JULIEN BOURNONVILLE
3 AVRIL 2017
M.S. TECHNOLOGIES DU WEB ET CYBER-SÉCURITÉ

Projet Ingénieur : ANGULAR 2

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Sommaire

Liste de figures

Liste d'extraits de code

1. BIBLIOGRAPHIE

1.1. QU'EST CE QU'ANGULAR

Angular est un framework Open Source développé par Google. Il est basé sur le langage JavaScript et est spécialisé sur la partie front-end [1]. Il utilise les possibilités offertes par HTML5 en créant des balises personnalisées.La vision de ce framework est de rendre le code HTML dynamique. On appelle cela un site en "Single Page Applcation" [2].

1.2. L'HISTORIQUE DU FRAMEWORK

Ce framework a commencé son développement en 2009. La première version nommée AngularJS est quant à elle sortie en version 1.0.0 en 2012. Aujourd'hui, la première version d'AngularJS est toujours maintenue et est disponible en version 1.6.3. Aujourd'hui Google propose une nouvelle version de son framework couramment nommée Angular 2. Cette version a été rendue disponible en version 2.0.0 en septembre 2016. La version actuellement en production est la version 2.2.0[2].

Date	Version	Note
2016-09-14	2.0.0	Major Version Release
2016-10-12	2.1.0	Minor Version Release
2016-11-09	2.2.0	Minor Version Release
2016-12-14	4.0.0-beta.0	
2017-03-22	4.0.0 + 2.4.12	Major Version Release

TABLE 1.1 – Release Schedule Angular

Courant le mois de mars 2017, Google va diffuser une nouvelle version d'Angular couramment nommée Angular 4. Elle devrait être disponible à partir du 22 mars 2017. Je détaillerais dans un prochain chapitre les nouvelles fonctionnalités apportées à cette version. Google prévoit la sortie d'une nouvelle version majeure tous les 6 mois.

1. Partie du site web visible de l'utilisateur.
2. Une application web monopage (en anglais single-page application ou SPA) est une application web accessible via une page web unique[1]

1.3. ANGULAR 2, QUOI DE NEUF ?

Tout d'abord, ce qu'il faut savoir c'est que les versions AngularJS et Angular 2 (récemment renommé Angular tout court), n'ont pas grand-chose en commun. En effet l'équipe de développement a décidé de réécrire complètement le coeur de son framework et utilisant des technologies en devenir, nous allons revenir sur ce point.

Cette décision n'est pas sans conséquences. En effet le passage de la version 1.x a la version 2.0 implique une réécriture complète sans compatibilité ascendante du code et nous allons voir pourquoi.

1.3.1 ECMAScript 6

ECMAScript est l'appellation standardisée du JavaScript. Actuellement nous utilisons la version 5 dans nos développements JavaScript. Cepedant, on attend de plus en plus parler d'une nouvelle version du JavaScript nommé ECMAScript 6 ou ECMAScript 2015 ou encore ES6. En réalité, c'est la prochaine version majeure du langage JavaScript.

Mais pourquoi en parle-t-on ici ?

Lors du passage à la version 2 d'Angular les développeurs ont voulu faire bénéficier de l'ES6 à leur framework pour le rendre plus moderne et l'ancrer dans l'avenir du développement web. Même s'il est toujours possible d'utiliser le JavaScript "standard", nous pouvons donc bénéficier des avantages de L4ES pour pour développer en Angular.

ES6, c'est quoi ?

Comme décrit plus tôt ES6 est la prochaine version du JavaScript. Celle-ci a d'ailleurs atteint son état final récemment. Cependant, elle n'est pas encore supportée par tous les navigateurs. Pour pouvoir en bénéficier actuellement il faut utiliser un transpileur qui aura pour but de compiler l'ES6 dans sa version antécédente, c'est-à-dire l'ES5. Les deux outils actuellement les plus répandues sont Traceur[3] et BabelJs[4]. Malheureusement certaine fonctionnalité de l'ES6 ne peuvent pas être transpileur en ES5.

Quoi de nouveau dans l'ES6 ?

L'ES6 n'étant pas l'objet de ce rapport, je ne vais détailler que quelques évolutions qui me paraissent intéressantes.

3. Projet développé par Google
4. Projet développé à l'origine par Sebastian McKenzie (17 ans) et soutenu par une forte communauté et donc plus populaire

Déclaration de variable avec let

En JavaScript, pour déclarer une variable on utilise le mot clé "`var`". Cette déclaration peut poser quelques problèmes. En effet en JavaScript, le concept de hosting[3] peut porter à confusion. Ce concept amène à déclarer la variable en tout début d'une fonction, peut l'endroit où elle a été codée. Voici un exemple pour illustrer :

Extrait de code 1.1 – Déclaration d'un bloc if

```
1   function getUserFullName(user) {
2       if (user.isChampion) {
3         var name = 'Champion ' + user.name;
4           return name;
5       }
6       return user.name;
7   }
```

Extrait de code 1.2 – Déclaration d'un bloc if equivalent

```
1  function getUserFullName(user) {
2    var name;
3      if (user.isChampion) {
4        name = 'Champion ' + user.name;
5        return name;
6      }
7      return user.name;
8  }
```

Avec ES6, un nouveau mot clé est arrivé pour contourner ce problème, c'est `let`. Celui se comporte comme on peut l'attendre, voici un exemple :

Extrait de code 1.3 – Déclaration d'un bloc if avec let

```
1  function getUserFullName(user) {
2      if (user.isChampion) {
3        let name = 'Champion ' + user.name;
4          return name;
5      }
6      // la variable name n'est pas accessible ici
7      return user.name;
8  }
```

Dans le même esprit, l'ES6 apporte la possibilité de créer des constantes avec le mot clé `const`. Comme pour les variables déclarées avec `let`, les constantes ne sont pas hoisted[5] et sont bien déclarée dans leur bloc.

Les classes

Une des nouvelles fonctionnalités importantes apportées par l'ES6 est l'arrivée de l'utilisation des classes en JavaScript.. On pourra ainsi faire plus aisément de l'héritage de classes. Voici un exemple :

5. "remontées"

Extrait de code 1.4 – Une classe avec héritage en ES6

```
class Vehicule {
  speed() {
    return 10;
  }
}

class Car extends Vehicule {
  speed() {
    return super.speed() + 10;
  }
}

const car = new Car();
console.log(car.speed()); // return 20.
```

Arrow Functions

Une des nouveautés de l'ES appréciée par les développeurs est la nouvelle syntaxe arrow functions [6] qui est utilisé principalement pour les callbacks et les fonctions anonymes. Cette syntaxe utilise l'opérateur => pour déclarer la fonction. La particularité des arrow functions est que le `this` reste attaché lexicalement, ce qui signifie que ces arrow functions n'ont pas un nouveau `this` comme les fonctions normales.Voici un exemple de son utilisation :

Extrait de code 1.5 – Exemple d'Arrow Function

```
const maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(element => {
      if (element > this.max) {
        this.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

Voici donc quelques notions importantes pour la suite de ce rapport sur l'utilisation d'Angular 2, mais très minimes quant à la quantité de nouveauté apportée par l'ES6.

1.3.2 TypeScript

Le TypeScript est un langage développé par Microsoft dans le but de sécuriser la programmation JavaScript[4]. C'est un langage libre et open source qui est un surensemble du

6. "fonction fléchée"

JavaScript. Le TypeScript n'est donc pas un langage compris par les navigateurs et nous avons donc besoin d'utiliser un transpileur pour l'utiliser. Le transpileur, comme pour l'ES6,

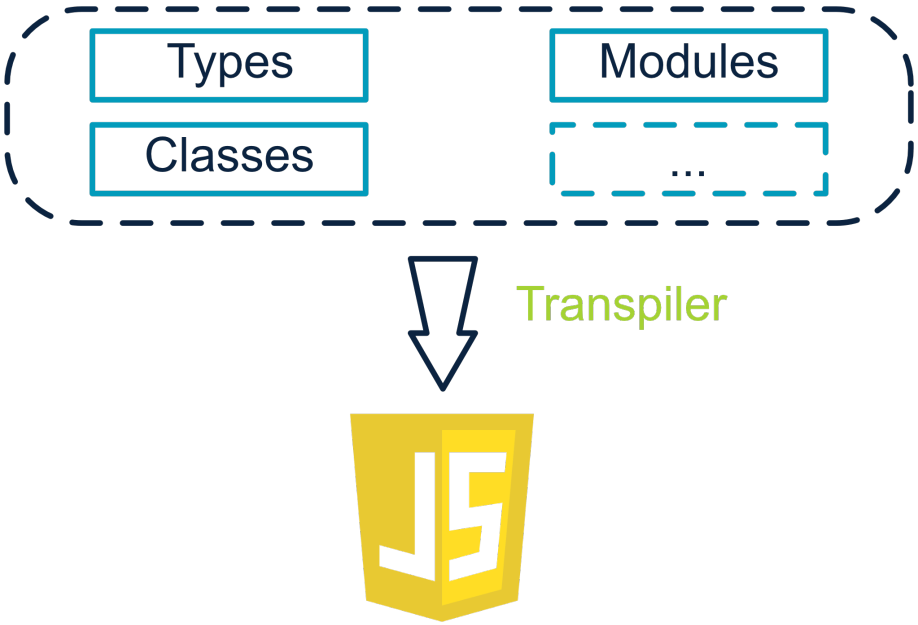

FIGURE 1.1 – Le TypeScript

a pour but de rendre le code compréhensible par les navigateurs actuels. Il le convertit donc en ES3 ou ES5.

Un surensemble de quoi ?

Le TypeScript a pour but d'améliorer la qualité et la sécurité du code JavaScript. Pour cela, il ajoute de nouvelles notions manquantes au JavaScript.

Le typage

Comme le nom du langage le laisse suggérer, Le TypeScript apporte le typage au JavaScript[5]. Ainsi nous allons pouvoir définir un type à nos variables et/ou nos fonctions. Ce type peut être implicite ou explicite. Voici des exemples :

Extrait de code 1.6 – Typage explicite

```
1 let a: number
2
3 function demo(selector: string, options: {ease: string, duration: number}):
     Element{
4     return document.querySelector(selector)
5 }
```

Extrait de code 1.7 – Typage implicite

```
1 let a = 3
```

```
2 a = "Salut les gens" // Type 'string' is not assignable to type 'number'
```

Comme on peut le voir dans l'extrait de code 1.7, ce code n'est pas correct. C'est dans ce type de cas que l'on aborde la notion de sécurité du code. En effet ici, le transpileur donnera une erreur, car il s'attend à avoir un type `number`, défini ici implicitement à la première ligne. Mais à la deuxième ligne, on essaye d'affecter la valeur `"Salut les gens"` qui est de type `string`.

Les classes

Pourquoi reparle-t-on des classesdans le TypeScript alors que nous les avons déjà abordés en ES6 ? En effet nous avions vu dans le chapitre 1.3.1 que l'ECMAScript 6 apportait la gestion des classes par le mot clé `class`. Cependant le TypeScript va encore plus loin. En effet, le langage permet de gérer la visibilité des différentes propriétés et la gestion des méthodes statiques comme en JAVA ou encore en PHP. Voici un exemple de code [5]

Extrait de code 1.8 – Les classes en TypeScript

```
1  class Demo {
2      private factor
3
4      constructor (factor: number) {
5          this.factor = factor
6      }
7
8      public multiplie (n: number): number {
9          return n * this.factor
10     }
11
12     static salut (): string {
13         return 'Salut'
14     }
15 }
```

Tout comme dans les langages orientés objet, il est possible de définir des accesseurs et des mutateurs avec les mots clé `set` et `get`.

Les interfaces

Avec TypeScript, il y a deux manières d'utiliser les interfaces. Il y a ma méthode "standard" connue dans d'autres langages de programmation qui consiste à implémenté une interface à une classe pour quelle en définissent les méthodes établies dans l'interface. La deuxième méthode d'implémentation dans le TypeScript est plus tournée vers l'utilisation des objets en JavaScript. En effet, l'on peut à l'aide de cette méthode définir la structure d'un objet que l'on attend en paramètre d'une fonction par exemple. Voici quelques lignes de code pour mieux comprendre :

Extrait de code 1.9 – Les interfaces en TypeScript

```
interface User {
    username: string;
    password: string;
    confirmPassword?: string; // Propriete optionnelle
}

let user:User;

user = {username: 'max', password: 'supersecret'};

// Les interfaces peuvent aussi contenir des fonctions

interface CanDrive {
    accelerate(speed:number): void;
}

let car:CanDrive = {
    accelerate: function (speed:number) {
        // ...
    }
};
```

Les décorateurs

Les décorateurs ne sont apparus qu'à partir de la version 1.5 de TypeScript et spécialement pour supporter Angular[3]. Le décorateur est une façon de faire de la métaprogrammation, ressemblant aux annotations qui sont utilisées en Java par exemple. La puissance des décorateurs réside dans le fait qu'ils peuvent modifier les paramètres ou encore le résultat retourné. Ils peuvent également appeler d'autres méthodes quand la cible est appelée ou ajoutée de métadonnées, c'est ce qui est notamment utilisé pour l'utilisation avec le framework Angular. Dans TypeScript les décorateurs sont préfixés d'un `@`.

Le bémol de cette solution c'est qu'elle est encore au stade expérimental[6]. Cependant il y a de bonnes chances que cela soit supporté officiellement à l'avenir (peut-être dans ES7 ?).

Extrait de code 1.10 – Exemple de décorateur dans Angular

```
@Component({ selector: 'ns-home' })
class HomeComponent {

  constructor(@Optional() hello: HelloService) {
    logger.log(hello);
  }

}
```

Dans cet exemple le décorateur est `@Component`. Il est ajouté à la classe `Home`. Cela permet a Angular de comprendre au chargement que cette classe est un composant.

Pourquoi utiliser le TypeScript ?

La question que l'on peut se poser quand on découvre le TypeScript est de savoir quel est l'intérêt par rapport au JavaScript, puisque le code TypeScript est transpileur en JavaScript. Tout d'abord, il y a des raisons assez générales et des raisons plus spécifiques a l'utilisation d'Angular. Une première raison est que ce langage est beaucoup moins libertaire que le JavaScript, dans le but d'avoir un code plus sécurité. Le typage permet de vérifier la cohérence des déclarations et des utilisations de variables et de fonction. De plus, la plupart des IDE tirent un avantage important de ce typage pour mettre en avant les erreurs de code ou permettre l'autocompletion. Ensuite le TypeScript utilise l'avantage de l'ES6 et pali au manque du JavaScript actuel en utilisant les classes, les interfaces et encore bien d'autres nouveautés.

Angular se base principalement sur le TypeScript. Si l'on regarde la documentation d'Angular[7], on peut voir qu'il existe une version d'Angular pour JavaScript ou encore pour Dart. Cependant actuellement la version la mieux documentée et la plus soutenue par la communauté et Google est la version écrite en TypeScript.

Il est donc largement préférable d'utiliser le langage TypeScript si vous voulez faire de l'Angular.

1.3.3 WEB COMPONENT

La notion de composant dans le monde du web n'est pas nouvelle, elle était déjà apparue dans d'autres frameworks comme jQuery ou encore dans la première version d'Angular (AngularJS). Ces composants avaient le défaut de nécessité de nombreuse dépendance ce qui rendait le code complexe. Avec Angular 2, Google a voulu corriger le problème en proposant d'avoir des Web Components [7] réutilisables et encapsulés, c'est à dire avec leur propre logique, leur propre affichage. Il repose sur quatre spécifications[3] :

- Custom Elements [8], sont des éléments du DOM créé par le développeur en fonction de ses besoins. Il peut alors créer des balises HTML personnalisées par exemple `<ng-app></ng-app>`.
- Shadow DOM [9] est un moyen d'encapsuler le DOM d'un composant dans le DOM principal. Cela permet de séparer le style et la logique de programmation de l'application globale de celle du composant.
- Le template, spécifié dans un élément `<template>` n'est pas affiché par le navigateur. Son but est d'être à terme cloné dans un autre élément. Ce qui est déclaré à l'intérieur sera inerte : les scripts ne s'exécuteront pas, les images ne se chargeront pas, etc. Son contenu peut être appelé par le reste de la page avec la méthode classique `getElementById()` et il peut être placé sans risque n'importe où dans la page.
- HTML Imports, ils permettent d'importer du HTML dans du HTML. Cela permet un code réutilisable facilement.

7. composants web
8. éléments personnalisés
9. DOM de l'ombre

Il faut noter que ces standards ne sont pas encore tout à fait supportés par les navigateurs, Chrome étant le plus avancé sur ce point.

1.3.4 ANGULAR CLI

Un autre apport de la version 2 d'Angular et l'arrivée d'un outil de gestion de l'application Angular par ligne de commandes. Cet outil, nommé Angular CLI [10], permets de gérer facilement le développement de l'application. Un prérequis est nécessaire pour son utilisation, il faut avoir installé localement NodeJS ainsi que le gestionnaire de paquet NPM [8].

ng new

Le CLI permet tout d'abord la création de l'application en elle même en créant la structure et le fichier standards et nécessaire pour que le framework fonction.

```
ng new my-project
```

Cette commande va donc générer les fichiers dans un dossier nommé `my-project`.

ng serve

Une autre commande importante du CLi est la commande `ng serve`. Celle-ci permet le lancement d'un serveur qui s'exécute en arrière-plan pour compiler l'application et la rendre utilisable en développement. De plus, toutes les modifications seront mises à jour en direct sur le navigateur ce qui est un vrai confort de programmation. L'application est accessible à l'adresse `http://localhost:4200/`

ng generate

La commande ng generate est sûrement la plus utilisée dans le développement Angular. En effet cette commande est centrale, elle permet de générer les principaux éléments sur lesquels nous allons travailler.

```
ng generate class // generer une class
ng generate component // generer un composant
ng generate interface // generer une interface
// ...
```

Cette commande va non seulement générer les différents fichiers, mais va aussi les déclarer dans le fichier de gestion de modules pour les rendre utilisables. Ce point sera détaillé dans la deuxième partie du rapport. C'est donc un vrai confort d'utilisation qui évite d'éditer une multitude de fichiers avant de pouvoir réellement coder le nouvel élément généré

10. Angular Command Line Interface

ng build

La dernière commande que je vais aborder, il y en a plusieurs d'autres[9], c'est `ng build`. Cette commande, comme on peut s'en douter permet de créer l'application finale pour la mise en production. En effet, avec cette commande est créé un dossier dans `dist` dans le projet qui contient l'application compilée en JavaScript et qui pourra être hébergée sur un serveur.

1.4. ANGULAR 4, PROCHAINE VERSION MAJEURE

Comme je l'ai abordé au chapitre 1.2, une nouvelle version d'Angular devrait sortir dans les prochains jours (mars 2017). Cette version sera donc Angular 4.

Mais où est passé Angular 3 ?

Il se trouve qu'actuellement nous utilisons déjà une partie d'Angular 3 sans même le savoir. En effet, le module de routage actuellement utilisé est en version 3.2.3. Pour une question de clarté et d'harmonisation, l'équipe de développement d'Angular a donc décidé de passer directement à la version 4 pour limiter les confusions.

Qu'apporte de nouveau Angular 4 ?

Tout d'abord, l'équipe de développement a tenu à rassurer les développeurs utilisant leur framework, passer de la version 2 a la version 4 demandera bien moins d'effort que lorsque nous étions passés de la version "1" a la version 2[10]. Voici les nouveautés attendues de cette version :

- TypeScript 2.1 pour tirer les avantages de cette nouvelle version.
- Rétro compatibilité avec Angular 2
- Meilleur compileur Angular
- Gain de rapidité
- Réduction de la taille

1.5. D'AUTRES FRAMEWORKS

Il existe d'autres frameworks front-end sur le marché qui prennent le parti du Web Component. En effet, Facebook a notamment développé sa propre solution et l'utilise pour son site phare, il s'agit de ReactJS. EmberJS lui se rapproche de la façon dont fonctionnait AngularJS. Enfin, il existe des frameworks plus récents, tels que VueJS ou encore le petit dernier Aurelia, qui sont plus spécialisés dans la création de petits composants web sans fournir les services autour, comme le fait Angular. Il faut alors passer par des librairies tierces pour ajouter les fonctions que l'on souhaite.

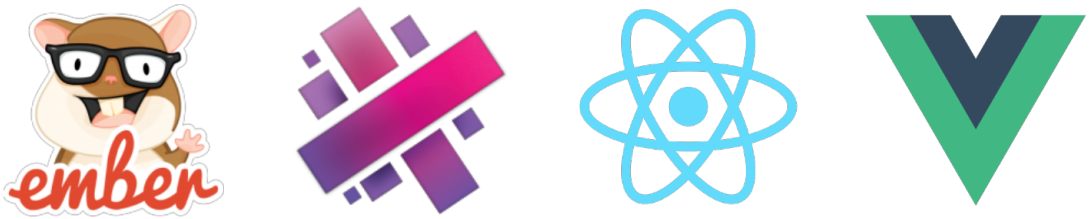

FIGURE 1.2 – EmberJS, Aurelia, ReactJS, VueJS

2. PROJET

2.1. PRÉSENTATION DU PROJET

Afin d'étudier le framework, j'ai décidé de créer une application web pour le mettre en oeuvre et mieux comprendre les mécanismes de fonctionnement. Cette étude portant sur Angular la partie back-end ne sera pas étudiée dans ce rapport.

2.1.1 SUJET LABO

L'application utilisée pour étudier le framework Angular est une application d'enregistrement de facture. Cette application ne gère pas la création du document contractuel en lui même, mais simplement la configuration de la facture pour pouvoir mettre en oeuvre l'utilisation des composants web. La création du document pourrait facilement être générée grâce à une librairie tierce telle que `jsPDF`[11]. Nous pouvons donc créer des clients, créer des produits et créer des factures auxquelles on attribue un client et un ou des produits. Le montant de la facture est ainsi calculé en fonction des produits et de leurs quantités.

2.1.2 BACK-END

Même si nous ne nous concentrerons pas sur la partie back-end dans ce rapport, certaines fonctions comme la protection du routage nécessitent l'utilisation de fonctions serveur comme l'authentification par exemple. Pour cela, j'utilise la solution Firebase de Google qui permet l'utilisation de son API pour mettre en place différents services comme la gestion d'authentification ou encore une base de données NoSQL [1]

2.2. ANGULAR

2.2.1 LES COMPOSANTS

Les composants sont le coeur du fonctionnement d'Angular. En effet, chaque élément fonctionnel peut être décomposé en composant web. Dans notre application, comme on

1. Firebase : https ://firebase.google.com/

peut le voir a la figure 2.1, chaque élément encordé par des pointillés représente une instance d'un composant.

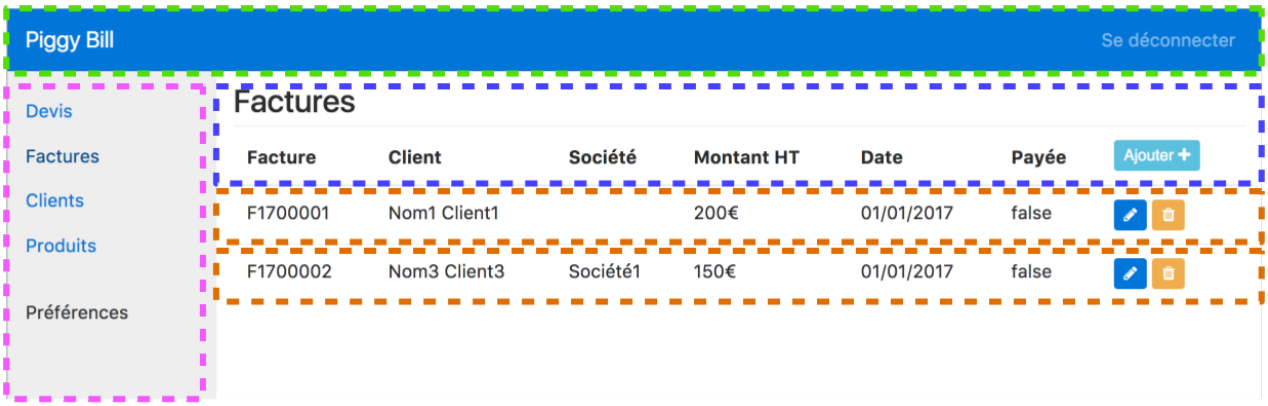

FIGURE 2.1 – Exemple de composant dans l'application

Description

Nous allons voir de quoi est composé un composant. Pour rappel, la commande pour générer un composant en utilisant le CLI est :

```
ng generate component mon-nouveau-composant
```

Cette commande créer les fichiers suivants :

FIGURE 2.2 – Structure de fichier d'un composant

Le premier fichier, `mon-nouveau-composant.component.css` contient la feuille de style propre au composant.

`mon-nouveau-composant.component.html`, contient le template du composant, c'est-à-dire le code HTML qui sera exécuté à chacun des appelles de celui-ci.

Le fichier `mon-nouveau-composant.component.spec.ts` est un fichier TypeScript destiné aux tests unitaires du composant.

Enfin, le fichier `mon-nouveau-composant.component.ts` est le fichier maître du composant, il déclare et contient les méthodes et la logique du composant. Sur ses quatre fichiers, il est possible de n'utiliser que ce dernier. En effet en ajoutant les commandes

`-inline-template` et `-inline-style`[9] on a la possibilité d'intégrer le template HTML et la feuille de style dans le fichier. Cette mise en oeuvre peut être pratique pour les petits composants, car nous n'avons qu'un seul fichier à gérer.

Attardons-nous sur le fichier `mon-nouveau-composant.component.ts` voici ce qu'il contient à sa création :

Extrait de code 2.1 – mon-nouveau-composant.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-mon-nouveau-composant',
5    templateUrl: './mon-nouveau-composant.component.html',
6    styleUrls: ['./mon-nouveau-composant.component.css']
7  })
8  export class MonNouveauComposantComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```

Le fichier commence par les imports des classes nécessaires au fonctionnement de notre application. Ici, le composant étant extrêmement basique, on importe uniquement la classe `Component` pour pouvoir le déclarer en tant que tel, ainsi que la classe `OnInit` dont je vous décrirais l'utilité plus tard.

`@Component` est un déporteur qui permet de configurer le composant pour qu'il soit compris par Angular. Ce décorateur attend un objet comprenant les propriétés de configuration de composant.

`selector` indique à Angular ce qu'il faudra chercher dans nos pages HTML. À chaque fois que le sélecteur défini sera trouvé dans notre HTML, Angular remplacera l'élément sélectionné par notre composant. Dans notre cas, si dans une page HTML est présent le code `<app-mon-nouveau-composant></app-mon-nouveau-composant>`, Angular créera une nouvelle instance de notre composant.

Le paramètre `templateUrl` indique le nom du fichier où se trouve le template HTML pour le composant. Dans le cas de l'utilisation de l'option `-inline-template` ce paramètre est remplacé par `template` qui se présente comme suit :

Extrait de code 2.2 – inline-template

```
1  template: '
2    <h1>Code du template</h1>
3    <p>Lorem ipsum</p>
4  '
```

Enfin, le paramètre `styleUrls` a une fonction équivalente que pour le template à la différence qu'ici on utilise un tableau en paramètre pour avoir la possibilité d'utiliser plu-

sieurs feuilles de style pour un seul composant. En utilisant l'option `-inline-style` on peut également intégrer la feuille de style directement dans le code du composant.

Enfin nous avons la déclaration de la classe `MonNouveauComposantComponent` qui hérite de la classe abstraite `OnInit` qui implique la mise en place de la méthode `ngOnInit()`. Par défaut, le CLI ajoute le constructeur à la classe.

Pour ensuite afficher le composant dans notre application, il suffira d'appeler la balise HTMl correspondante au `selector` définit, par exemple, pour l'extrait , la balise devra être `<app-mon-nouveau-composant></app-mon-nouveau-composant>`.

ngOnInit(), ça sert a quoi ?

La méthode `ngOnInit()` est appelée par l'implémentation de la classe `OnInit`, qui fait partie d'un groupe de méthodes qui réagissent en fonction du cycle de vie du composant. Une notion à bien comprendre est que les entrées d'un composant ne sont pas initialisées dans son constructeur, c'est pour cela que l'on utilise les méthodes suivantes :

- `ngOnChanges()` sera la première méthode appelé lorsque qu'une propriété "bindée" aura été modifiée.
- `ngOnInit()` sera appelée une seule fois après le premier changement (alors que `ngOnChange` est appelée à chaque changement). Cela en fait la phase parfaite pour du travail d'initialisation, comme son nom le laisse à penser. C'est pour cela qu'elle est ajoutée par défaut par le CLI.
- `ngOnDestroy()` est la méthode qui est appelée quand le composant est supprimé du DOM. Cela peut être utilisé pour fermé des connections par exemple.

2.2.2 DataBinding

Le data-binding est une des méthodes faire évoluer dynamique le contenu du DOM, il peut permet notamment de passer des données au template HTML ou encore de réagir à des interactions de l'utilisateur. Voici un schéma pour illustrer.

Interpolation de string

L'interpolation de string permet d'injecter des propriétés d'un composant dans le template HTML en tant que string. Pour cela on utilise la syntaxe avec une double accolade `{{...}}`. Dans celle-ci on indiquera soit le non d'une variable que l'on souhaite afficher ou une méthode qui renvoie un string. Les nombres seront convertis pour être affichés. Voici quelques exemples tirés de l'application.

Extrait de code 2.3 – Interpolation de string

```
1 <div class="card-block">
2   <h4 class="card-title">{{product.designation}}</h4>
3   <p class="card-text">Type : {{product.productType}}</p>
4   <p class="card-text">Prix HT : {{product.priceExTax}} Euros</p>
5     ...
```

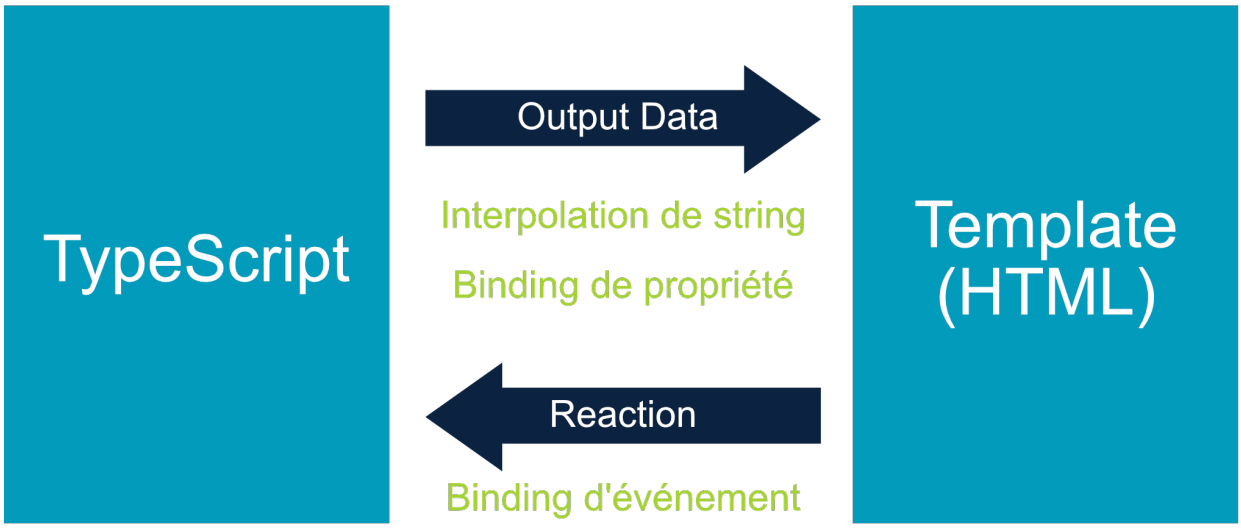

FIGURE 2.3 – Concept de Databinding

```
6 </div>
```

Ici, on a déclaré un objet de type `Product` nommé `product` dans notre fichier TypeScript du composant. Dans l'extrait de code 2.3, on affiche les propriétés `designation`, `productType` et `priceExTax` dans notre template HTML.

Extrait de code 2.4 – Interpolation de string par une méthode

```
1 <div class="form-group col-md-2" formGroupName="{{i}}">
2   <label for="price">Price HT</label>
3   <p id="price">{{getArticlePrice(i)}}</p>
4 </div>
```

Dans cet exemple, on fait appel à une méthode `getArticlePrice()` pour afficher les informations sur le prix d'un article, `i` étant l'index de l'article.

L'interpolation de string est donc la méthode à utiliser pour afficher des données dynamiques dans les templates HTML.

Binding de propriété

Le binding de propriété permet de modifier dynamiquement les attributs des éléments HTML du template. Pour cela, on utilise la syntaxe avec des crochets `[...]` pour déclarer le binding. Grâce à cette approche nous sommes en mesure de gérer de façon dynamique des propriétés tel que `hidden disabled` ou encore `selected` pour les options d'une liste déroulante comme dans l'exemple qui suit.

Extrait de code 2.5 – Binding de propriété

```
1  <div class="form-group row">
2    <label for="client" class="col-2 col-form-label">Client</label>
3    <div class="col-10">
4      <select
5        formControlName="client"
6        class="form-control"
7        type="text"
8        id="client"
9        required>
10       <option *ngFor="let c of clientsList; let i = index"
11               [value]="c"
12               [selected]="isObjectEquals(c, bill.client)">
13         {{c.firstName}} {{c.lastName}}
14       </option>
15     </select>
16   </div>
17 </div>
```

Dans l'extrait de code 2.5 on peut voir a la ligne 11 on bind la valeur `[value]` de l'option et a la ligne 9 on bind la sélection de l'option `[selected]` dans la liste déroulante.

Binding d'événement

Le binding d'événement permet de réagir à des actions de l'utilisateur sur des éléments de notre composant. Cela peut être utile pour exécuter une certaine logique après avoir cliqué sur un bouton ou après avoir modifié un champ de formulaire. Ici, on utilisera la syntaxe avec des parenthèses `(...)`. Il y a de nombreux événements qui peuvent être écouté comme `(click)`, `(keyup)`, `(mousemove)`, `(change)`, etc. Il existe un binding d'événement pour change événement du DOM[2]. Voici un cas d'utilisation.

Extrait de code 2.6 – Binding d'événement

```
1  <button
2    type="button"
3    class="btn btn-primary" (click)="onAddItem(itemProduct.value,
       itemQuantity.value)">+
4  </button>
```

Dans l'extrait de code 2.6, on écoute le clic sur un bouton pour lancer l'exécution de la méthode `onAddItem()`.

Binding bidirectionnel

Le binding bidirectionnel est une combinaison entre le binding de propriété et d'événement avec la syntaxe `[(ngModel)]`. Cela permet donc le transfert de données dans les deux directions, du code TypeScript vers le template HTML et vice-versa.

2. Liste des événements : https ://developer.mozilla.org/fr/docs/Web/Events

2.2.3 LES DIRECTIVES

Les directives sont une des notions incontournables d'Angular. Elles permettent de passer des instructions dans le DOM. C'est donc dans le code HTML que les directives sont mises en place.

ngIf

Comme son nom le laisse deviner, cette directive permet d'apporter une condition à l'affichage d'un élément. Elle fait partie des directives de structure, c'est-à-dire qu'elle modifie la structure du template en fonction de la condition. Voyons un exemple.

Extrait de code 2.7 – ngIf

```
1  <div class="form-group row">
2    <label for="imageUrl">Lien Image</label>
3    <div class="input-group">
4      <span class="input-group-addon"><i class="fa fa-external-link"
           aria-hidden="true"></i></span>
5      <input
6        formControlName="imageUrl"
7        class="form-control"
8        type="text"
9        value="Piggy"
10       id="imageUrl"
11       #imageUrl>
12   </div>
13 </div>
14 <div *ngIf="imageUrl.value" class="form-group row">
15   <div class="img-container">
16     <img class="img-thumbnail mx-auto" [src]="imageUrl.value">
17   </div>
18 </div>
```

Ici, la `div` débutant à la ligne 14 ne sera créée dans le DOM uniquement si l'`input` définit ligne 5 a été complété.

ngFor

Faisant également partie des directives de structure,`ngFor` permet de répéter un élément du DOM en fonction d'un paramètre. Cette directive fonctionne comme un `forEach`. On peut donc lui apporter un tableau en entrée et répéter l'élément du DOM en fonction du nombre d'éléments du tableau. Nous l'avons déjà rencontré dans des extraits de code précédents, notamment l'extrait 2.5. Voici un rappel du code qui nous intéresse.

Extrait de code 2.8 – ngFor

```
1  <select
2        formControlName="client"
3        class="form-control"
4        type="text"
```

```
5       id="client"
6       required>
7       <option *ngFor="let c of clientsList; let i = index"
8               [value]="c"
9               [selected]="isObjectEquals(c, bill.client)">
10        {{c.firstName}} {{c.lastName}}
11      </option>
12    </select>
```

Dans cet exemple, on utilise la directive `ngFor` pour lister les options d'une liste déroulante. À la ligne 7, on utilise le tableau `clientsList` défini en TypeScript. Pour chaque client dans ce tableau, on définit une variable `c` représentant un client unique que l'on peut utiliser pour en extraire les propriétés pour l'affichage de l'option, dans notre exemple `{{c.firstName}} {{c.lastName}}`.

Autres directives

Il existe d'autres directives comme `ngStyle` ou `ngClass` qui sont des directives d'attribut et qui permettent respectivement d'influer dynamique sur le style ou sur les classes de l'élément visé.

Créer ses propres directives

Angular permet également de créer nos propres directives avec la commande :

```
ng generate directive ma-directive
```

Voici un exemple de directive personalisée.

Extrait de code 2.9 – Directive personnalisée

```
1  import { Directive, ElementRef, OnInit } from '@angular/core';
2
3  @Directive({
4    selector: '[appMaDirective]'
5  })
6  export class MaDirectiveDirective implements OnInit {
7
8    constructor(private elementRef: ElementRef)  {
9      this.elementRef.nativeElement.style.backgroundColor = 'green'
10   }
11 }
12
13 //utilisation dans un template :
14 <p appMaDirective> Lorem ipsum </p>
```

Ici, chaque élément du DOM où sera appliqué cette directive, dans l'exemple le paragraphe ligne 14, recevra une couleur de fond verte.

Directives de composant

Nous pouvons également définir des composants en tant que directive en utilisant la même syntaxe au niveau du `selector`.

Extrait de code 2.10 – Directive de composant

```
1  // extrait du composant client-item
2  @Component({
3    selector: '[client-item]',
4    template: `
5      <td>{{client.firstName}} {{client.lastName}}</td>
6      <td>{{client.mail}}</td>
7      <td>{{client.company}}</td>
8      <td>
9        <button type="button" class="btn btn-primary btn-sm" (click)="onEdit()
             "><i class="fa fa-pencil" aria-hidden="true"></i></button>
10       <button type="button" class="btn btn-warning btn-sm" (click)="onDelete
             ()"><i class="fa fa-trash" aria-hidden="true"></i></button>
11     </td>
12   `
13 })
14
15 // utilisation dans le template du composant client-list
16 <table class="table table-hover">
17       <thead>
18         <tr>
19           <th>Nom</th>
20           <th>Adresse mail</th>
21           <th>SociÃ©tÃ©</th>
22           <th><button type="button" class="btn btn-info btn-sm" [routerLink]
                 ="['new']">Ajouter <i class="fa fa-plus" aria-hidden="true">
                 </i></button></th>
23         </tr>
24       </thead>
25       <tbody>
26         <tr *ngFor="let client of clients; let i = index" client-item  [
               client]="client" [clientId]="i"></tr>
27       </tbody>
28     </table>
```

Dans cet exemple, on peut voir que l'on a déclaré la directive ligne 3. Dans le template de `client-list`, chaque ligne du tableau feront appelle a cette directive `client-item` utilisée ligne 26

2.2.4 Services

Angular propose le concept de services, il s'agit de classes que l'on peut injecter dans d'autres. Quelques services sont fournis par le framework, certains par les modules communs, mais l'on peut surtout créer nos propres services. Le seul service fourni par qui est réellement utile est le service `Title`. Celui-ci créera automatique l'élément `title` dans la

section `head` et le rendra dynamique en fonction du composant affiché. Voilà comment on le met en oeuvre.

Extrait de code 2.11 – Service Title

```
1  import { Component } from '@angular/core';
2  import { Title } from '@angular/platform-browser';
3
4  @Component({
5    selector: 'mon-app',
6    viewProviders: [Title],
7    template: `<h1>Mon Composant</h1>`
8  })
9
10 export class PonyRacerAppComponent {
11
12   constructor(title: Title) {
13     title.setTitle('Mon App - Mon Composant');
14   }
15 }
```

Comme on peut le voir dans l'exemple 2.11, les services utilisent l'injection de dépendance pour être mis en oeuvre. Voyons maintenant comment créer nos propres services. Avec le CLI on peut utiliser la commande suivante pour générer un service.

```
ng generate service mon-service
```

Voici un service qui a été créé dans le cadre de l'application en lien avec ce rapport.

Extrait de code 2.12 – Service Client

```
1  import {Injectable} from '@angular/core';
2  import {Client} from "./client";
3
4  @Injectable()
5  export class ClientService {
6
7    private clients: Client[] = [
8      new Client('Client1', 'Nom1', 'client1.nom1@test.fr'),
9      new Client('Client2', 'Nom2', 'client2.nom2@test.fr', 'SociÃ©tÃ©1'),
10     new Client('Client3', 'Nom3', 'client3.nom3@test.fr', 'SociÃ©tÃ©1'),
11   ];
12
13   constructor() {
14   }
15
16   getClients() {
17     return this.clients
18   }
19
20   addClient(client: Client) {
21     this.clients.push(client)
22   }
23
24   deleteClient (client: Client){
25     this.clients.splice(this.clients.indexOf(client), 1)
```

```
26    }
27
28    getClient(clientId: number) {
29      return this.clients[clientId];
30    }
31
32    editClient(newClient: Client, oldClient: Client) {
33      this.clients[this.clients.indexOf(oldClient)] = newClient;
34    }
35  }
```

Dans Angular, il est recommandé d'utiliser les services pour interagir avec les données. Pour notre application nous utilisons des données uniquement définies en local "pour l'exemple" comme on peut le voir ligne 7. Une méthode avec une requête HTTP pourra être mise en place facilement, mais la gestion de la partie serveur et du stockage des données n'est pas l'objet de ce rapport. Nous reviendrons tout de même sur les requêtes HTTP pour voir comment elles sont implémentées dans Angular.

Attention pour que les services soient opérationnels, il faut les injecter au niveau de chaque composant qui les utilisent au niveau du paramètre `providers` comme dans l'exemple qui suit.On peut également les injecter au plus au niveau possible pour le rendre disponible à tous les composants dans le fichier de configuration d'Angular `app.module.ts`.

Extrait de code 2.13 – Injection d'un service dans un composant

```
1  @Component({
2    selector: 'app-client-edit',
3    templateUrl: './client-edit.component.html',
4    providers: [ClientService]
5  })
```

Pour instancier le service il suffira ensuite d'y faire appel dans le constructeur du composant, comme dans l'extrait suivant ligne 2.

Extrait de code 2.14 – Instanciation d'un service dans un composant

```
1  constructor(private route: ActivatedRoute,
2              private clientService: ClientService,
3              private formBuilder: FormBuilder,
4              private router: Router) {
5    }
```

2.2.5 Routage

Routes

Dans Angular, la notion de routage permet de naviguer entre plusieurs composants en se basant sur l'URL. Aujourd'hui, il n'existe pas de commande dans le CLI pour générer un fichier pour gérer le routage. Cela s'explique par le fait qu'il existe plusieurs façons différentes de faire, soit en paramétrant directement les routes dans le fichier de configuration

d'Angular `app.module.ts`, c'est ce qui est démontré dans la documentation. Il est également possible de créer un fichier séparé et c'est la démarche que je vais vous exposer.

Il est intéressant de pouvoir gérer les routes en fonction des fonctionnalités de l'application, par exemple un fichier pour en lien avec la gestion `Clients`, un autre pour la gestion `Produits`, etc. Il en faut aussi nécessairement un global au niveau de la racine de l'application.

Voici comment se compose un fichier de routage.

Extrait de code 2.15 – Fichier de routage app.routing.ts

```
1  import ...
2
3  const APP_ROUTES: Routes = [
4    {path: '', redirectTo: '/dashboard', pathMatch: 'full'},
5    {path: 'products', component: ProductsComponent, children: PRODUCT_ROUTES
       , canActivate:[AuthGuard]},
6    {path: 'clients', component: ClientsComponent,  children: CLIENT_ROUTES},
7    {path: 'bills', component: BillsComponent,  children: BILL_ROUTES},
8    {path: 'quotes', component: QuotesComponent},
9    {path: 'dashboard', component: DashboardComponent, canActivate:[AuthGuard
       ]},
10   {path: 'signin', component: SigninComponent},
11   {path: 'signup', component: SignupComponent},
12 ];
13
14 export const routing = RouterModule.forRoot(APP_ROUTES);
```

Ces fichiers sont assez simples à comprendre et à mettre en oeuvre. Ils se composent principalement :

- d'une déclaration de constante qui est un tableau de type `Routes`
- de la définition des différentes routes qui comprend plusieurs paramètres
  1. `path` qui définit la syntaxe de l'URL
  2. `redirectTo` qui définit une URL de redirection
  3. `pathMatch` qui peut être utilisé pour définir si on veut une correspondante complète ou partielle de l'URL, indispensable dans le cas de l'accès à la racine du chemin.
  4. `component` qui définit le composant à afficher
  5. `children` qui définit la variable des routes enfant
  6. `canActivate` qui permet la restriction d'accès par exemple en fonction d'une authentification.
- de l'export d'une constante qui sera importée dans le fichier de configuration d'Angular `app.module.ts` pour rendre le routage fonctionnel

Les fichiers des routes enfants diffèrent très peu de hormis l'absence de l'export de constante, comme vous pouvez le voir à l'annexe 1-A

Pour afficher les composants dynamiquement en fonction de la route, on remplacera la balise définit dans le `selector` du composant par une balise générique propre au router `<router-outlet></router-outlet>`

La redirection à partir de lien hypertexte ou de bouton utilise un binding d'événement propre au router. En effet, on pourra ajouter le paramètre `[routerLink]` pour définir l'URL à atteindre. Voici le code du composant `sidebar`.

Extrait de code 2.16 – routerLink

```
<ul class="nav nav-pills flex-column">

  <li class="nav-item">
    <a class="nav-link" [routerLink]="['/bills']">Factures</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" [routerLink]="['/clients']">Clients</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" [routerLink]="['/products']">Produits</a>
  </li>

</ul>

<ul class="nav nav-pills flex-column">
  <li class="nav-item">
    <a class="nav-link" [routerLink]="['/demo']">Demo</a>
  </li>
</ul>
```

Guards

Comme nous l'avons abordé plus tôt, il est possible de limiter l'accès aux différentes URL. Pour cela on met en oeuvre les `Guards` qui en fonction d'une logique définit vont autoriser ou non l'accès aux URL. Cela peut être utilisé pour des accès avec authentification ou pour la gestion des back-office. Voilà comment se présente un fichier de définition d'un `Guard`.

Extrait de code 2.17 – Guard

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

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {
  }

  canActivate(next: ActivatedRouteSnapshot,
              state: RouterStateSnapshot): Observable<boolean> | boolean {
```

```
13      return this.authService.isAuthenticated();
14    }
15  }
```

Comme on peut le voir dans cet extrait, les `Guard` doivent hériter de l'interface `CanActivate`. C'est grâce à la méthode du même nom qu'il sera possible de filtrer les accès. Dans l'exemple, on teste si l'utilisateur est connecté avec la méthode `isAuthenticated()` du service `AuthService`. Si cette méthode renvoie `true`, l'accès est autorisé.

2.2.6 FORMULAIRES

Angular nous propose une façon élégante d'écrire nos formulaires. En fait, il en propose même deux !

On peut écrire un formulaire en utilisant seulement des directives dans un template : c'est la façon "pilotée par le template". Cette méthode est particulièrement utile pour des formulaires simples, sans trop de validation.

L'autre façon de procéder est la façon "pilotée par le code", on écrit une description du formulaire dans un composant, puis on utilise ensuite des directives pour lier ce formulaire aux `inputs/textareas/selects` du template. C'est plus verbeux, mais aussi plus puissant, notamment pour faire de la validation personnalisée, ou pour générer des formulaires dynamiquement. Dans l'application, je n'ai utilisé que cette deuxième méthode, car c'est aussi la plus répandue.

Quelleque soit la méthode choisie, un champ de formulaire sera représenté par un `FormControl` en Angular. L'intérêt c'est que ce `FormControl` possède des attributs :

- `valid` : si le champ est valide, au regard des contraintes et des validations qui lui sont appliquées.
- `errors` : un objet contenant les erreurs du champ.
- `dirty` : est `false` jusqu'à ce que l'utilisateur modifie la valeur du champ.
- `pristine` : l'inverse de `dirty`
- `touched` : est `false` jusqu'à ce que l'utilisateur soit entré dans le champ.
- `untouched` : l'inverse de `touched`
- `value` : la valeur du champ
- `valueChanges` : un élément observable qui émet à chaque changement sur le champ.

Ces contrôles peuvent être regroupés dans un `FormGroup` ("groupe de formulaire") pour constituer une partie du formulaire avec des règles de validation communes. Les `FormGroup` ont les mêmes propriétés qu'un `FormControl`, avec quelques différences :

- `valid` : si tous les champs sont valides, alors le groupe est valide.
- `errors` : un objet contenant les erreurs du groupe, ou null si le groupe est entièrement valide. Chaque erreur en constitue la clé, et la valeur associée est un tableau contenant chaque contrôle affecté par cette erreur.
- `dirty` : est `false` jusqu'à ce qu'un des contrôles devienne "dirty".
- `pristine` : l'inverse de `dirty`
- `touched` : est `false` jusqu'à ce qu'un des contrôles devienne "touched".

- `untouched` : l'inverse de `touched`
- `value` : valeur du groupe sous forme d'un objet dont les clé/valeurs sont les contrôles et leur valeur respective.
- `valueChanges` : un élément observable qui émet à chaque changement sur un contrôle du groupe.

Les extraits de code 2.18 et 2.19 font partie du code du composant `client-edit`. Vous pouvez retrouvez le code dans son intégralité en annexe 2-A et 2-B.

Extrait de code 2.18 – Formulaire édition Client - TypeScript

```
1  export class ClientEditComponent implements OnInit {
2    ...
3
4    private clientForm: FormGroup;
5    private clientEditModeLabel: string;
6
7    constructor(private route: ActivatedRoute,
8                private clientService: ClientService,
9                private formBuilder: FormBuilder,
10               private router: Router) {
11   }
12
13   ...
14
15   clearForm() {
16     this.clientForm.reset();
17   }
18
19   ...
20
21   private initForm() {
22     ...
23
24     this.clientForm = this.formBuilder.group(
25       {
26         firstName: [clientFirstName, Validators.required],
27         lastName: [clientLastName, Validators.required],
28         mail: [clientMail, Validators.required],
29         company: [clientCompany]
30       }
31     );
32   }
33 }
```

Dans l'extrait de code 2.18, nous sommes dans le code TypeScript du composant. On retrouve des notions liées à la définition du formulaire dans plusieurs endroits. Tout d'abord en ligne 4 on définit un objet `FormGroup` qui nus permettra dans exploiter les propriétés. Ensuite, ligne 9, dans le `constructor` on instancie la classe FormBuilder pour pouvoir définir les composants du formulaire. Enfin, à partir de la ligne 24, on définit les champs du formulaire, et on leur affecte des paramètres, par exemple ici `clientFirstName` qui représente le paramètre `value` et `Validators.required` qui permet de faire la validation du champ en indiquant qu'il est obligatoire. Il existe d'autres façons d'exploiter cette classe

`Validator` en utilisant par exemple du `regex`.

Extrait de code 2.19 – Formulaire édition Client - Template

```
1  <form [formGroup]="clientForm" (ngSubmit)="onSubmit()">
2    <div class="form-group row">
3      <label for="firstName" class="col-2 col-form-label">PrÃ©nom</label>
4      <div class="col-10">
5        <input
6          formControlName="firstName"
7          class="form-control"
8          type="text"
9          value="John"
10         id="firstName"
11         required>
12     </div>
13   </div>
14
15   ...
16
17   <button type="submit" class="btn btn-primary" [disabled]="!
       clientForm.valid">Enregistrer</button>
```

Dans le template 2.19, il ne reste plus qu'à faire la relation entre ce qui a été codé dans le TypeScript et les champs du formulaire. Ligne 1 on retrouve le `FormGroup` définit ligne 4 de l'extrait précédent. Ensuite pour faire la relation avec les champs il faut utiliser le paramètre `formControlName`.

2.3. POUR ALLER PLUS LOIN

Requêtes HTTP

Les requêtes HTTP n'ont pas été abordées dans le détail dans ce rapport, elle n'en reste pas moins importante pour toute application web qui se respecte. Cependant elles sont assez faciles à mettre en place dans un système. Voici à quoi cela pourrait ressembler :

Extrait de code 2.20 – Requetes HTTP

```
1  storeClients(clients: Client[]){
2    const headers = new Headers({'Content-Type': 'application/json'});
3    return this.http.put('https://piggy-bill.firebaseio.com/data.json',
4        clients,
5        {headers: headers}
6    );
7  }
8
9  downloadClients(){
10   return this.http.get('https://piggy-bill.firebaseio.com/data.json').map(
11       (response: Response) => {
12         console.log(response.json());
13         return response.json()
14       }
```

```
15    )
16  }
```

Ensuite il suffira d'intégrer ces méthodes à nos composants.

Pipes

Les pipes dans Angular permettent traitées les données brutes pour par exemple filtrer un texte, formater un nombre, tronquer un mot, etc. Il en existe des prédéfinis comme le `slice`, le `json`, le `uppercase` ou encore le `currency`. Il est également possible de créer ses propres pipes pour répondre à des demandes particulières. Voici un exemple avec le currency.

Extrait de code 2.21 – Pipe

```
1  <p id="price">{{article.controls.product.value.priceExTax*quantity.value |
       currency:'EUR':true:'1.2-2'}}</p>
```

AppModule

Nous n'avons pas beaucoup parlé de l'`AppModule` ou plutôt du fichier `app.modules.ts` qui est un des fichiers sur lequel repose le fonctionnement d'Angular. En effet en utilisant le CLI, il est très peu nécessaire d'intervenir dessus. Cependant à chaque composant créé, plusieurs lignes sont ajoutées à ce fichier.

Il est possible en le retravaillant et notamment en le scindant en plusieurs modules d'améliorer les performances de l'application. C'est donc un point à ne pas négliger pour une application destinée à être mise en production.

MEAN Stack

Enfin, à titre personnel, dans le but de pouvoir gérer une application complète, font-end et back-end, je m'intéresserais à la stack MEAN qui est l'utilisation conjointe des technologies MongoDB, ExpressJs, Angular et NodeJs.

Cette stack est assez réputée sur internet lorsque l'on utilise Angular et permet la prise en main d'une application de A à Z.

Annexe 1 – Routage

A. FICHIER CLIENT.ROUTING.TS

```
1 import {Routes} from "@angular/router";
2 import {ClientEditComponent} from "./client-edit/client-edit.component";
3
4 export const CLIENT_ROUTES : Routes = [
5   {path: 'new', component: ClientEditComponent},
6   {path: ':id/edit', component: ClientEditComponent}
7 ];
```

Annexe 2 – Composant d'édition client

A. FICHIER CLIENT-EDIT.COMPONENT.TS

```
1  import {Component, OnInit, OnDestroy} from '@angular/core';
2  import {Subscription} from "rxjs";
3  import {Client} from "../client";
4  import {FormGroup, FormControl, Validator, Validators, FormBuilder} from "
       @angular/forms";
5  import {ActivatedRoute, Router} from "@angular/router";
6  import {ClientService} from "../client.service";
7
8  @Component({
9    selector: 'app-client-edit',
10   templateUrl: './client-edit.component.html'
11 })
12 export class ClientEditComponent implements OnInit {
13   private subscription: Subscription;
14   private clientId: number;
15   private client: Client;
16   private isNew: boolean = true;
17   private clientForm: FormGroup;
18   private clientEditModeLabel: string;
19
20   constructor(private route: ActivatedRoute,
21               private clientService: ClientService,
22               private formBuilder: FormBuilder,
23               private router: Router) {
24   }
25
26   ngOnInit() {
27     this.subscription = this.route.params.subscribe(
28       (params: any) => {
29         if (params.hasOwnProperty('id')) {
30           this.isNew = false;
31           this.clientEditModeLabel = "Edition client";
32           this.clientId = +params['id'];
33           this.client = this.clientService.getClient(this.clientId);
34         } else {
35           this.isNew = true;
36           this.clientEditModeLabel = "Ajout client";
37           this.client = null;
38         }
39         this.initForm();
40       }
41     )
42   }
43
44   ngOnDestroy() {
45     this.subscription.unsubscribe()
46   }
47
48   onSubmit() {
```

```
    if (this.isNew) {
      // Add mode
      this.clientService.addClient(this.clientForm.value);
      this.clearForm();
    } else {
      // Edit mode
      this.clientService.editClient(this.clientForm.value, this.client);
    }
  }

  clearForm() {
    this.clientForm.reset();
  }

  onCancel() {
    this.router.navigate(['clients']);
  }

  private initForm() {
    let clientFirstName = '';
    let clientLastName = '';
    let clientMail = '';
    let clientCompany = '';

    if (!this.isNew) {
      clientFirstName = this.client.firstName;
      clientLastName = this.client.lastName;
      clientMail = this.client.mail;
      clientCompany = this.client.company;
    }

    this.clientForm = this.formBuilder.group(
      {
        firstName: [clientFirstName, Validators.required],
        lastName: [clientLastName, Validators.required],
        mail: [clientMail, Validators.required],
        company: [clientCompany]
      }
    );
  }
}
```

B. FICHIER CLIENT-EDIT.COMPONENT.HTML

```
<h3>{{clientEditModeLabel}}</h3>
<form [formGroup]="clientForm" (ngSubmit)="onSubmit()">
  <div class="form-group row">
    <label for="firstName" class="col-2 col-form-label">Prénom</label>
    <div class="col-10">
      <input
        formControlName="firstName"
        class="form-control"
        type="text"
        value="John"
```

```
11          id="firstName"
12          required>
13      </div>
14    </div>
15    <div class="form-group row">
16      <label for="lastName" class="col-2 col-form-label">Nom</label>
17      <div class="col-10">
18        <input
19          formControlName="lastName"
20          class="form-control"
21          type="text"
22          value="Doe"
23          id="lastName"
24          required>
25      </div>
26    </div>
27    <div class="form-group row">
28      <label for="mail" class="col-2 col-form-label">Email</label>
29      <div class="col-10">
30        <input
31          formControlName="mail"
32          class="form-control"
33          type="email"
34          value="bootstrap@example.com"
35          id="mail"
36          required>
37      </div>
38
39    </div>
40    <div class="form-group row">
41      <label for="company" class="col-2 col-form-label">Société</label>
42      <div class="col-10">
43        <input
44          formControlName="company"
45          class="form-control"
46          type="text"
47          value="Doe&Co"
48          id="company">
49      </div>
50    </div>
51    <button type="submit" class="btn btn-primary" [disabled]="!
        clientForm.valid">Enregistrer</button>
52    <button type="button" class="btn btn-primary" (click)="onCancel()">Annuler
        </button>
53  </form>
54  <hr>
```

Références

[1] Wikipedia. Single-page application — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Single-page%20application&oldid=767035182, 2017. [Online ; accessed 15-March-2017].

[2] Google. Angular Release Schedule. https://github.com/angular/angular/blob/master/docs/RELEASE_SCHEDULE.md, 2017.

[3] Ninja-Squad. *Deviens un Ninja avec Angular*. 2017. Ebook version 1.5.

[4] Wikipedia. TypeScript — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=TypeScript&oldid=770407028, 2017. [Online ; accessed 16-March-2017].

[5] Grafikart. Typescrpit. https://www.grafikart.fr/tutoriels/javascript/typescript-781, 2016. [Online ; accessed 16-March-2017].

[6] Microsoft. Typescript handbook - decorators. https://www.typescriptlang.org/docs/handbook/decorators.html.

[7] Google. Angular Docs. https://angular.io/docs/ts/latest/, 2017.

[8] Google. Angular cli. https://cli.angular.io/.

[9] Google. Angular cli documentation. https://github.com/angular/angular-cli/wiki.

[10] NG BE. Igor minar - opening keynote - version 4 announcement - ng-be 2016. https://youtu.be/aJIMoLgqU_o.

[11] MrRio. GitHub Repo de jsPDF. https://github.com/MrRio/jsPDF, 2017.

Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 3
France
+33 (0)2 29 00 11 11
www.imt-atlantique.fr

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom