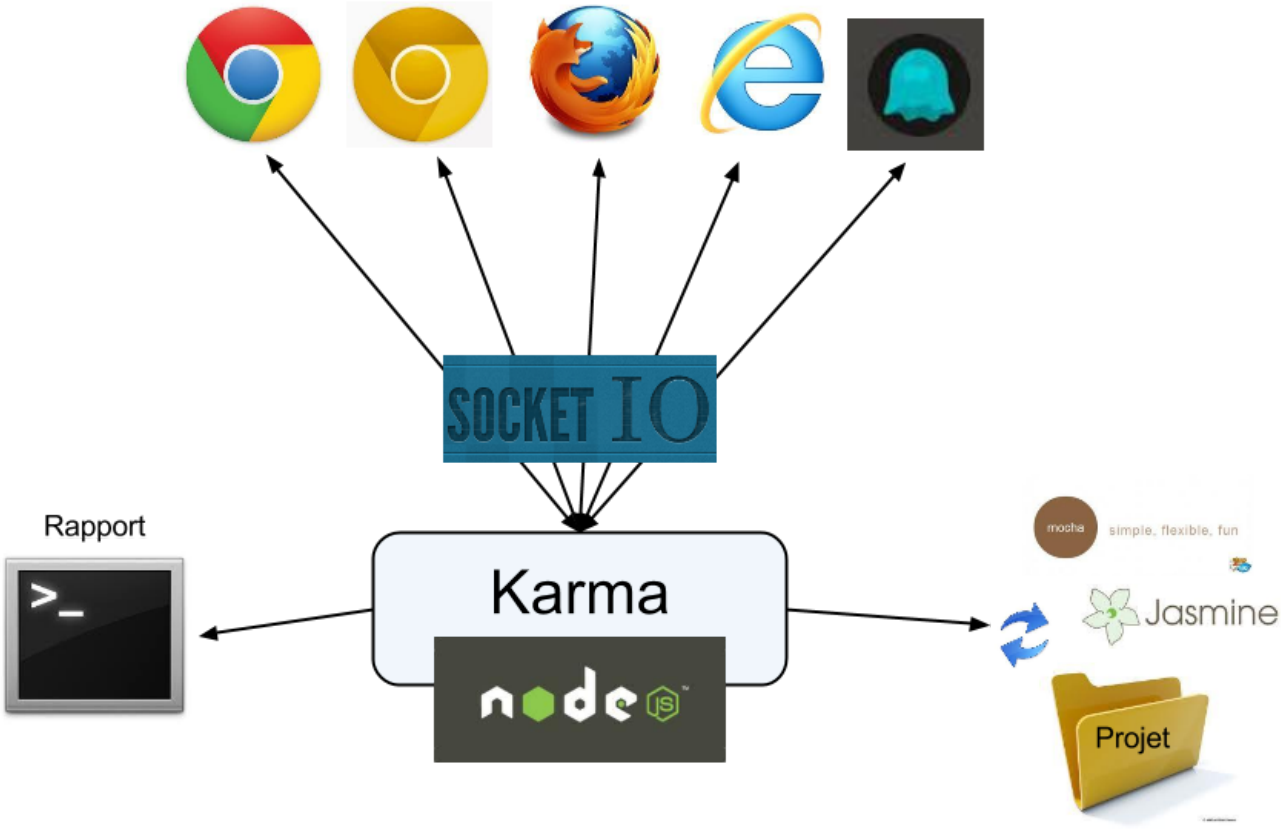
- Possibilité d'écrire des tests **Jasmine** en **TypeScript**

```
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



Avec Angular CLI

- Configuration automatique réalisée par **Angular CLI**
- Les outils suivants sont prêts à fonctionner ensemble :

  Webpack, **TypeScript**, **Angular**, **Jasmine**, **Karma**
- Les fichiers de tests sont automatiquement créés avec `ng generate (...)`

  `Composant` / `Service` / `Pipe`
- Ils se trouvent dans le même répertoire que l'élément à tester

  mon-service.spec.ts
- Exécution des tests :

```
ng test
```

Lab 2

Template & Composants

Sommaire

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

Composants

- Les composants sont les éléments de base d'Angular
- Définis à partir d'une classe TypeScript avec l'annotation `@Component`
- Seront activés par le sélecteur **CSS** de la propriété `selector`
- Un template est configuré de deux façons :
  - `template` : String literal (penser à la string multiline `)
  - `templateUrl` : Url d'un fichier HTML (relatif au composant)

Composants

- Les styles peuvent être configurés via deux propriétés :
  - styles

```
@Component({
  selector: 'app-root',
  template: `
    <h1>App Works</h1>`,
  styles: [`
    h1 { font-weight: normal; }
  `]
})
export class AppComponent { }
```

- styleUrls

```
@Component({
  selector: 'app-root',
  template: `
    <h1>App Works</h1>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

Composants

- Pour définir un composant qui sera appelé de cette façon :

```
<div>
  <h1>My Product</h1>
  <product></product>
</div>

<!-- attention, <product/> ne fonctionne pas -->
```

- Le composant **Angular** est implémenté ainsi

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

@Component({
  selector: 'product',
  template: `
    <article>
      <h1>My Product</h1>
    </article>
  `
})
export class ProductComponent { }
```

Templates

- Les templates d'Angular sont compilés avant d'être exécutés
  - Soit à chaud : **JIT** (Just In Time) par défaut
  - Soit au build : **AOT** (Ahead Of Time) `--aot` dans Angular CLI
- La compilation permet de détecter des erreurs dans le template
- Implique également que les templates doivent être syntaxiquement exacts
- Fonctionnement très différent d'AngularJS
  - AngularJS ne compilait pas les templates
  - Les templates d'AngularJS étaient du pur Web transmis au navigateur

Interpolation

- Système d'interpolation grâce à la syntaxe `{{ expression }}`
- L'expression doit retourner une valeur qui sera convertie en `string`
- Angular définit une syntaxe précise pour ces expressions
- https://angular.io/docs/ts/latest/guide/template-syntax.html#!#template-expressions
- La syntaxe est celle du JavaScript avec quelques exceptions
- Toutes les propriétés du composant sont accessibles directement
- Une expression ne doit pas modifier l'état de l'application

```
@Component({
  selector: 'product',
  template: `<p>{{ add(myProp, 2) }}</p>`
})
export class ProductComponent {
  myProp = 1;
  add(value1, value2) { return value1 + value2; }
}
```

Propriétés

- Syntaxe générique pour définir la valeur d'une propriété d'un élément **HTML**
- Différent d'AngularJS, où nous utilisons les attributs **HTML**
- Utilisation de la syntaxe `[property-name]="expression"`
- Syntaxe identique pour les propriétés des **éléments HTML standards**, les **composants** et les **directives** Angular et même les **Web Components**

```
<button [disabled]="isUnchanged">Save</button> <!-- propriété HTML -->
<button bind-disabled="isUnchanged">Save</button> <!-- alternative sans [] -->
<button data-bind-disabled="isUnchanged">Save</button> <!-- html5 strict -->
<hero-detail [hero]="currentHero"></hero-detail> <!-- propriété d'un composant --
>

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

- Les propriétés sont ***bindées***, la valeur sera mise à jour automatiquement si la valeur de l'expression change

Propriétés

- ***Attention à la différence entre attribut et propriété***
- Il existe des écarts entre les **propriétés du DOM** et les **attributs HTML**
- Angular propose alors un système appelé `Attribute Binding`
- Cas les plus courants : `aria-*`, `colspan`, `rowspan`, `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>
```

Input

- Un composant peut recevoir des paramètres
- Annotation `@Input()` sur une propriété de la classe du composant
- Le nom de la propriété sera celle à utiliser dans le template

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

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

Input

- Possibilité de surcharger le nom de la propriété avec `@Input('discount')`
- Les noms de propriétés sont sensible à la casse

```
@Component({ selector: 'product-detail', /* ... */ })
export class ProductComponent {
  @Input() product: Product;
  @Input('discount') percentDiscount: number;
}
```

- Pour utiliser ce composant

```
<product-detail [product]="myProduct" [discount]="10">
</product-detail>
```

- **Angular** vérifie les propriétés passées à un composant
- Il refusera une propriété qui n'existe pas ou non annotée `@Input()`

Évènements

- Syntaxe générique pour écouter un évènement d'un élément **HTML**
- Différent d'AngularJS, où nous utilisons les attributs **HTML**
- Utilisation de la syntaxe `(event-name)="expression"`
- Syntaxe identique pour les évènements des **éléments HTML standards**, des **composants** et des **directives** Angular et même des **Web Components**
- Les méthodes et propriétés utilisées doivent être définies dans la classe

```
<button (click)="handler()"></button> <!-- évènement HTML -->
<button on-click="handler()"></button> <!-- alternative sans () -->
<button data-on-click="handler()"></button> <!-- html5 strict -->

<!-- évènement d'un composant -->
<hero-detail (deleted)="onHeroDeleted()"></hero-detail>
```

Évènements

- **Angular** permet d'accéder à l'évènement via la variable `$event`
- Cet objet peut être utilisé dans l'expression
- Tous les évènements natifs sont propagés vers les éléments parents

  Possibilité de stopper la propagation en retournant `false` dans l'expression qui traite l'évènement
- Les évènements provenant des composants **Angular** ne se propagent jamais
- Exemple d'utilisation de `$event` avec la reproduction d'un ***double binding***

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

Output

- Un composant peut envoyer des évènements
- Annotation `@Output` sur une propriété de type `EventEmitter`
- Le nom de la propriété sera celui de l'évènement à utiliser dans le template

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

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

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

Output

- Possibilité de surcharger le nom de l'évènement

  `@Output('myOtherName')`

- Les noms des évènements sont sensibles à la casse

```
@Component({ selector: 'product-detail', /* ... */ })
export class ProductComponent {
  @Output('add') addToBasket = new EventEmitter<Product>();
}
```

- Pour utiliser ce composant

```
<product-detail (add)="myHandler()">
</product-detail>
```

- **Angular** vérifie les évènements d'un composant
- Il refusera un évènement qui n'existe pas ou non annoté `@Output()`

Output

- L'objet évènement transmis peut être de n'importe quel type
- Il est spécifié dans le paramètre de la classe `EventEmitter`
- Pour émettre un évènement, il faut passer un objet de cette classe

```
@Component({ selector: 'hello-component', /* ... */ })
export class HelloComponent {
  @Output() hello = new EventEmitter<string>();
  constructor() { this.hello.emit('hello!'); }
}
```

- Côté réception de l'évènement, la variable `$event` correspond à cet objet

```
@Component({
  selector: 'main',
  template: '<hello-component (hello)="myHandler($event)"></hello-component>'
})
export class MainComponent {
  myHandler(value) {
    console.log(value); //-> 'hello!'
  }
}
```

Déclaration

- Utilisation des **NgModule** définis en détail plus loin dans la formation
- Pour qu'un composant soit accessible, il faut :
  - qu'il soit dans un autre **NgModule** listé dans la liste des `imports`
  - qu'il soit dans la liste des `declarations` de votre module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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

Projection

- Permet de mettre du contenu HTML dans la balise d'un composant Angular
- Le composant `ng-content` permet de réinserer le contenu dans le template
- Correspond à la directive `ngTransclude` en **AngularJS**

```
<post>
  <h2>Title</h2>
  <p>Content</p>
</post>
```

```
@Component({
  selector: 'post',
  template: `
    <article>
      <ng-content></ng-content>
    </article>
  `
})
export class PostComponent { }
```

Projection

- Possibilité d'avoir plusieurs points d'insertion avec la propriété `select`
- La valeur doit être le sélecteur **CSS** de la section à utiliser

```
<post>
  <h2>Title</h2>
  <p>Content</p>
</post>
```

```
@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 { }
```

Cycle de vie

- Chaque composant a un cycle de vie bien définit
- https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html
- Il est possible d'exécuter du code à chacune de ces étapes
- La plus utilisée est l'initialisation avec l'interface `OnInit`
- L'utilisation d'`OnInit` est recommandé plutôt que celle du constructeur

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

@Component({ selector: 'user', /* ... */ })
export class UserComponent implements OnInit {

  @Input() data: User;
  products: Product[];

  ngOnInit(): void {
    this.products = this.getProducts(this.data.id);
  }

  getProducts(id){ ... }
}
```

Tests

- `TestBed` est l'outil central pour les tests **Angular**
- On l'importe depuis le module `@angular/core/testing`
- Permet de créer un module **Angular** spécifique pour un test
  Utilisation de `TestBed.configureTestingModule({ ... })`
- L'objectif est d'inclure le moins de choses possibles pour isoler le test

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

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

Tests

- Le module créé permet de créer un composant
- Ce composant se présente sous la forme d'un `ComponentFixture`
  - Contient une référence vers l'instance de la classe TypeScript
  - Contient une référence vers l'élément du DOM où il est rattaché

```
class TestBed implements Injector {
  static configureTestingModule(moduleDef: TestModuleMetadata): typeof TestBed
  createComponent(component: Type<T>) : ComponentFixture<T>

  /* ... */
}

class ComponentFixture {
  componentInstance : T
  nativeElement : any
  debugElement : DebugElement
  elementRef : ElementRef
  detectChanges(checkNoChanges?: boolean) : void

  /* ... */
}
```

Tests

- La méthode `detectChanges` permet de piloter la détection de changements
- Attention, pas de détection de changements automatiques

```
import { TestBed } from '@angular/core/testing';
import { TitleComponent } from './title.component';

describe('TitleComponent', () => {
  let fixture
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ TitleComponent ]
    });
    fixture = TestBed.createComponent(TitleComponent);
  });

  it('should have a title', () => {
    const {componentInstance, nativeElement} = fixture;

    componentInstance.title = 'Hello World';
    fixture.detectChanges();
    const h1 = nativeElement.querySelector('h1');
    expect(h1.textContent).toBe('Hello World');
```

Lab 3

Directives

Sommaire

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

Directives

- Schématiquement les directives sont des composants sans template
- Techniquement les composants héritent des directives
- Permet d'intervenir sur l'apparence ou le fonctionnement d'un élément HTML
- **Angular** propose plusieurs directives dans ses différents modules
- Création de directive personnalisée avec l'annotation `@Directive`
- Peuvent accepter des paramètres (`Input`) et émettre des évènements (`Output`)
- Les directives sont l'endroit où faire des manipulation du DOM
  - Les composants peuvent aussi le faire, mais c'est une mauvaise pratique
  - Toujours utiliser le service `Renderer2`, pas avec du code natif

Directives

- Premier exemple de directive
- On utilise traditionnellement un selector sur une propriété `[myProp]`

```
import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({
  selector: '[myHighlight]'
})
export class HighlightDirective {
  constructor(element: ElementRef, renderer: Renderer2) {
    //element.nativeElement.style.backgroundColor = 'yellow';
    renderer.setElementStyle(element.nativeElement, 'backgroundColor', 'yellow');
  }
}
```

- S'utilise dans un template de la façon suivante

```
<p myHighlight>
  Highlight me!
</p>
```

Action utilisateur

- Le ***Host*** est l'élément du DOM qui porte la directive
- Possibilité d'écouter les évènements de l'élément du Host
  - Par l'utilisation de la propriété `host` de l'annotation `@Directive`
  - Par l'utilisation les annotations `HostListener` et `HostBinding`
- Éviter d'écouter des évènement via le DOM pour éviter les fuites mémoires

```
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'; }
}
```

Déclaration

- Fonctionne comme les composants
  - dans un autre **NgModule** listé dans la liste des `imports`
  - dans la liste des `declarations` de votre module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HighlightDirective } from './highlight.directive';

@NgModule({
  declarations: [
    HighlightDirective
  ],
  imports: [
    BrowserModule
  ]
})
export class AppModule {}
```

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`

ngStyle

- Directive permettant d'ajouter des propriétés CSS
- Prend un objet avec les propriétés CSS comme clés
- N'utiliser que pour dans des cas où le pure CSS ne suffit pas

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

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

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

ngClass

- La directive `ngClass` ajoute ou enlève des classes CSS.
- Peut s'utiliser en addition à l'attribut class standard
- Trois syntaxes coexistent
  - `[ngClass]="'class class1'"`
  - `[ngClass]="['class', 'class1']"`
  - `[ngClass]="{'class': isClass, 'class1': isClass1}"`
- La 3e syntaxe est la plus courante
- Elle permet de tout exprimer depuis le template

ngClass

- Exemple d'utilisation de la directive `ngClass`

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

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

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

ngFor

- Permet de dupliquer un template pour chaque élément d'une collection
- Correspond à la directive `ngRepeat` en **AngularJS**
- Définition du contenu à dupliquer dans un élément `<ng-template>`
- Utilisation de la propriété `ngForOf` pour définir la collection
- On crée une variable depuis le template pour l'itérateur
  Nouvelle syntaxe pour créer une variable `let-myVarName`
- Angular met à disposition cinq données supplémentaires
  `index`, `first`, `last`, `even` et `odd`
- Syntaxe finale pour une iterration sur le tableau `items`

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

ngFor microsyntax

- La syntaxe complète pour un ngFor est assez fastidieuse
- **Angular** propose une alternative plus facile à lire
- Cette syntaxe est presque toujours préférée à la syntaxe complète
- **Angular** appelle le système ***Microsyntax***
- Il s'agit purement de sucre syntaxique, le comportement est identique
- Ajout du caractères `*` devant `ngFor` pour indiquer la microsyntax

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

- Noter que le `*ngFor` se trouve directement sur l'élément à dupliquer

ngIf

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

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

- Possibilité de définir un clause `else`

```
<div *ngIf="condition; else elseBlock">...</div>
<ng-template #elseBlock>No data</ng-template>
```

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

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

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>
```

Lab 4

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

- Composants techniques utilisés pour injecter les services
- Nombreux injecteurs qui collaborent
  (Contrairement à **AngularJS** qui n'a qu'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`
- ***Au sein du même injecteur*** les services sont des **singletons**

Configuration globale de l'injecteur

- `@NgModule` a une propriété `providers` pour ajouter les services
- Les services inscrits dans un module sont injectable dans tous les composants de ce module ou d'un module qui `import` ce module

```
// 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 { AppComponent } from './application.components';
import { UserService } from './user.service';

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

Configuration locale de l'Injecteur

- Possibilité d'utiliser la propriété `providers` dans l'annotation `@Component`
- Même syntaxe que la configuration globale
- Les services définit dans un `Component` sont injectables dans ce composant et ses fils
- Déconseillé au profit de l'utilisation des `NgModule`

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

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

Service

- Un service **Angular** n'est rien de plus qu'une classe TypeScript
- Sans annotation, le service ne bénéficie pas de l'injection de dépendance
- Nécessité d'ajouter l'annotation `@Injectable`
- Inutile pour les composants, c'est implicite avec `@Component`

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

@Injectable()
export class UserService {
    constructor(private logger: Logger) { }

    getUsers(): void {
      this.logger.log('getUsers called!');
    }
}
```

Configurer les providers

- Un provider est une description pour l'injecteur :
  comment obtenir une instance de l'élément demandé
- Il est impossible d'utiliser des interfaces dans l'identifiant du provider

```
export function serverConfigFactory(appService: AppService){
  return appService.getConfig();
}

@NgModule({
  providers: [
    UserService, // Le plus simple et le plus courant : une classe
    {
      provide: LoginService, // Pour un élément de ce type
      useClass: LoginServiceImpl // Utiliser cette classe (ou implémentation)
    },
    {
      provide: ServerConfig, // Pour un élément de ce type
      useFactory: serverConfigFactory, // Utiliser une fonction factory
      deps: [ AppService ] // La factory peut elle même avoir des injections
    }
  ]
})
export class AppModule { }
```

Configurer les providers

- Par défaut l'injection se base sur les types des paramètres
- Impossible pour des valeurs tel que des `string` ou `number`
- Possibilité de définir une chaîne de caractère comme identifiant
- Nécessité d'utiliser l'annotation `Inject` pour injecter ce genre de valeurs

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

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

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

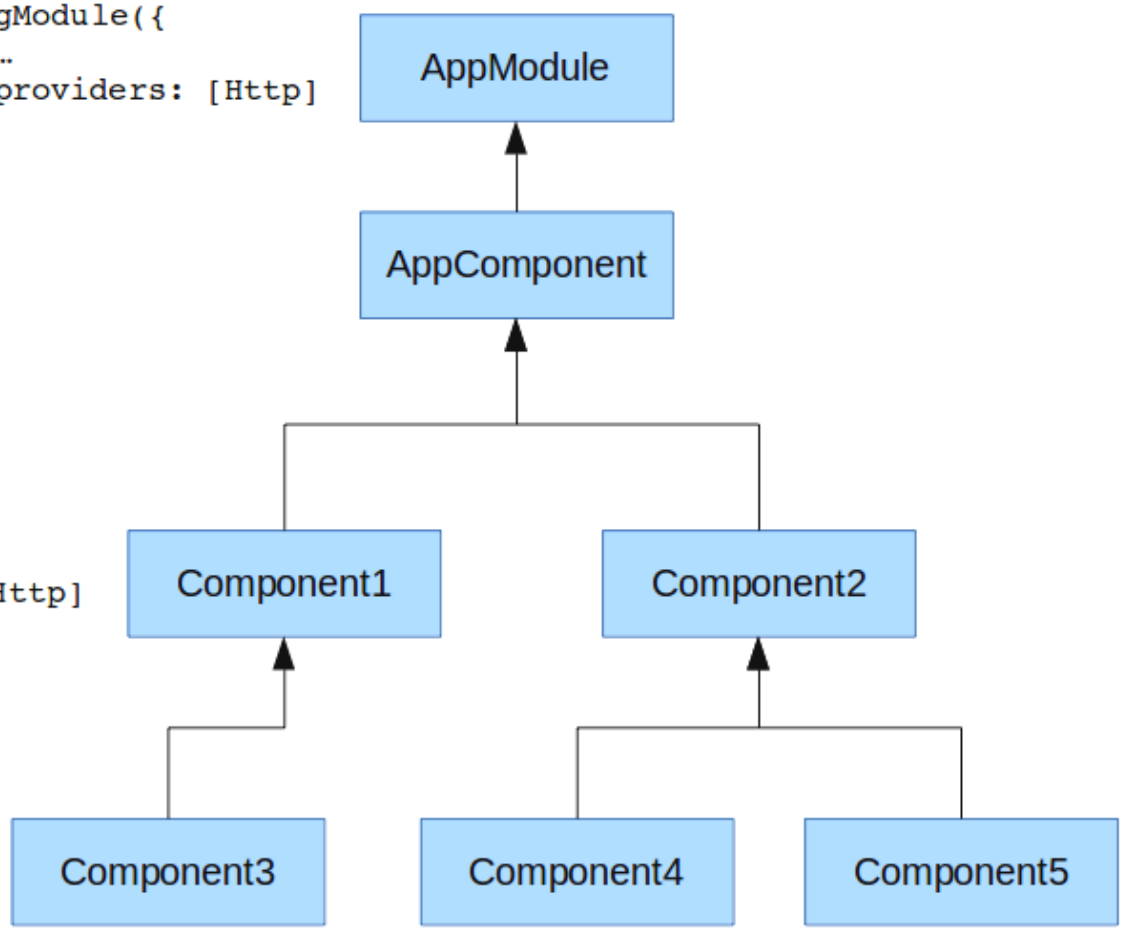
Hiérarchie d'injecteurs

- Chaque injecteur contient un certain nombre de providers
- Chaque injecteur gère un singleton pour chaque provider
- Lors d'une injection de dépendance
  - L'injecteur local essaye de trouver un provider compatible
  - S'il ne trouve pas, il transmet la demande à son parent
  - Ainsi de suite jusqu'à l'injecteur principal de l'application
  - Si aucun provider n'a pu être trouvé, **Angular** affiche une erreur
- Ce mécanisme est très puissant mais peut être complexe
  - Possibilité de faire des surchages locales à des services
  - Mais peut aussi masquer le bon service par inadvertance

Hiérarchie d'injecteurs

```
@NgModule({
  …
  providers: [Http]
})
```

```
@Component({
  …
  providers: [
   provide: Http, useClass: CustomHttp]
})
```



Tests

- Ajouter les `providers` du module de test de `TestBed`
- Ne pas hésiter à surcharger "***mocker***" des services
- Mécanisme puissant qui permet d'isoler l'élément que l'on veut tester
- Deux fonctions utilitaires disponibles :
  - `inject(tokens: any[], fn: Function)`
    injecte des services à la fonction en paramètre
  - `async(fn: Function)`
    retarde automatiquement le test par rapport aux actions asynchrones (fonctionne grâce à ***ZoneJS***)

Tests

- Exemple de test utilisant les providers
- On suppose que `UserService` utilise `LoggerService`

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

class LoggerServiceMock {}

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

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

Lab 5

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 transformation d'une donnée avant son utilisation
- Similaire aux filtres dans **AngularJS**
- Utilisation avec le caractères `|` dans les expressions des templates
- Possibilité d'écrire ses propres **Pipe**
- Ajout de la notion de **Pipe** pure et impure
- Pipes disponibles par défaut dans le framework `@angular/common`
  - `LowerCasePipe` , `UpperCasePipe`
  - `CurrencyPipe`, `DecimalPipe`, `PercentPipe`
  - `DatePipe`, `JSONPipe`, `SlicePipe`
  - `I18nPluralPipe`, `I18nSelectPipe`
  - `AsyncPipe`

Utilisation dans les Templates

- Les **Pipes** disponibles par défaut sont directement utilisables
- Possibilité de chaîner les pipes les uns à la suite des autres
- Possibilité de passer des paramètres avec le caractère `:`
- Les paramètres sont ***bindés*** et le résultat est recalculé à chaque changement
- La syntaxe est la suivante

```
{{ myData | pipeName:pipeArg1:pipeArg2 | anotherPipe }}
```

```
{{ myVar | date | uppercase}}
<!-- FRIDAY, APRIL 15, 1988 -->

{{ price | currency:'EUR':'symbol' }}
<!-- 53.12€ -->
```

Création

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

```
import { isString, isBlank } from '@angular/core/src/facade/lang';
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 Error('MyLowerCasePipe value should be a string');
    }
    return value.toLowerCase();
  }
}
```

Déclarations

- Se déclare comme les composants et les directives
- Le pipe doit être ajouté au tableau `declarations`

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { MyLowerCasePipe } from './mylowercase.pipe';

@NgModule({
  declarations: [
    MyLowerCasePipe
  ],
  imports: [
    BrowserModule
  ]
})
export class AppModule {}
```

Utilisation

- Toujours comme les composants et les directives
- Un pipe est utilisable s'il a été déclaré dans le module ou un module importé

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

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

Injection

- Il est possible d'utiliser un pipe depuis le code TypeScript
- Utilisation de l'injection de dépendances pour utiliser un **Pipe**
- Pas de service `$filter` comme dans **AngularJS**
- Il faut ajouter le pipe dans les `providers` (composant ou module)

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

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

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

Pipes pures

- Fait référence à la notion de fonction pure
- Les **Pipes** sont pure par défaut
- Exécuter uniquement pour un changement de référence de la valeur
- Ne sera pas réévalué pour une mutation sans changement de référence
- Optimise les performances du mécanisme de détection de changement
- N'est pas toujours le comportement souhaité :
  - Ajout / Suppression d'un objet dans un tableau
  - Modification d'une propriété d'un objet

Pipes impures

- Exécuté à chaque cycle du système de détection de changement
- Plus consommateur qu'un pipe pure, n'utiliser que lorsque c'est nécessaire
- Pour définir un **Pipe** impure, mettre la propriété `pure` à `false`

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

AsyncPipe

- Fourni par **Angular** par défaut, exemple de pipe impure
- **Pipe** recevant une `Promise` ou un `Observable` en entrée
- La valeur doit pouvoir changer alors que la référence de la `Promise` ou de l'`Observable` n'a pas changée

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

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

Tests

- Un **Pipe** n'est rien d'autre qu'une fonction !
- 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();
  });

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

Lab 6

Service HTTP

Sommaire

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

RxJS

- **Angular** a une dépendance forte sur la librairie **RxJS 5+**
- Elle est très utilisée dans le coeur du framework
- **RxJS** est une librairie permettant de faire du ***Reactive Programming***
- C'est un nouveau paradigme de programmation très en vogue
- Il en existe de nombreuses implémentations : http://reactivex.io/
- Documentaion pour **RxJS** : https://github.com/ReactiveX/rxjs

Observables

- Les `Observable` sont la notion centrale dans la librairie **RxJS**
- Ils représentent un flux de données, on parle souvent de ***stream***
- Permet le traitement de tâches asynchrones similaires à des tableaux
- Remplace l'utilisation des promesses qu'il y avait dans **AngularJS**
- Apporte des avantages par rapport aux promessses
  - Permet d'avoir des traitements asynchrones retournant plusieurs données
  - Un Observable peut être **cancelable**
  - Propose de nombreux outils pour traiter les données
- Utilisable pour tous les traitements asynchrones

  Requêtes HTTP, WebSocket, gestion des événements

Observables

- **RxJS** fourni une liste importante d'opérateurs pour les `Observable`
- Ces opérateurs s'inspirent largement des transformations sur un tableau
- `take(n)` : pioche les n premiers éléments et coupe le flux
- `filter(fn)` : laisser passer les événements pour lesquels fn rend `true`
- `map(fn)` : applique la fonction fn sur chaque élément et retourner le résultat
- `merge(s1, s2)` : fusionne la source aux observables en argument
- `mergeMap(fn)` : applique fn comme map mais merge les valeurs qui sont des observables
- `debounce(ms)` : retarde et filtre pour n'envoyer un élément que lorsqu'il n'y a pas eu de nouveaux éléments depuis le temps en argument
- Ressource importante pour apprendre les opérateurs : http://rxmarbles.com/

Subscriptions

- Pour écouter le résultat d'un flux, il faut utiliser la méthode `subscribe`
- ***Attention***
  - `subscribe` n'est pas un opérateur, il ne peux pas être chaîné
  - Il rend un objet `subscription` qui permet de stopper l'écoute
  - Un observable qui n'a pas été ***subscribed*** ne ***démarre*** pas
  - Un observable ne peut être écouté qu'une seule fois
- `subscribe` prend trois fonctions en arguments, tous optionnels
  - `next` : Appelé pour chaque élément dans le flux
  - `error` : Appelé pour chaque erreur dans le flux
  - `complete` : Appelé lors de la fermeture du flux

Exemple

- Exemple complet d'utilisation des Observables

```
function getDataFromNetwork(): Observable<SomeClass> {
  /* ... */
}

function getDataFromAnotherRequest(arg: SomeClass): Observable<SomeOtherClass> {
  /* ... */
}

getDataFromNetwork()
  .debounce(300)
  .filter((rep1: SomeClass): boolean => rep1 !== null)
  .mergeMap((rep1: SomeClass): Observable<SomeOtherClass> => {
    return getDataFromAnotherRequest(rep1);
  })
  .map((rep2: SomeOtherClass): string => `${rep2} transformed`)
  .subscribe((value: string) => console.log(`next => ${value}`));
```

Création

- Il existe de nombreux initialiseur à partir d'un tableau par exemple
- Possibilité également d'en créer un via le constructeur

```
import { Observable, Subscriber } from "rxjs";

@Component({ ... })
export class AppComponent {
  private subscriber: Subscriber;

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

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

RxJS et Angular

- **Angular** utilise énormément **RxJS** en interne
- La dépendance est en mode ***peer*** c'est à dire qu'elle est à ajouter en plus
- ***Attention***, il faut la version **5+**, alors que la **4** est encore répendue
- **Angular** expose des objets **RxJS** dans plusieurs cas :
  - Requêtes HTTP
  - Intéraction avec un formulaire
  - Affichage des vues par le **router**
- **ngrx** est un projet qui propose d'étendre l'utilisation d'Rx avec Angular
  - **@ngrx/store**, **@ngrx/devtools**, **@ngrx/router**, ...

HTTP

- **Angular** fournit un module `HttpClientModule` dédié à la communication HTTP
- Ce module contient un ensemble de service pour les requêtes HTTP
- Avant **Angular 4.3**, utilisation du module `HttpModule`
- Se base sur le pattern `Observable`
  - Contrairement à AngularJS qui utilisait le pattern `Promises`
  - Plus grande flexibilité grâce aux différents opérateurs de `RxJS`
- Le point d'entré est le service `HttpClient` accessible via l'injection de dépendance
- Nombreuses configurations pour paramétrer ou transformer les requêtes
- Bonne pratique : implémenter les appels REST dans les services

HTTP - Exemple

- Exemple d'un service utilisant `HttpClient`
- Penser à `import` le `HttpClientModule` dans votre module
- Import de la classe `HttpClient` depuis le module `@angular/common/http`
- Injection du service via le constructeur
- La méthode du service retournera l'observable de la requête `HTTP`

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { Person } from './model/person';

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

  getContacts(): Observable<Person[]> {
    return this.http.get<Person[]>('people.json');
  }
}
```

HTTP - Configuration

- La requête HTTP peut être configurée via un paramètre supplémentaire

```
interface RequestOptionsArgs {
  body?: any;
  headers?: Headers;
  observe?: 'body';
  reportProgress?: boolean:
  withCredentials?: boolean;
  responseType?: ResponseContentType;
}
```

HTTP - Configuration

- `HttpClient` propose également de nombreux raccourcis

```
class HttpClient {
  request(url: string|Request, options?: RequestOptionsArgs): Observable<any>

  get(url: string, options?: RequestOptionsArgs): Observable<any>

  post(url: string, body: any, options?: RequestOptionsArgs): Observable<any>

  put(url: string, body: any, options?: RequestOptionsArgs): Observable<any>

  delete(url: string, options?: RequestOptionsArgs): Observable<any>
  /* ... */
}
```

HTTP - Exemple

- Requête HTTP de type PUT avec surcharge des Headers

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { Contact } from './model/contact';

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

  save(contact: Contact): Observable<Contact> {
    const headers = new HttpHeaders();
    headers.set('Authorization', 'xxxxxxx');

    const requestOptions: RequestOptionsArgs = {
      headers
    };
    return this.http.put(`rest/contacts/${contact.id}`, contact, requestOptions);
  }
}
```

HTTP - Exemple

- Exemple avec l'utilisation d'opérateurs **RxJS**

```
import {Component} from '@angular/core';
import {ContactService} from './contact.service';

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

  constructor(private contactService: ContactService) {
    contactService.getContacts().subscribe(contacts => {
      this.displayedData = contacts;
    });
  }
}
```

HTTP - Exemple

- Exemple utilisant d'avantage d'opérateurs

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Component } from '@angular/core';
import { Project, Person } from './model/';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/mergeMap';

@Component({
  selector: 'app',
  template: `<ul>
    <li *ngFor="let project of (projects$ | async)">{{project.name}}</li>
  </ul>`
})
export class AppComponent {
  projects$: Observable<Project[]>
  constructor(http: HttpClient) {
    this.projects$ = http.get<Person[]>('person.json')
      .mergeMap((person: Person): Observable<Project[]> => getProjects(person))
  }
}
```

HTTP - Intercepteurs

- Possibilité de créer des intercepteurs
- S'appliqueront sur les requêtes et les réponses

```
import {
  HttpInterceptor,
  HttpRequest,
  HttpHandler,
  HttpEvent } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class HeaderInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
{
    const clone = req.clone({ setHeaders: {'Authorization': `token ${TOKEN}`} });
    return next.handle(clone);
  }

}
```

HTTP - Intercepteurs

- Enregistrement de l'intercepteurs via le token `HTTP_INTERCEPTORS` dans la configuration du module

```
import { NgModule } from '@angular/core';
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { HeaderInterceptor } from './header.interceptor';

@NgModule({
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: HeaderInterceptor,
    multi: true,
  }],
})
export class AppModule {}
```

HTTP - Tests

- **Angular** propose un module de test pour le système de requêtage : `HttpClientTestingModule`

```
import { TestBed } from '@angular/core/testing';
import { HttpClientTestingModule } from '@angular/common/http/testing';

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

  /* ... */
});
```

HTTP - Tests

- `HttpTestingController` permet de programmer des requêtes et leurs réponses

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

/* ... */

it('return return 1 user', async(inject([ UserService, HttpTestingController ],
  (userService, http) => {

    const mockedUsers = [{ name: 'Zenika' }];

    userService.getUsers().subscribe((users: User[]) => {
      expect(users.length).toBe(1);
    });

    http.expectOne('/api/users').flush(mockedUsers);
  }
)));
```

Lab 7

Router

Sommaire

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

Router

- **Angular** fournit par défaut un routeur dans un module dédié
- Fonctionnement très différent de `ngRoute` d'**AngularJS**
- Phase de développement mouvementé : 2 refontes majeurs
- `@angular/router` est maintenant fiable et recommandé
- Propose de nombreuses fonctionnalités
  - Gestion des routes imbriquées
  - Possibilité d'avoir plusieurs points d'insertions par routes
  - Système de ***Guard*** permettant de gérer l'autorisation à une route
  - Gestion de routes avec chargement asynchrone

Router

- `@angular/router` est orienté **composant**
- Le principe est d'associer les composants à charger en fonction de l'URL
- Association d'un composant principal avec une URL de votre application
- Création de la configuration à partir de la fonction `RouterModule.forRoot`
- Prend en argument un objet de configuration de type `RouterConfig`
- Utilisation de la directive `RouterOutlet` pour définir un point d'insertion
- Navigation entre les pages via la directive `RouterLink`

Router

- `RouterModule.forRoot(...)` rend un module à importer
- Elle prend en paramètre un objet de type `Routes`
- 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 { }
```

RouterOutlet

- Directive à utiliser via la balise `router-outlet`
- Permet de définir le point d'insertion dans un composant
- Le composant sera inséré en tant qu'enfant de la directive
- Possibilité de nommer le point d'insertion via un attribut `name`
- Nommer les outlets sert lorsqu'on a plusieurs vue pour une même route

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

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

RouterLink

- Permet de naviguer d'une route à une autre
- Utiliser des ***vrais*** liens avec l'attribut href fonctionne aussi
- La directive utilise la méthode `navigate` du service `Router`
- `RouterLink` prend un tableau de ***segments*** du chemin

```
@Component({
  template: `
    <nav>
      <ul>
        <li><a routerLink="contacts">Link 1</a></li>
        <li><a [routerLink]="['contact', 1]">Link 2</a></li>
        <li><a [routerLink]="['contact', id]">Link 3</a></li>
      </ul>
    </nav>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {
  id = 2;
}
```

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);
```

- Le template du composant `ContactComponent` devra contenir un `router-outlet` pour pouvoir insérer les composants `EditCmp` ou `ViewCmp`

Stratégies pour le génération des URLs

- **@angular/router** propose deux stratégies possible pour les URLs
- Les configurations se font par le système d'injection de dépendances
- `PathLocationStrategy` (stratégie par défaut)

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

- `HashLocationStrategy`

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

- `PathLocationStrategy` est la solution recommandée aujourd'hui
  - Si votre application n'est pas déployé à la racine de votre domaine
  - Nécessite d'ajouter un paramétrage : APP_BASE_HREF

Stratégies pour le génération des URLs

- Configurer l'implémentation à utiliser

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

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

- Configurer le contexte de l'application pour `PathLocationStrategy`

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

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

Récupération des paramètres d'URL

- Utilisation du service `ActivatedRoute` et `params`
- L'API est sous forme d'un flux de la valeur des paramètre au cours du temps

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';

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

  ngOnInit() {
    this.route.params.subscribe((params: Params): void => {
      const id = Number(params.id); // Les paramètres sont toujours des string
      /* ... */
    });
  }
}
```

Récupération des paramètres d'URL

- Si vous êtes sur que le paramètre ne pourra pas changer
- La propriété `snapshot` donne les valeurs à un instant T

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

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

  ngOnInit(): void {
    const snapshot: ActivatedRouteSnapshot = this.route.snapshot;
    const id = Number(snapshot.params.id);
    /* ... */
  }
}
```

Cycle de Vie

- Possibilité d'intéragir avec le cycle de vie de la navigation
- Interface `CanActivate` permet d'interdire ou d'autoriser l'accès à une route

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

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

export const routes: Routes = [
  { path: 'admin', component: AdminComponent, canActivate: [ AuthGuard ] }
];
```

Lazy Loading

- Permet de diviser la taille du ***bundle*** JavaScript à charger pour démarrer
- Chaque section du site est isolé dans un `NgModule` différent
- Le module est chargé lorsque l'utilisateur visitera une de ses pages
- Création automatique de `chunk` via **Webpack** grâce à **@angular/cli**
- Configuration du router avec la propriété `loadChildren`
- Bien séparer les éléments (composants, services) de chaque module
- Plusieurs stratégies de chargement
  - `PreloadAllModules` : Pré-charge les modules dès que possible
  - `NoPreloading` : Chargement lors d'une navigation (stratégie par défaut)

Lazy Loading

- Chargement à la demande du module `AdminModule`

```
const routes: Routes = [{
  path: 'admin', loadChildren: './admin/admin.module#AdminModule'
}];

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

- Configuration des routes d'`AdminModule` via la méthode `forChild`

```
const adminRoutes: Routes = [{
  {path: '', component: HomeComponent},
  {path: 'users', component: AdminUsersComponent}
}];

@NgModule({
  declarations: [ AdminHomeComponent, AdminUsersComponent ],
  imports: [ RouterModule.forChild(adminRoutes) ]
})
export class AdminModule { }
```

Lab 8

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

Stratégies de gestion des formulaires

- **Angular** fournit par défaut un module dédié à la gestion de formulaire
- Disponible via le module `FormsModule` dans `@angular/forms`
- Le module propose deux stratégies différentes
- **Template-driven forms**
  - Contrôle du formulaire depuis les templates
  - Binding automatique de variables contenant l'état du formulaire
  - **Solution recommandée et par défaut**
- **Reactive forms** (ou Model-driven forms)
  - Méthode programmatique depuis le contrôleur
  - Recommandé pour certains cas complexe
- La suite de la formation traite uniquement des **Template-driven forms**

Principe général

- S'appuie ou reproduit 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
- Utiliser les fonctionnalité d'`@angular/forms` apporte
  - Le ***binding*** de vos données aux champs de formulaire
  - La gestion de l'état et de la validation des champs

"Banana in the Box"

- Le **2-way data-binding** (par défaut dans AngularJS) est désactivé par défaut
- On peut le reproduire avec les syntaxes qu'on a vu jusque là

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

- **Angular** fournit du sucre syntaxique pour ce besoin récurrent

  (Utilise la directive `ngModel` qu'on verra en détail au chapitre **Formulaires**)
- Première solution

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

- Deuxième solution **Banana in the Box**

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

Persistance des données

- Écouter l'évènement `submit` du formulaire pour traiter le formulaire

```
@Component({
  selector: 'contact-form',
  template: `
    <form (submit)="saveForm()">
      <input type="text" [(ngModel)]="contact.name" name="name">
      <button type="submit">Save</button>
    </form>
  `
})
export class ContactFormComponent implements OnInit {
  contact: Contact;

  constructor(private contactService: ContactService) { }

  ngOnInit(): void {
    this.contactService.load().subscribe(contact => this.contact = contact);
  }

  saveForm(): void {
    this.contactService.save(this.contact);
  }
}
```

Validation

- Par défaut, les navigateurs effectuent les validations nativement
- **Angular** reprend certaines syntaxe mais va bien plus loin
- Les mécanismes natifs vont donc rentrer en conflit avec **Angular**
- ***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>
```

Validation

- Pour gérer la validation **Angular** va gérer un objet `AbstractControl`
  - Sur le formulaire : `FormGroup`
  - Sur chaque champ : `FormControl`
- Le `FormGroup` est une aggrégation de l'état des chacun des `FormControl`
- Un `AbstractControl` contient :
  - L'état : `dirty` / `pristine`, `valid` / `invalid`, `touched` / `untouched`
  - Les erreurs de validation dans la propriété `errors`
- Ces données sont mis à jour automatiquement
- On peut s'en servir dans les templates ou dans le contrôleur

État du formulaire et des champs

- Angular expose 6 propriétés dans un `AbstractControl`
  - `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

    Un élément est considéré `touched` dès que le focus a été pris
- La directive `NgControlStatus` (activée par défaut) gère des classes CSS

  `ng-valid`, `ng-invalid`, `ng-pristine`, `ng-dirty`, `ng-untouched`, `ng-touched`

FormControl

- Angular crée un `FormControl` dès l'utilisation de la directive `ngModel`
- `FormControl` permet également d'accéder à la valeur du champ via la propriété `value`
- On peut l'associer à une propriété du composant
- Nouvelle syntaxe dans le template : ***Template reference variables***
- Associe une référence d'une directive à une variable du composant
- Syntaxe générique : `#myPropertyName="role"`
- Pour ngModel : `#myFormControl="ngModel"`

FormControl

- Exemple avec un `FormControl`

```
@Component({
  selector: 'contact-form',
  template: `
    <form novalidate (submit)="saveForm()">
      <input name="name" type="text" [(ngModel)]="contact.name"
            #nameInput="ngModel" required>
      <span [hidden]="nameInput.valid">Error</span>
      <button type="submit">Save</button>
    </form>
  `
})
export class ContactFormComponent implements OnInit {
  contact: Contact;
  nameInput: FormControl;

  constructor(private contactService: ContactService) { }
  /* ... */
}
```

Validateurs

- Un champ peut posséder un ou plusieurs validateurs
  - Support des validateurs standards HTML5 : `required`, `min`, `max`, `minlength`, `maxlength` et `pattern`
  - Possibilité d'ajouter des validateurs personnalisés
- La propriété `valid` correspond à l'aggregation de l'état des validateurs
- Possibilité d'avoir le détail avec la propriété `errors`

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

Création d'un validateur

- Pour créer validateur personnalisé, implémenter la classe `Validator`

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

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

- Pour utiliser le validateur

```
<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`
- Créée 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 name="myName" type="text" [(ngModel)]="contact.name"
         #nameInput="ngModel" required>

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

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

Lab 9

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

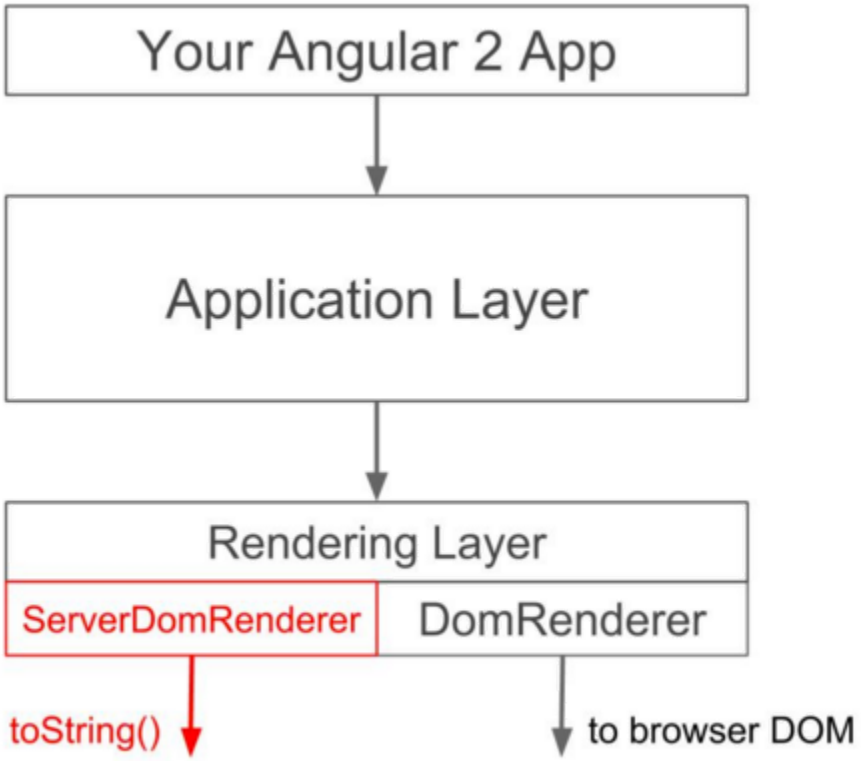
- Indexation par les moteurs de recherche (SEO)
- Prévisualisation (comme dans le partage facebook)
- Amélioration progressive
  - Proposer une version simple pour tous
  - Enrichir l'expérience en fonction du client
- **Accélérer le chargement de l'application**

Angular Universal

- Projet Angular officiel
- Intégré au projet principal depuis Angular 4.0.0
- Contient deux modules
  - Rendu d'une application Angular côté serveur (NodeJS)
  - 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

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
- Utilise 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

- Apperçu de la configuration d'**Angular** dans **Express**

```
const express = require('express');
const ngUniversal = require('@nguniversal/express-engine');

const renderModuleFactory =
  require('@angular/platform-server').renderModuleFactory;

const appServer = require('./dist-server/main.bundle');

const app = express();

app.get('/', function angularRouter(req, res) {
   res.render('index', { req, res });
});

app.use(express.static(`${__dirname}/dist`));

app.engine('html', ngUniversal.ngExpressEngine({
  bootstrap: appServer.AppServerModuleNgFactory
}));
app.set('view engine', 'html');
app.set('views', 'dist');
```