

Développement d'application avec Angular

Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES

par :

Bruno REGADAS

Conseiller au travail de Bachelor :

Monsieur Rolf Hauri, enseignant HES

Petit-Lancy, 17 juillet 2023

Haute École de Gestion de Genève (HEG-GE)

Filière informatique de gestion

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre « Bachelor of science HES-SO en informatique de gestion ».

L'étudiant a envoyé ce document par email à l'adresse remise par son conseiller au travail de Bachelor pour analyse par le logiciel de détection de plagiat URKUND, selon la procédure détaillée à l'URL suivante : <https://www.orkund.com> .

L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul< e > le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Petit-Lancy, le 04 mars 2023

Bruno REGADAS

Remerciements

Premièrement, je tiens à remercier fortement Monsieur Rolf Hauri qui m'a aidé à trouver le sujet de mon travail de bachelor et qui m'a suivi tout au long de ce projet en me guidant et en m'apportant de précieux conseils.

Ensuite, je tiens à fortement remercier madame Francine Quennoz et monsieur Pascal Duriaux pour la relecture de mon travail.

Je tiens aussi à remercier mon futur juré qui prendra de son temps pour lire mon travail et assister à ma soutenance ainsi que pour le jugement de cette dernière.

Résumé

L'objectif de ce travail est d'apprendre les concepts principaux d'Angular permettant de mener à bien un projet complet en utilisant ce Framework.

Le début de ce travail raconte l'histoire du Framework, cela permet d'en apprendre davantage sur l'origine de celui-ci. J'aborde aussi de façons plus rapides ce qui touche indirectement à Angular comme TypeScript, Node.js et autre.

Ensuite, je parcours en théorie et en pratique les différents concepts d'Angular. Le but étant de démontrer l'utilité de chacun d'eux afin de comprendre pourquoi il est important de les mettre en place. Puis, je démontre en pratique avec un cas concret comment implémenter ces différents concepts en utilisant ce que j'ai mis en place dans une application développée par moi-même nommée « LiveBall. »

Le but est d'être capable de concevoir une application de la génération de cette dernière au déploiement de celle-ci en ayant utilisé les concepts principaux qu'offre Angular. Les derniers points abordent des sujets tels que le déploiement de la solution ou la mise en place de tests.

Table des matières

Déclaration.....	i
Remerciements	ii
Résumé	iii
Liste des tableaux	vii
Liste des figures.....	vii
1. Introduction.....	1
2. Angular	2
2.1 Présentation	2
2.2 Qu'est-ce qu'un Framework ?	2
2.2.1 Définition.....	2
2.2.2 L'avantage d'utiliser un Framework.....	3
2.3 TypeScript	3
2.3.1 Présentation.....	3
2.3.2 Comment fonctionne TypeScript ?	4
2.4 Single page applications (SPA).....	4
2.4.1 Définition.....	4
2.4.2 Avantages et inconvénient	5
3. Configuration de l'environnement de développement.....	6
3.1 Node JS	6
3.1.1 Présentation.....	6
3.1.2 NPM (Node Package Manager)	7
3.1.3 Utilisation	7
3.2 IDE utilisable	8
3.2.1 Présentation.....	8
3.2.2 Exemples d'IDE	8
3.3 Angular CLI.....	8
4. Configuration du projet Angular	10
4.1 Génération du projet.....	10
4.1.1 Architecture de dossier du projet.....	11
4.1.2 Autres fichiers	12
4.2 Premier démarrage	13
4.2.1 Composant racine	13
4.2.1.1 Description du contenu d'un composant racine	13
4.2.2 Module racine	14
4.2.2.1 Description du contenu d'un module racine	15
4.2.3 Configuration de TypeScript.....	16
4.2.4 Commande pour le démarrage de l'application	17

5. Conception d'une application Angular	18
5.1 Présentation de l'application.....	18
5.2 Les composants web	18
5.2.1 Présentation	18
5.2.2 Cycle de vie d'un composant	19
5.2.2.1 Exemple d'interaction avec le cycle de vie.....	19
5.2.3 Les méthodes	20
5.2.3.1 Syntaxe.....	20
5.2.4 Les objets métiers	21
5.2.5 Les injections de dépendances	22
5.2.6 Les templates.....	22
5.2.6.1 L'interpolation	23
5.2.6.2 Les événements	24
5.2.6.3 Les variables référencées dans le template.....	25
5.2.6.4 Générer un composant.....	25
5.3 Le routing	26
5.3.1 Qu'est-ce que le routing Angular	26
5.3.2 L'implémentation du routing	27
5.3.3 La balise <router-outlet>	28
5.3.4 La gestion des routes par Angular.....	29
5.3.5 Les routes paramétrées	29
5.3.6 Utilisation de notre routing.....	30
5.3.7 L'organisation de nos routes	31
5.4 Les pipes	32
5.4.1 Présentation des pipes.....	32
5.4.2 Utilisation des pipes	32
5.4.3 Création d'un pipe personnalisé	33
5.5 Les directives	35
5.5.1 Présentation d'une directive	35
5.5.2 ngIf et ngFor.....	35
5.5.3 Création d'une directive d'attribut	36
5.5.4 Utilisation de notre directive	38
5.5.5 Les directives paramétrées	38
5.6 Les services	39
5.6.1 Présentation des services	39
5.6.2 Création d'un service	40
5.6.3 Utilisation d'un service	41
5.6.4 Niveau d'accès au service.....	41
5.7 Les modules	42
5.7.1 La modularité sur Angular	42
5.7.2 Contenu d'un module	43
5.7.3 Création d'un module	44

5.8 Les requêtes http	45
5.8.1 Présentation du HttpClientModule	45
5.8.2 Mise en place du « HttpClientModule »	45
5.8.3 Récupérer des données	46
5.8.4 Utiliser nos données asynchrones	48
5.9 Les formulaires	49
5.9.1 Les formulaires dans le développement web	49
5.9.2 NgForm	49
5.9.3 NgModel	50
5.9.4 Les règles de validation	51
5.10 Les tests sur Angular	53
5.10.1 Pourquoi tester son application ?	53
5.10.2 Les différents outils pour tester son application Angular	53
6. Le déploiement de notre solution Angular	55
6.1 Où déployer mon application Angular	55
6.2 Mise en place du déploiement	55
6.3 Exemple de déploiement sur Firebase Hosting	56
7. Conclusion	58
8. Bibliographie	59
Annexe 1 : Composant racine généré	63
Annexe 2 : fichier de configuration TypeScript	64
Annexe 3 : Pipe permettant d'afficher les équipes qualifiées à une compétition européenne	65
Annexe 4 : Service pour la récupération des matchs	66
Annexe 5 : Création d'une instance et retour de cette dernière	67

Liste des tableaux

Tableau 1 : Avantages et inconvénients de l'utilisation des SPA	5
---	---

Liste des figures

Figure 1 : Logo d'Angular	2
Figure 2 : Fonctionnement de TypeScript.....	4
Figure 3 : Logo de Node JS.....	6
Figure 4 : Différents moteurs pour l'exécution de JavaScript selon le navigateur	6
Figure 5 : Commande pour l'installation d'Angular CLI	8
Figure 6 : Affichage de la version de Angular CLI.....	9
Figure 7 : Commande permettant la génération d'un projet Angular	10
Figure 8 : Architecture de fichier d'un projet Angular	11
Figure 9 : Squelette du composant racine	14
Figure 10 : Module racine « app.module.ts ».....	16
Figure 11 : Démarrage de notre application Angular	17
Figure 12 : Exemple d'interaction avec le cycle de vie d'un composant	20
Figure 13 : Exemple de méthode.....	20
Figure 14 : Exemple d'objet métier avec constructeur représentant une équipe	21
Figure 15 : Exemple d'injections de dépendances.....	22
Figure 16 : Exemple d'utilisation d'une dépendance injectée	22
Figure 17 : Liaison d'un template avec un composant	23
Figure 18 : Exemple d'interpolation pour l'affichage du logo et du nom d'une ligue	23
Figure 19 : Exemple d'utilisation de l'événement KeyUp.....	25
Figure 20 : Génération d'un composant avec Angular CLI.....	26
Figure 21 : Exemple de « app-routing.module.ts »	27
Figure 22 : Exemple d'utilisation de <router-outlet> dans le template de notre composant racine	28
Figure 23 : Exemple de route paramétrée	29
Figure 24 : Utilisation d'un "router navigate" et d'une récupération de paramètre	31
Figure 25 : Exemple de génération d'un pipe	33
Figure 26 : Exemple de pipe.....	34
Figure 27 : Exemple de création d'une directive	36
Figure 28 : Directive permettant la surbrillance d'une carte d'un match lorsque l'utilisateur passe sa souris dessus.....	37
Figure 29 : Exemple d'utilisation d'une directive	38
Figure 30 : Exemple de directive paramétrée	39
Figure 31 : Exemple de création d'un service	40
Figure 32 : Exemple d'injection de service dans un composant.....	41
Figure 33 : Représentation de la modularité dans une application Angular	43
Figure 34 : Génération d'un module avec Angular CLI	44
Figure 35 : Exemple de module pour l'ensemble de fonctionnalités "fixtures"	45
Figure 36 : Exemple de méthode retournant un observable	47
Figure 37 : Exemple de consommation de données asynchrones.....	48
Figure 38 : Balise HTML « form » avec ngForm	49
Figure 39 : Bouton vérifiant si le formulaire est valide avant d'être cliquable	50
Figure 40 : Exemple d'utilisation de ngModel	51
Figure 41 : Exemple d'implémentation d'un message d'erreur en cas de champ non valide	52
Figure 42 : Résultat de tests avec Karma.....	54
Figure 43 : Fichier de configuration Firebase.....	57

Figure 44 :Exemple de réussite de déploiement.....	57
--	----

1. Introduction

De nos jours, la technologie est omniprésente, le développement d'application web est un domaine extrêmement important pour le bon fonctionnement de notre société.

Pour répondre aux demandes en hausse d'applications web robuste, performantes et évolutives, beaucoup d'outils de développement ont été créés. Angular, créé par Google, fait partie de ces outils.

Ce dernier permet la création d'applications web ou « clients riches » à la fois complexes et efficacement maintenable. Angular est plutôt populaire et s'est imposé comme un des premiers choix parmi énormément de Framework disponibles. Il se distingue des autres outils par ses fonctionnalités fortement intéressantes, comme la modularité, l'injection de dépendance ou le binding bidirectionnel. Étant open-source, Google permet à sa communauté de soutenir le Framework lui apportant ainsi une évolution rapide.

Comme absolument tous les outils de développement, Angular a des concurrents, les principaux sont « React.JS » et « Vue.js ». Ces derniers sont également des solutions fortement puissantes et intéressantes pour le développement de clients riches. Le point bénéfique de cette concurrence est que cela conduit à une obligation de faire rapidement évoluer les fonctionnalités et d'offrir toujours plus d'outils intéressants aux développeurs afin de répondre à leurs besoins et ainsi éviter de se faire devancer par les autres Framework. Cependant, il est important de noter que les trois n'ont pas forcément la même utilité. « React.JS », contrairement aux deux autres, est une bibliothèque JavaScript et non un Framework, elle permet la création d'interfaces utilisateur et est basée sur des composants web. Ensuite, nous avons « Vue.js » qui est un Framework dit progressif, ce qui veut dire que nous ne sommes pas obligés de partir de zéro avec ce Framework, mais que nous pouvons l'intégrer pour de petites parties de nos projets, ce qui est fortement utile quand nous avons un projet de base déjà créé et que nous ne pouvons pas nous permettre de repartir à zéro. Pour finir, Angular est un Framework dit « tout en un » et fournit tout ce dont nous avons besoin pour la création d'une application web.

En somme, la compréhension de ces outils et l'apprentissage pratique de ces derniers est devenue une compétence essentielle pour tout développeur web. Dans ce travail, nous allons explorer en détail le Framework Angular et ainsi parler de ses forces, ses faiblesses, ses différentes fonctionnalités, etc. Cela permettra de démontrer l'importance de l'apprentissage d'un outil comme celui-ci.

2. Angular

2.1 Présentation

Angular est un Framework JavaScript côté client open-source qui a été développé par Google et sa communauté. Ce Framework permet la création d'application web et mobile, il est une réécriture de son ancêtre AngularJS. La première version d'Angular est sortie en septembre 2016 et se nommait initialement « Angular 2 », il a fini par être renommé « Angular » tout court. Ce Framework est basé sur TypeScript. Étant open-source la communauté d'Angular participe activement à son développement.

Figure 1 : Logo d'Angular



(https://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/Angular_full_color_logo.svg/langfr-220px-Angular_full_color_logo.svg.png)

Cet outil permet la réalisation de plusieurs types d'applications. Nous avons la possibilité de créer des applications web aussi appelé client riche, ainsi que des applications mobile hybride.

Le Framework est aujourd'hui à la version 16.1.2 qui est sortie le 21 juin 2023.

2.2 Qu'est-ce qu'un Framework ?

2.2.1 Définition

Un Framework, traduit en français, est un cadre de travail, c'est un ensemble d'outils permettant au développeur d'avoir un ensemble de pratiques permettant de normaliser leurs codes.

Le but principal est de permettre au développeur web d'avoir des ressources préconçues et réutilisables afin de développer une application web plus efficacement.

2.2.2 L'avantage d'utiliser un Framework

Il y a beaucoup d'avantages à utiliser un Framework comme Angular pour la réalisation d'applications web :

- Premièrement, ce dernier permet un travail d'équipe de qualité étant donné qu'il est souvent pensé pour cela.
- Ensuite, il y a une organisation des fichiers qui est généralement bien pensée, cela permet un gain en productivité considérable, car le projet est organisé.
- Il y a aussi un énorme gain de temps, car les développeurs ont des outils préétablis leur permettant de se concentrer sur les aspects importants et plus spécifiques de leurs codes.
- Et pour finir, il y a généralement une communauté fortement présente qui apporte une aide précieuse et aide à l'évolution du Framework.

Ces avantages ne sont pas négligeables, l'utilisation et la maîtrise de l'un d'entre eux est devenu une compétence essentielle.

2.3 TypeScript

2.3.1 Présentation

Si on souhaite se lancer dans l'apprentissage d'Angular, il est obligatoire de maîtriser le TypeScript. C'est un langage de programmation open-source basé sur JavaScript, il a été créé par Microsoft et la première version est sortie en 2012. Aujourd'hui, le langage est à la version 5.1.6 qui est sortie le 05 juillet 2023.

Comme mentionné dans un article du « codeur blog ¹ » visant à comparer les deux langages, TypeScript résout un problème que beaucoup de développeurs soulignaient, le fait que JavaScript n'est pas excellent pour les gros projets. La mise en place d'un nouveau langage permettant d'améliorer cela était donc imminent.

Le but était alors de créer un langage qui va ressembler fortement voire être une copie conforme du JavaScript, mais qui permettra de faire de la programmation orientée objet (POO) et le tout sera fortement typé. Un point très important est que tout cela est optionnel, si l'on souhaite écrire du JavaScript classique dans un fichier avec l'extension « .ts² » c'est possible. On appelle ce type de langage un « langage multi paradigmes » ce qui signifie que l'on peut, effectuer de la programmation fonctionnelle ou de la programmation orientée objet avec ce dernier.

¹ [Lien de l'article du codeur blog](#)

² Extension de fichier pour les fichiers « TypeScript »

2.3.2 Comment fonctionne TypeScript ?

Un des points forts de ce langage, est qu'il ne dépayse pas les utilisateurs habitués à JavaScript, car la syntaxe est fortement similaire.

Comme dit précédemment, un code en TypeScript sera codé dans un fichier avec l'extension « .ts » et permettra d'effectuer du JavaScript en programmation orienté objet et avec un environnement typé.

La magie opère grâce à son compilateur nommé le TSC (TypeScript Compiler) qui va transformer le code TypeScript en JavaScript durant la compilation afin que le navigateur puisse correctement l'interpréter. Cela permet d'avoir un langage accepté par tous les navigateurs, mais qui offre beaucoup plus de possibilité au développeur pour effectuer leurs applications web.

Pour résumé, TypeScript offre une qualité de développement supérieure à JavaScript et permet tout de même d'être correctement interprété par les navigateurs grâce à son compilateur.

Figure 2 :
Fonctionnement de
TypeScript



(Reconstitution inspirée de : <https://www.jesuisundev.com/comprendre-typescript-en-5-minutes/>)

2.4 Single page applications (SPA)

2.4.1 Définition

Angular permet la création d'applications monopages il est donc important de comprendre le principe de ces dernières.

Une Single page application (SPA) ou en français application monopages est un type d'application web que l'on développe pour que tout le contenu soit accessible sur une seule page. Le but est d'éviter aux futurs utilisateurs de l'application d'avoir un chargement à chaque changement de page.

Au lieu de devoir charger une toute nouvelle page, les SPA mettent à jour une partie de celle-ci ce qui permet de rester sur la même page peu importe où on se trouve sur notre application en sachant que chaque page a ses fonctionnalités et son contenu. Cela permet d'optimiser le fonctionnement du site et de le rendre moins lourd.

2.4.2 Avantages et inconvénient

Il est important de voir que l'utilisation des SPA est réservée à une catégorie d'application assez précise sinon cela peut devenir contre-productif, car cela demande plus de développement. Elles sont généralement utilisées pour les applications web qui vont nécessiter beaucoup d'interaction avec leurs utilisateurs. Par exemple, pour des boutiques en ligne, l'utilisation des SPA est une bonne solution, car l'utilisateur aura beaucoup d'interaction avec le site et cela évitera un rechargement total des pages à chaque fois que l'utilisateur veut revenir en arrière ou changer de page.

Voici quelques avantages et inconvénients :

Tableau 1 : Avantages et inconvénients de l'utilisation des SPA

Avantages	Inconvénients
Temps de chargement réduit, car moins de choses à demander au serveur.	Moins bon référencement (SEO) car difficulté à lire le JavaScript pour les moteurs de recherche.
Nécessite moins de ressources.	Demande plus de développement.
Adaptation en applications mobiles facilitée.	
Possibilité d'utiliser le même backend dans l'application web et mobile	
Gains de performance	

(<https://www.ionos.fr/digitalguide/sites-internet/creation-de-sites-internet/single-page-application/>)

Les SPA sont aussi fortement performantes sur téléphone et peuvent très bien remplacer une application mobile. Cependant, il est important de noter qu'une application mobile reste la meilleure solution sur ce type d'appareil et sera forcément plus performante.

3. Configuration de l'environnement de développement

3.1 Node JS

3.1.1 Présentation

Pour pouvoir correctement utiliser Angular, il faut utiliser Node JS. Cet outil est ce qu'on appelle un « environnement d'exécution » et permet l'exécution de notre code JavaScript non pas sur le navigateur, mais côté serveur.

Figure 3 : Logo de Node JS

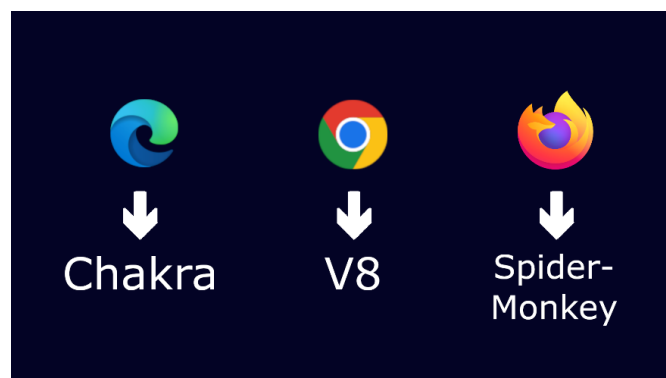


(https://upload.wikimedia.org/wikipedia/commons/thumb/d/d9/Node.js_logo.svg/1200px-Node.js_logo.svg.png)

Pour comprendre ce qu'est Node JS, il est important de voir premièrement ce qu'est un environnement. Un environnement en informatique représente le lieu où notre code va être exécuté, c'est donc là où il va être déchiffré afin de produire ce qu'on lui a demandé de faire, par exemple, les différentes opérations permettant d'afficher notre application web.

De base, si on n'utilise pas Node JS ou quelque chose de semblable, le navigateur sera considéré comme l'environnement d'exécution. De nos jours, ils possèdent tous, ce que l'on appelle un « moteur ». Il est essentiel de savoir que selon le navigateur, le moteur n'est pas le même :

Figure 4 : Différents moteurs pour l'exécution de JavaScript selon le navigateur



(Reconstitution inspirée de :

https://www.youtube.com/watch?v=Vz9RqPiyUo8&list=PLhVogk7htzNiO_cbnPqa7fkVzzRhUqQLj)

Étant donné que Node JS utilise JavaScript, il doit utiliser un moteur afin d'exécuter le code. Celui-ci est intégré directement dans Node JS et ce dernier n'est autre que le moteur V8.

Il faut savoir que ce moteur est utilisé par Node JS, car, à l'époque, c'était un des moteurs les plus puissants. De plus, il est open-source.

Pour résumer, un site web classique exécutera son code JavaScript côté navigateur grâce aux différents « moteurs » disponibles dans ces derniers tandis qu'une application web utilisant Node JS exécutera son code côté serveur avec le moteur V8 de Google Chrome.

3.1.2 NPM (Node Package Manager)

NPM est le gestionnaire de paquets de Node JS, sa fonction est de permettre aux développeurs de télécharger ce qu'on appelle des paquets externes pour leurs applications web.

Ces paquets servent à alléger notre travail en tant que développeur, car il va nous fournir des fonctionnalités qui ont déjà été faites par d'autres.

NPM fonctionne avec des fichiers nommé « package.json » et « package-lock.json », ces fichiers vont répertorier toutes les librairies et packages installés dans le projet concerné.

Tous les paquets que nous installons ainsi que leurs dépendances, sont stockées dans un dossier qui se nomme « node_modules ». Ce dossier contient littéralement le code des fonctionnalités souhaitées et nous permet d'utiliser ces dernières dans notre propre projet.

3.1.3 Utilisation

Dans ce travail, nous nous concentrons essentiellement sur Angular, c'est pour cela que la partie backend proposé par Node JS ne sera pas exploitée. Nous explorerons uniquement la partie environnement qui nous permettra d'exécuter notre code côté serveur ainsi que son gestionnaire de paquets « npm » qui nous permettra dans un premier temps de générer le squelette de notre projet ainsi que les dépendances de base dont le projet a besoin en installant « Angular CLI³ ».

³ Interface en ligne de commande

3.2 IDE utilisable

3.2.1 Présentation

Un IDE ou « environnement de développement » est un outil qui permet de faciliter le développement. L'IDE fournit en général certaines fonctionnalités très intéressantes qui permettront une augmentation de la productivité du développeur, par exemple grâce à l'auto-complétion ou à la génération automatique de squelettes de projet.

3.2.2 Exemples d'IDE

Pour effectuer une application avec Angular, il existe énormément d'environnement de développement nous permettant de le faire.

« Visual studio code » permet une bonne expérience de développement, car il possède un catalogue d'extension très bien fourni. De plus, il nous permettra d'installer une extension permettant de nous aider avec Angular nommé « Angular Language Service ».

« WebStorm » est excellent IDE lui aussi, il permet nativement de générer des projets Angular, React.JS et Vue.js et ne demande pas l'installation d'une quelconque extension afin de connaître le langage. Cependant, cet IDE est payant contrairement à « Visual studio code. »

3.3 Angular CLI

Afin de pouvoir gérer notre application Angular à partir de l'invite de commande, il va nous falloir un outil qui nous est directement fourni par Google nommé « Angular CLI ».

Angular CLI s'installe très simplement, exactement de la même façon qu'un package npm, il suffit d'utiliser la commande :

Figure 5 : Commande pour l'installation d'Angular CLI

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

(<https://angular.io/cli>)

Une fois cette commande entrée, Angular CLI est installé en global sur la machine, ce qui signifie qu'il sera utilisable sur tous les futurs projets effectués sur le poste de travail concerné. Afin de vérifier que l'installation s'est bien déroulée, il faut utiliser la commande « ng version », voici ce que doit retourner cette dernière :

Figure 6 : Affichage de la version de Angular CLI

```
Angular CLI
Angular CLI: 15.2.3
Node: 18.14.2
Package Manager: npm 9.5.0
OS: win32 x64

Angular:
...

Package                                Version
-----
@angular-devkit/architect              0.1502.3 (cli-only)
@angular-devkit/core                   15.2.3 (cli-only)
@angular-devkit/schematics             15.2.3 (cli-only)
@schematics/angular                   15.2.3 (cli-only)
```

(Screen personnel)

Une fois Angular CLI installé, le poste de travail est prêt à générer des projets Angular ainsi que d'autres types de choses tel que des composants, modules, etc. Il possède également toutes les commandes qui permettront par la suite de gérer le projet.

4. Configuration du projet Angular

4.1 Génération du projet

La première étape de toute création d'un projet Angular, est sa génération ou sa création.

Pour cela, il faut utiliser un terminal placé au niveau du dossier qui va accueillir ce dernier, ensuite, il faut entrer la commande suivante :

Figure 7 : Commande permettant la génération d'un projet Angular

```
ng new app-name --minimal --style=css
```

(Screen personnel)

Cette commande, lorsqu'elle est exécutée, va générer l'entièreté des fichiers et dossiers nécessaires pour une application Angular.

Le « ng new » permet de transmettre à Angular CLI que l'on souhaite générer un nouveau projet, ensuite, il faut fournir le nom du projet.

Pour finir, on peut, si on le souhaite, ajouter des options à la génération de notre projet. Par exemple, « --minimal » permet de générer une version plus légère qui va fournir au projet moins de fichiers de configuration et moins de librairies natives, puis « --style=CSS » désigne le CSS⁴ comme langage utilisé pour le style de notre application.

Une fois cette dernière exécutée, notre invite de commande nous demande si nous souhaitons ajouter du routage à notre application, le routage nous permettra de naviguer entre les futurs pages, il est donc préférable de répondre oui afin de ne pas avoir à le faire nous-même.

Le projet est maintenant généré et prêt pour le développement.

⁴ Cascading Style Sheets, langage permettant de gérer le style d'un site web

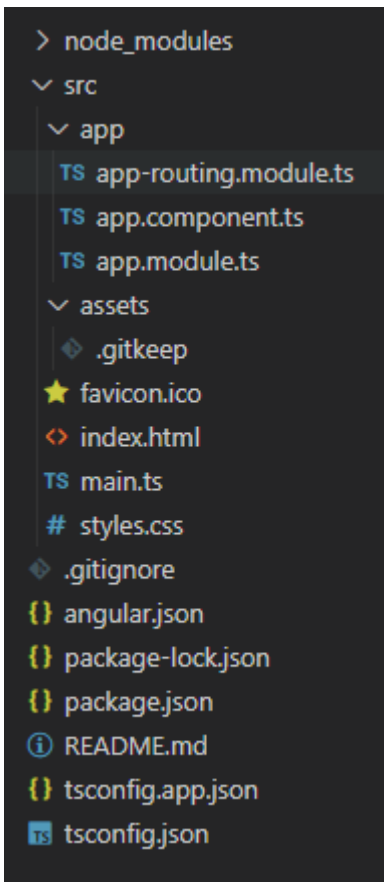
4.1.1 Architecture de dossier du projet

Il est important de comprendre à quoi servent les fichiers présents dans le dossier afin de les modifier si besoin et d'utiliser le plein potentiel d'Angular.

Premièrement, nous retrouvons le dossier « `node_modules` » qui va stocker toutes les dépendances dont notre projet a besoin pour fonctionner correctement. Par exemple, si nous installons une nouvelle librairie comme Angular Materialize CSS⁵, tout le code nécessaire au bon fonctionnement de cette librairie sera stocké dans le dossier « `node_modules` ».

Ensuite, nous avons le dossier dans lequel le développeur passe la majorité de son temps durant la conception de l'application, il s'agit du dossier « `src` » qui contiendra, comme son nom l'indique, les sources de notre projet. Dans ce dossier « `src` », il y a le sous-dossier « `app` » qui va contenir le code source de notre projet. Lorsque l'on génère un projet, certains fichiers se trouvent déjà à l'intérieur :

Figure 8 : Architecture de fichier d'un projet Angular



(Screen personnel)

- « **app-routing-module.ts** » ce fichier contiendra les routes de notre projet, il permettra notamment d'attribuer un URL à une page afin que nous gardions un historique malgré le fait qu'Angular fasse des SPA.
- « **app-components.ts** » est le composant racine permettant l'affichage de l'application.
- « **app-module.ts** » est le module racine.

Dans ce même dossier « `src` » nous avons le sous-dossier « `assets` » qui permet de stocker des éléments que nous utiliserons dans le projet tel que des images par exemple.

Si nous regardons directement dans le dossier « `src` » nous avons plusieurs fichiers disponibles, « `main.ts` » est le fichier permettant d'initialiser l'application et « `index.html` » est le point d'entrée de notre application, dans ce fichier nous appelons « `<app-root>` » qui est une balise permettant d'appeler notre composant racine.

⁵ Libraries permettant de fournir des classes facilitant l'implémentation du CSS dans notre code. Créé par Google.

Ensuite, à la racine du projet, nous avons tous les fichiers de configuration de notre application. Premièrement, nous avons « `angular.json` » qui contient la configuration de Angular CLI. Par exemple, lors de la génération du projet, si nous avons mis l'option « `--minimal` » dans notre commande, c'est ce fichier qui va préciser ce que nous souhaitons générer pour notre projet.

Ensuite, « `package.json` » contient toutes les descriptions des dépendances de notre projet ainsi que leurs versions. Par exemple, dans celui-ci, nous avons toutes les dépendances permettant la création de notre projet Angular.

Nous avons aussi le fichier « `package-lock.json` », qui lui, permet de figer la description de nos dépendances et d'avoir des informations beaucoup plus précise que dans le « `package.json` ».

Pour finir, nous avons les fichiers de configuration de TypeScript :

- « **`tsconfig.app.json`** » contient la configuration du « TypeScript compiler ». La configuration du compilateur dans une application Angular est faite en deux fichiers, c'est pour cela que dans notre « `tsconfig.app.json` » nous avons la ligne « `« extends » : « ./tsconfig.json »` » qui permet de faire comprendre que la suite de la configuration se trouve dans le deuxième fichier.
- « **`tsconfig.json`** » contient la suite de la configuration du compilateur TypeScript, il y a un peu plus d'éléments que dans l'autre fichier de configuration.

4.1.2 Autres fichiers

Il y a certains fichiers qui n'apparaissent pas forcément dès la génération du projet, mais qui peuvent s'avérer utiles pour les projets Angular :

- « **`.browserslistrc`** » est un fichier permettant de configurer notre application en paramétrant certains outils pour que notre application fonctionne sur tous les navigateurs mentionnés dans ce fichier. Si nous n'avons pas ce fichier, Angular possède des configurations de base.
- Dans le dossier « `src` » nous avons la possibilité de créer des fichiers permettant d'y stocker nos variables d'environnement. Ces derniers se nomment « `environment.prod.ts` » (contient les variables globales de production) et « `environment.ts` » (contient les variables globales pour le développement). Ces fichiers se trouvent généralement dans un dossier nommé « `environments` ».

Ces fichiers étaient générés automatiquement, mais depuis Angular 15.1, ils sont à configurer manuellement.

4.2 Premier démarrage

4.2.1 Composant racine

Lors de la génération, on peut voir qu'Angular fournit un premier « composant » qui se nomme « app.components.ts ». Il a été généré avec un contenu de base permettant d'avoir une page non-vide lors du premier démarrage.

Ce fichier est ce que l'on appelle un composant racine. C'est celui qui sera appelé en premier lors de l'exécution du projet et il contiendra le « template » principal du projet.

4.2.1.1 Description du contenu d'un composant racine

La plupart du code présent sur ce fichier est effaçable, car nous n'en avons pas besoin étant donné qu'il y a juste une présentation du Framework avec quelques liens redirigeant vers de la documentation. La première version du document est disponible en [annexe](#).

Il est important de connaître chaque point présent dans ce fichier, premièrement, nous avons, tout en haut de ce dernier, les « imports » qui représentent tous les éléments dont nous avons besoin pour le bon fonctionnement de la page dans laquelle nous nous trouvons. Dans notre composant racine, nous importons l'élément « Components » issu de la librairie nommée « @angular/core » qui représente le « cœur » du Framework Angular.

Ensuite, il y a un bloc commençant par « @Component » utilisable grâce à l'importation faite précédemment. Ce qui commence par un « @ » est ce qu'on appelle un décorateur et cela permet grâce à un « @ » d'effectuer une modification sur le comportement d'une classe ou d'une fonction. Dans notre cas, on lui fait comprendre que nous souhaitons faire de ce fichier un composant Angular.

L'annotation « @Component » a besoin de deux options pour fonctionner correctement, la première se nomme « selector » et va permettre d'attribuer un nom à notre « composant » offrant ensuite la possibilité d'y accéder étant donné qu'il sera identifié par ce dernier. Par exemple, notre composant racine se nomme « app-root », il représente la racine de notre projet et on pourra donc l'appeler en l'important et en utilisant la balise « <app-root> ». Nous utilisons ce que l'on appelle des composants web et cela permet de créer nos « balises HTML personnalisées ».

Puis, nous avons l'option « template » qui elle contient notre HTML. Pour pouvoir écrire notre code, il faut le mettre entre deux « accents graves » ou en anglais « backtick », cela vient du JavaScript et permet de fournir des chaînes de caractères sur plusieurs lignes sans avoir à concaténer, car dans ce cas le code HTML que l'on fournit est

considéré comme une chaîne de caractères multilignes. Dans « template », nous avons aussi la possibilité d'appeler des propriétés que nous mettons entre deux accolades « {{maPropriété}} ». Cela nous permet d'afficher du contenu de façon dynamique. Par exemple, lorsque nous générons notre projet, il y a une variable « title » qui est initialisée avec le nom de l'application comme valeur.

Pour finir, nous retrouvons une classe qui se nomme « AppComponent » et qui stocke les différentes propriétés de la page. Cette classe est utilisée comme composant étant donné qu'elle est accompagnée du décorateur « @Component » juste au-dessus. Finalement, le mot-clé « export » permet au composant, d'être appelé hors de son fichier.

Figure 9 : Squelette du composant racine

```
1  import { Component } from '@angular/core';
2
3  You, maintenant | 1 author (You)
4  @Component({
5    selector: 'app-root',
6    template: `
7      <!--The content below is only a placeholder and can be replaced.-->
8      <h1>
9        Welcome to {{title}}!
10     </h1>
11     <router-outlet></router-outlet>
12   `
13 })
14 export class AppComponent {
15   title: String = 'app-training';
16 }
```

(Screen personnel récupéré du code généré par Angular CLI)

4.2.2 Module racine

Le module racine dans Angular permet de regrouper tous les modules, composants, services et autres utilisés dans l'application. Il permet aussi de les initialiser lors de l'exécution de cette dernière. Ce module racine est créé automatiquement lors de la génération des fichiers de l'application et se nomme « app.module.ts », il se trouve dans le dossier « src/app ».

4.2.2.1 Description du contenu d'un module racine

Lors de la génération du module racine, il y a du code fourni qui cette fois-ci ne doit pas être supprimé, car il contient toutes les informations de bases pour le bon fonctionnement de l'application.

Au fil du temps, nous allons logiquement ajouter du contenu au module racine, car nous allons créer des composants, services, pipes ou autres modules plus spécifiques qui regrouperont des fonctionnalités précises. Par exemple, nous avons la possibilité de créer un module spécifique qui regroupera tous les composants, services, directives et autres nécessaires à l'authentification. Ce module devra ensuite être appelé dans le module racine, cela permet de regrouper plus facilement des fichiers qui représente la même fonctionnalité et d'organiser de façon optimale son application.

Premièrement, dans un module, et plus particulièrement dans notre module racine, nous avons les imports, il y a plusieurs modules déjà importés qui ont chacun leurs fonctions dans ce fichier :

- « **NgModule** » : Tout comme notre élément « Component », « NgModule » est lui aussi issu de la librairie « @angular/core » qui représente le cœur d'Angular. Le décorateur « @NgModule » définit ce fichier comme un module.
- « **BrowserModule** » : Il fournit des éléments très importants pour le bon fonctionnement de notre application.
- « **AppRoutingModule** » : Le fichier contenant les routes du projet.
- « **AppComponent** » : Le composant racine.

Ensuite, nous avons notre décorateur « @NgModule », comme pour le composant racine, c'est dans ce décorateur que nous aurons la possibilité de définir un certain nombre d'éléments :

- « **declarations** » : C'est dans cet élément que l'on déclare la liste de nos composants, de nos pipes, etc.
- « **imports** » : Dans cet élément, nous pouvons lui fournir tous les autres modules dont nous avons besoin. Ce dernier n'accepte que des modules, nous ne pouvons pas lui fournir de composant ici. Pour résumer, il faut mettre tout ce qui n'est pas un module dans l'onglet « declarations » et tous les modules dans « imports ».
- « **providers** » : Permet d'utiliser le « système d'injection de dépendances ⁶ » d'Angular.
- « **bootstrap** » : Cet élément est uniquement présent dans le module racine, il ne sera donc pas présent dans tous les autres modules que nous créerons dans un projet Angular. Il permet de définir auprès d'Angular quel est notre module racine. L'application saura donc quel composant afficher quand nous la lançons.

⁶ Expliqué au point 5.2.5

Pour finir, nous avons notre classe « AppModule » qui est également un module, car elle est accompagnée du décorateur « NgModule » et permettra de stocker les différentes propriétés dont nous aurons besoin.

Figure 10 : Module racine « app.module.ts »

```
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6
7  @NgModule({
8    declarations: [
9      AppComponent
10   ],
11   imports: [
12     BrowserModule,
13     AppRoutingModule
14   ],
15   providers: [],
16   bootstrap: [AppComponent]
17 })
18 export class AppModule { }
```

(Screen personnel du code généré par Angular CLI)

4.2.3 Configuration de TypeScript

Avant de démarrer son projet, il est essentiel de savoir configurer correctement le compilateur TypeScript, car cela évitera beaucoup de mauvaises surprises.

Par défaut, TypeScript est configuré de façon plutôt rigoureuse. Par exemple, si nous souhaitons typer une variable avec le type « number » sans lui attribuer de valeur par défaut, nous serons obligés de préciser que le type de notre variable est soit « number » soit « undefined », car TypeScript a besoin de savoir quel type lui donner en cas de problème et le type par défaut de JavaScript est « undefined ».

Il est possible de changer cela si nous nous rendons dans la configuration de TypeScript, le fichier se nomme « tsconfig.json » et possède plusieurs éléments déjà en place avec une valeur attribué par défaut. Dans notre cas, celle qui nous intéresse est la propriété « strictPropertyInitialization » qui n'est pas présente dans notre fichier de base. Il faut donc l'ajouter nous-même juste en dessous de la propriété « strict » et la mettre à « false ». Le fichier de configuration est disponible en [annexe](#).

Il y a bien évidemment énormément de possibilité de configuration, cet exemple était simplement essentiel pour améliorer le confort d'utilisation de TypeScript.

4.2.4 Commande pour le démarrage de l'application

Une fois l'application correctement configurée, nous pouvons compiler ce code à l'aide d'une commande fournie par Angular CLI. Lorsque qu'elle est exécutée, nous avons accès au rendu de notre application. Nous avons aussi la possibilité de développer et de voir le résultat de notre développement en direct.

La commande pour lancer notre application est « ng serve », pour utiliser cette commande, il faut se rendre dans un terminal et se placer dans le répertoire du projet. Si l'IDE possède un terminal, cette action est facilitée, il suffira d'utiliser le terminal de ce dernier qui normalement se trouve directement dans le bon répertoire.

Figure 11 : Démarrage de notre application Angular

```
PS C:\Users\bruno\Desktop\HEG\Travail-de-bachelor\trainingProject> ng serve
✓ Browser application bundle generation complete.

Initial Chunk Files | Names          | Raw Size
vendor.js           | vendor         | 2.04 MB
polyfills.js        | polyfills      | 314.28 kB
styles.css, styles.js | styles         | 209.41 kB
runtime.js          | runtime        | 6.52 kB
main.js             | main           | 6.27 kB
                    | Initial Total  | 2.56 MB

Build at: 2023-04-03T15:45:56.199Z - Hash: 55ae4a7742b9a634 - Time: 2286ms

** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

✓ Compiled successfully.
```

(Screen personnel)

Nous avons la possibilité d'ajouter l'option « --open » à notre commande afin d'ouvrir automatiquement un navigateur lors de l'exécution de cette dernière.

Lorsque l'application est exécutée, elle est disponible au port « 4200 » de notre « localhost ».

5. Conception d'une application Angular

5.1 Présentation de l'application

Avant d'exposer tous les principaux concepts d'Angular, il est important de présenter l'application qui sera démontrée dans les prochaines figures en guise d'exemples.

« LiveBall » est une application permettant la consultation de matchs de football en direct. Dans cette application, nous avons l'aperçu des matchs en cours avec le score, le statut, les événements passés, les effectifs et les statistiques. Puis, il y a aussi la possibilité de consulter certaines informations pour chaque ligue comme le classement et les prochains matchs. L'application utilise l'API « api-football ⁷ ».

Le but de cette application, est de mettre en place tous les principaux concepts d'Angular afin d'avoir des exemples d'utilisation concrets.

Dans les prochains points, il y aura beaucoup de références aux « modules » qui eux seront présentés uniquement au point 5.7.

5.2 Les composants web

5.2.1 Présentation

Nous avons pu voir ce qu'était un « composant racine » dans Angular, cependant un « composant racine » est avant tout un « composant », il est donc essentiel de bien comprendre ce que c'est, étant donné qu'Angular est un Framework orienté « composant ».

Un composant web, est un système totalement indépendant du reste de l'application, ce qui veut dire qu'il a sa propre logique, son propre style, etc. Un composant nous permet de gérer une partie plus ou moins grande de la page que l'on nomme « une vue » et ne servira qu'à gérer cette dernière. Par exemple, si l'on souhaite créer le header⁸ de l'application, on va devoir créer un composant qui contiendra tout ce que ce dernier nécessite pour son bon fonctionnement. Il lui faudra sa logique qui sera présente dans la « class » du composant, sa vue qui est la partie « template » de ce dernier, ainsi que tous les imports nécessaires.

⁷ Api-football : <https://www.api-football.com/>

⁸ Bar de navigation présente en haut d'un site web

5.2.2 Cycle de vie d'un composant

Un composant a ce qu'on appelle un cycle de vie, ce dernier est géré par Angular qui va décider à quel moment le composant est initialisé afin de l'afficher, à quel moment ce dernier est détruit, car l'utilisateur n'en a plus besoin, etc.

Angular nous donne la possibilité de gérer les cycles de vie du composant, grâce à ce qu'on appelle des « interfaces ». Ces interfaces auront chacune leurs fonctions et fourniront des méthodes permettant d'effectuer, par exemple, des actions lors de la destruction d'un composant si nous en avons besoin.

Voici tout ce qui est fourni afin de gérer le cycle de vie :

- **ngOnChanges** : cette méthode est la toute première appelée lors de la création d'un composant, elle permet de détecter le moindre changement de valeur dans le composant et prend en paramètre l'ancienne valeur et la nouvelle.
- **ngOnInit** : Cette méthode permet d'effectuer des actions au moment de l'initialisation du composant. Cela peut paraître spécial, mais elle est bien appelée après ngOnChanges.
- **ngDoCheck** : C'est un excellent complément de ngOnChanges, il va permettre d'élargir le comportement de ce dernier afin de détecter des variations qu'Angular n'aurait pas forcément vu, car il n'est pas apte à le faire par défaut.
- **ngAfterViewInit** : méthode appelée juste après l'affichage de la vue du composant.
- **ngOnDestroy** : Est appelée en dernier juste avant la suppression du composant qui sera totalement retiré du DOM.

Les méthodes qui seront le plus communément utilisées seront « ngOnInit » qui nous permettra d'effectuer des actions à l'initialisation de notre composant et « ngOnDestroy » qui nous permettra de nettoyer la mémoire du navigateur pour plus de performances.

5.2.2.1 Exemple d'interaction avec le cycle de vie

Premièrement, si l'on souhaite utiliser une des méthodes permettant de gérer le cycle de vie d'un composant, il faut l'importer. Toutes les méthodes permettant cela se trouvent dans le cœur d'Angular, exactement comme l'élément « component ». Il faut donc importer « OnInit » qui se trouve dans « @angular/core ».

Ensuite, étant donné que « OnInit » est une interface, il faut l'implémenter au niveau de la classe de notre composant.

Pour finir, il faut appeler la méthode fournie par cette interface qui se nomme « ngOnInit ». Une fois ceci fait, tout ce qui sera codé à l'intérieur de cette méthode s'exécutera à l'initialisation du composant.

Figure 12 : Exemple d'interaction avec le cycle de vie d'un composant

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

...

@Component({
  selector: 'app-root',
  template: `
    <h1>
      Welcome to {{title}}!
    </h1>
    <router-outlet></router-outlet>
  `
})
export class AppComponent implements OnInit {
  ngOnInit() {
    console.log('Bienvenue sur l\'application');
  }

  title: String = 'app-training';
}
```

(Screen personnel)

5.2.3 Les méthodes

Un des points essentiels à traiter est la création de méthode permettant, par exemple, des interactions de l'utilisateur avec l'application.

5.2.3.1 Syntaxe

Une méthode sur Angular doit être créée dans une classe, le premier élément à fournir est le nom de la méthode accompagné de parenthèses qui permettront d'accueillir les paramètres que nous souhaitons donner à notre méthode. Il y a la possibilité de fournir un type de retour, il suffit de mettre l'opérateur « : » puis le type à la suite des parenthèses de la méthode, cela démontre bien la capacité de TypeScript à typer tout en utilisant du JavaScript.

Figure 13 : Exemple de méthode

```
returnString() : string {
  return 'Hello World';
}
```

(Screen personnel)

5.2.4 Les objets métiers

TypeScript permet d'effectuer de la programmation orientée objet (POO), nous avons donc la possibilité de créer ce qu'on appelle des objets métiers. Si nous prenons comme exemple une application de football qui recense des matchs en direct, nous devons à un moment créer un objet métier « Match » qui prendra toutes les caractéristiques d'un match, par exemple le nom des équipes, le score, le statut du match, etc.

La création d'un objet métier permettra ensuite d'attribuer cet objet en tant que type dans un composant ou autre pour pouvoir recevoir uniquement des matchs en retour d'une méthode par exemple. Pour commencer la création d'un objet, il faut créer un fichier en « .ts » qui ne contiendra qu'une classe qui représentera notre objet. Ensuite, il faut lui passer les attributs que nous souhaitons avec leurs types. Cela permettra d'avoir un objet nécessitant tous ces attributs obligatoirement afin d'être créé.

Si l'application nécessite la création de nouvelles instances de l'objet, il y a la possibilité de mettre en place un constructeur qui permettra d'utiliser la fonction « new » et de directement attribuer les valeurs des attributs à la nouvelle instance.

Figure 14 : Exemple d'objet métier avec constructeur représentant une équipe

```
export class Team {  
  id: number;  
  name: string;  
  code: string;  
  country: string;  
  founded: number;  
  national: boolean;  
  logo: string;  
  
  constructor(  
    id: number,  
    name: string,  
    code: string,  
    country: string,  
    founded: number,  
    national: boolean,  
    logo: string  
  ) {  
    this.id = id;  
    this.name = name;  
    this.code = code;  
    this.country = country;  
    this.founded = founded;  
    this.national = national;  
    this.logo = logo;  
  }  
}
```

(Screen personnel)

5.2.5 Les injections de dépendances

Dans notre composant, nous avons la possibilité d'effectuer des « injections de dépendances ». Cela signifie, que nous allons fournir dans les parenthèses du constructeur de notre composant, des dépendances qu'il pourra utiliser dans toute sa classe. C'est souvent le cas pour les services, si par exemple nous en créons un permettant d'interagir avec l'API, nous devons injecter ce service dans les parenthèses de notre constructeur et nous pourrions ensuite utiliser toutes les méthodes permettant les interactions avec l'API dont on a besoin.

Figure 15 : Exemple d'injections de dépendances

```
constructor(private fds: FootballDataService, private router: Router) {}
```

(Code tiré de mon application LiveBall)

Dans cette figure, nous pouvons voir que j'injecte comme dépendances un service nommé « FootballDataService » et un service « Router » qui me permet d'utiliser des fonctions qui interagissent avec mes routers.

Figure 16 : Exemple d'utilisation d'une dépendance injectée

```
this.fds.getFixturesByDate(this.formatedDate).subscribe((res: any[]) => {  
  this.fixtures = (res as any).response;  
  this.OnListFixtures();  
});
```

(Code tiré de mon application LiveBall)

Puis dans celle-ci, nous voyons un exemple d'utilisation de mes dépendances injectées, dans ce cas, j'utilise mon « service » grâce au mot-clé « this » et je peux maintenant utiliser la fonction « getFixturesByDate » présente dans le service.

5.2.6 Les templates

Un template sur Angular représente la vue d'un composant, donc ce qu'il va afficher dans le navigateur. Dans la figure 12, nous avons pu voir un tout petit exemple directement écrit dans la partie « template » entre « backtick ». Cependant, il y a beaucoup de soucis qui peuvent se présenter lorsqu'on écrit directement notre HTML dans notre composant. Par exemple, dans des applications plus grosses, le HTML peut être amené à contenir beaucoup de lignes ce qui rendra notre composant très grand et très difficile à maintenir dû au nombre important d'information.

La solution à cela, est de séparer la partie vue dans un fichier en « .component.html » et de garder uniquement la partie logique dans notre composant. Cela apportera beaucoup plus de clarté et de maintenabilités.

Pour lier ce template qui est maintenant séparé de notre composant, il va falloir remplacer le mot-clé « template » dans notre composant par « templateUrl » et lui fournir le nom de notre fichier contenant le « html ».

Figure 17 : Liaison d'un template avec un composant

```
@Component({
  selector: "app-list-matches",
  templateUrl: "../list-matches.component.html",
  styles: [],
})
export class ListMatchesComponent implements OnInit {
```

(Code tiré de mon application LiveBall)

5.2.6.1 L'interpolation

Ensuite, pour utiliser la logique de notre composant dans notre template, rien ne change, il suffit d'utiliser le concept d'interpolation.

Le concept d'interpolation est le fait d'utiliser deux accolades pour appeler une propriété, fonction ou autre présente dans la logique de notre composant.

Figure 18 : Exemple d'interpolation pour l'affichage du logo et du nom d'une ligue

```
<div class="card-content white-text">
  <div class="col s12">
    
    &nbsp;&nbsp;&nbsp;{{ league.name }}
  </div>
  <br />
</div>
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, nous pouvons y voir un exemple d'interpolation. J'utilise une propriété qui est un objet nommé « league » contenant des attributs comme le « logo »

et le « name ». De cette façon, je peux afficher toutes les propriétés présentes dans mon composant. De plus, l'interpolation permet aussi d'exécuter des expressions TypeScript, par exemple nous avons la possibilité d'y écrire directement des additions, multiplications, etc.

Cependant, si une propriété est déclarée comme « private » dans notre composant, elle ne sera pas accessible depuis le template. Pour contourner cela, nous pouvons soit la déclarer comme « public », soit fournir une méthode « getter » pour accéder à cette propriété.

L'interpolation est aussi utilisable dans les propriétés de notre balise par exemple si nous voulons utiliser un URL pour notre image présent dans une propriété d'un de nos objets dans la logique du composant. Nous pouvons utiliser l'interpolation dans notre « src » de cette façon « ».

5.2.6.2 Les événements

Dans notre template, nous avons la possibilité d'intercepter plusieurs types d'événements qui permettront ensuite d'effectuer des actions.

Voici quelques exemples des différents types d'événements disponibles avec Angular :

- **Click** : L'événement se déclenche dès que l'utilisateur clique sur l'élément concerné.
- **DbClick** : L'événement se déclenche dès que l'utilisateur double-clique sur l'élément concerné.
- **MouseEnter** : L'événement se déclenche dès que l'utilisateur met sa souris sur l'élément concerné.
- **MouseLeave** : L'événement se déclenche dès que l'utilisateur enlève sa souris de l'élément concerné.
- **KeyUp** : L'événement se déclenche dès que l'utilisateur relâche une touche de clavier.
- **KeyDown** : L'événement se déclenche dès que l'utilisateur appuie sur une touche de clavier.
- **Submit** : L'événement se déclenche dès que l'utilisateur soumet un formulaire.
- **Change** : L'événement se déclenche dès qu'un changement est détecté dans l'élément concerné.

Bien évidemment, ce ne sont que des exemples très couramment utilisés et il existe bien plus d'événements que ceux cités ci-dessus.

Figure 19 : Exemple d'utilisation de l'événement KeyUp

```
<div class="input-field">
  <input
    id="search"
    type="search"
    placeholder="Rechercher"
    #input
    (keyup)="searchLeagueByName(input.value)"
  />
  <label class="label-icon" for="search"></label>
</div>
```

(Code tiré de mon application LiveBall)

Dans cette figure, j'ai utilisé l'événement « KeyUp » qui va détecter dès que l'utilisateur va relâcher sa touche de clavier et effectuer une recherche dans une liste afin de n'afficher que les ligues qui correspondent à ce que l'utilisateur est en train d'écrire.

5.2.6.3 Les variables référencées dans le template

Angular nous donne la possibilité d'ajouter directement des variables dans notre template qui permettront par exemple, de fournir dynamiquement la valeur de notre « textbox » à une méthode de notre composant.

Dans la figure ci-dessus, nous pouvons voir qu'il y a un « #input », ce mot-clé va tout simplement récupérer la valeur de notre « textbox » et la fournir directement à la méthode « searchLeagueByName » grâce à sa propriété « value ».

Dans ce cas, la variable s'appelle « input » mais nous avons la possibilité de la nommée comme on le souhaite. En revanche le caractère « # » est obligatoire.

5.2.6.4 Générer un composant

Angular CLI nous offre la possibilité de générer directement nos composants et tous nos autres types de fichiers, fournissant une multitude d'options pour configurer cela. Par exemple, nous pouvons choisir de générer un composant avec son template directement séparé, ce qui permet d'obtenir deux fichiers distincts, mais directement lié avec templateURL.

L'avantage majeur de l'utilisation de cette méthode de génération est sa capacité à produire tous les fichiers nécessaires et à mettre à jour simultanément le module correspondant qui accueillera le composant.

Figure 20 : Génération d'un composant avec Angular CLI

```
PS C:\Users\bruno\Desktop\HEG\99-99_Travail-de-bachelor\TB-ON-AIR-FOOTBALL> ng generate component /login/auth --inline-template=false
CREATE src/app/login/auth/auth.component.html (19 bytes)
CREATE src/app/login/auth/auth.component.ts (172 bytes)
UPDATE src/app/login/login.module.ts (487 bytes)
```

(Screen personnel)

Dans la figure ci-dessus, nous voyons un exemple de génération d'un composant. Premièrement, dès que l'on souhaite générer quoi que ce soit (module, composant, service) nous devons utiliser la commande « ng generate » suivi de ce que l'on souhaite générer (dans notre exemple « component »).

Nous devons ensuite renseigner le nom du composant (auth) précédé du répertoire où l'on souhaite le créer si besoin (/login/). Si aucun répertoire spécifique n'est indiqué, Angular CLI en créera un automatiquement portant le nom du composant créé. Nous avons également la possibilité de spécifier des options. Dans ce cas, l'utilisation de l'option « --inline-template=false » permet de générer un composant avec son template séparé.

Pour finir, on peut constater qu'Angular CLI a créé les deux fichiers (le composant et le template) et qu'il a mis à jour le module qui va accueillir ce dernier.

5.3 Le routing

5.3.1 Qu'est-ce que le routing Angular

Comme nous l'avons vu précédemment, Angular permet la création de Single Page Applications (SPA) qui rendent possible la réalisation d'une solution finale performante. En effet, ce type d'application n'a plus besoin de recharger la totalité d'une page, mais se contente simplement de mettre à jour une partie de la page en appelant un composant spécifique.

Afin de pouvoir créer une SPA correctement, il faut implémenter du routing. Cette fonctionnalité permet de simuler la navigation classique du navigateur en conservant un historique des pages visitées en leur attribuant un URL spécifique. Ainsi, le navigateur sera capable d'afficher la bonne combinaison de composant en fonction de l'URL qui lui a été fourni.

Le routing permet aussi l'accès à une page spécifique en fournissant directement l'URL souhaité au navigateur. En conclusion, Angular nous donne l'impression d'utiliser un système de navigation classique alors qu'en réalité, il cache et affiche des composants lorsque nous changeons de page.

5.3.2 L'implémentation du routing

Lorsque nous générons notre application, Angular CLI nous donne la possibilité de générer automatiquement les fichiers de routing. Généralement, il est conseillé d'accepter cette proposition afin de faciliter la mise en place de cette fonctionnalité.

Une fois l'option acceptée, un fichier nommé « app-routing.module.ts » va être créé. Ce module permettra d'attribuer à un composant son URL permettant d'y accéder durant notre navigation. Quand nous changerons de page, Angular viendra simplement regarder à l'intérieur de ce fichier afin de savoir quel composant afficher.

Ce fichier est composé de plusieurs parties, nous avons premièrement les importations qui nous permettent d'importer :

- Les composants auxquels nous souhaitons attribuer nos routes
- Le « NgModule » étant donné que le fichier en question est un module.
- Les éléments « Routes » et « RoutesModule » qui permettent la gestion de nos routes, ces deux éléments sont importés du paquet « @angular/router ».

Ensuite, nous avons la liste de nos routes. Cette dernière demande une syntaxe spécifique, premièrement nous lui donnons notre « path » qui représente le chemin qui permettra d'accéder au composant. Puis, nous lui donnons le composant qu'il doit afficher une fois sur cet URL. Il faut typer cette liste avec le type « Routes ».

Pour finir, dans le décorateur « NgModule », nous utilisons l'élément « RouterModule.forRoot(nomListe) » auquel on va fournir la liste de nos routes.

Figure 21 : Exemple de « app-routing.module.ts »

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/router";
import { LoginComponent } from "../login/login.component";

const routes: Routes = [
  { path: "auth/", component: LoginComponent },
  { path: "", redirectTo: "matches/", pathMatch: "full" },
  { path: "**", redirectTo: "matches/", pathMatch: "full" },
];

You, 2 months ago | 1 author (You)
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

(Code tiré de mon application LiveBall)

5.3.3 La balise <router-outlet>

Une fois que nous avons correctement configuré le routing dans notre application, il va falloir ajouter quelque chose à notre « template » du composant racine afin que le routing soit disponible côté utilisateur.

Pour cela, nous avons besoin de la balise « <router-outlet > ». Cette balise permettra de relier les routes que nous avons définies, avec notre template. Pour le moment, ce dernier est figé et seul le contenu directement codé à l'intérieur s'affiche lorsque nous nous rendons sur l'application. Notre but est que la partie principale (le contenu) de notre page soit mise à jour en affichant le composant que nous souhaitons quand l'URL change. Pour cela, nous allons ajouter la balise dans le template de notre composant racine. Cela permettra à Angular de savoir quelle partie de notre application sera dynamique et changera selon l'URL entré par l'utilisateur. Il s'occupera donc à chaque changement d'URL de masquer et afficher les composant nécessaire.

Par exemple, imaginons que nous nous trouvons sur une page d'accueil avec un header, une partie principale présentant tous les matchs du jour ainsi qu'un footer. Lorsque nous cliquons sur un de ces matchs, nous souhaitons que la partie principale, normalement chargée d'afficher le composant affichant tous les matchs du jour, soit masquée et que le composant permettant l'affichage des détails d'un seul match s'affiche.

Figure 22 : Exemple d'utilisation de <router-outlet> dans le template de notre composant racine

```
<nav class="grey darken-3" style="padding: 0px 10px">
  <div class="nav.wrapper">
    <a href="#" class="brand-logo">Liveball</a>
    <a href="#" data-target="mobile-nav" class="sidenav-trigger"
      ><i class="material-icons">menu</i></a>
    >
    <ul class="right hide-on-med-and-down">
      <li><a href="#">Accueil</a></li>
      <li><a href="#">Mes favoris</a></li>
      <li><a href="#">Ligues</a></li>
      <li><a (click)="toAuth()">S'identifier</a></li>
    </ul>
  </div>
</nav>

<body style="height: 100%">
  <router-outlet></router-outlet>
</body>

<br />
<footer class="page-footer grey darken-3">
```

(Code tiré de mon application LiveBall)

5.3.4 La gestion des routes par Angular

Dans la figure 21, nous pouvons observer deux routes un peu différentes de celles qui ont été décrites précédemment. Ces routes sont ce qu'on appelle des « routes générales ». Celle qui contient les guillemets vides, permet de dire à Angular à quel URL, il doit nous rediriger lorsque nous ne lui fournissons aucune information après le nom de domaine de notre application web. Puis, celle qui contient les « ** », permet de faire comprendre à Angular où nous souhaitons rediriger l'utilisateur lorsque qu'il fournit un chemin inexistant (erreur 404).

Quand nous fournissons un chemin, Angular va lire toutes les routes présentent dans notre liste de haut en bas. Il va donc parcourir chaque route présente dans notre liste jusqu'à trouver la route qui correspond à ce qu'on lui a donné.

Il est donc primordial de faire particulièrement attention à l'ordre dans lequel nous fournissons nos routes à Angular. Par exemple, si nous mettons notre route « ** » permettant la gestion de l'erreur 404 en premier, Angular nous redirigera à chaque fois au composant renseigné dans cette route. Cette dernière permet l'interception de n'importe quel URL, c'est pour cela qu'elle doit être placée en dernier afin de n'être utilisé qu'en dernier recours si aucune route correspondante n'a été trouvée avec l'URL fourni.

Pour conclure, les routes doivent être fournies de la plus spécifique à la plus générale afin que tout fonctionne correctement.

5.3.5 Les routes paramétrées

Le système de routing d'Angular nous permet, en plus de la navigation, de fournir un paramètre par l'URL qui pourra donc être utilisé sur la page en question. Par exemple, imaginons que nous avons une page affichant tous les matchs du jour avec l'URL « /matchs » et que nous souhaitons accéder à un match en particulier en cliquant dessus.

Dans ce cas, nous pouvons passer l'identifiant du match sur lequel nous avons cliqué directement en paramètre du router dans l'URL, cela permettra ensuite d'utiliser cet identifiant pour faire un appel à l'API en récupérant uniquement le match en question.

Pour mettre en place une route paramétrée, nous devons simplement lui définir un chemin suivi du paramètre précédé de l'opérateur « : », par exemple :

Figure 23 : Exemple de route paramétrée

```
{ path: "match/:id", component: DetailMatchComponent },
```

(Code tiré de mon application LiveBall)

La route sur la figure ci-dessus permet donc l'accès au composant « DetailMatchComponent » qui est chargé d'afficher le détail d'un match. Dans ce composant, nous pourrions utiliser notre paramètre « :id » qui sera l'identifiant du match sur lequel nous avons cliqué. Cela nous permettra de faire un appel à l'API en lui fournissant cet identifiant afin de récupérer toutes les données du match en question.

5.3.6 Utilisation de notre routing

Lorsque nous avons correctement implémenté notre routing, un dernier problème s'offre à nous. Lorsque nous cliquons sur un bouton ou un élément de notre barre de navigation, nous souhaitons être redirigés en utilisant nos routers.

Pour cela, il faut effectuer quelques opérations dans la logique d'un composant de notre choix. Premièrement, dans le constructeur de la classe de notre composant match par exemple, nous devons injecter le service permettant le pilotage de nos routes. Il faut commencer par importer l'élément « Router » du paquet « @angular/router ».

Une fois ceci fait, il suffit de créer une injection de dépendance pour « router » de type « Router » dans les parenthèses de notre constructeur. Il nous suffira d'utiliser la fonction « navigate » en lui fournissant entre parenthèse puis entre crochets, le chemin du composant souhaité ainsi que le paramètre si c'est une route paramétrée. Angular va ensuite parcourir toutes nos routes et s'il trouve celle qui correspond au chemin fournis, il affichera le composant demandé.

Si nous avons effectué un « router navigate » contenant un paramètre, nous devons ensuite le récupérer afin de pouvoir l'utiliser. Pour cela, nous avons besoin de l'élément « ActivatedRoute » du paquet « @angular/router ». Ensuite, dans la logique du composant sur lequel nous avons été redirigés, nous devons injecter, dans les paramètres du constructeur, « ActivatedRoute » en créant une injection de dépendance typée avec ce dernier.

Ensuite, cette injection de dépendance nous donnera accès à « snapshot » qui va récupérer la route courante suivie de « paramMap » qui va récupérer les paramètres transmis par la route sous forme de liste « clé, valeur ».

Pour finir, il nous suffit d'utiliser la fonction « get » avec le nom de notre paramètre entre parenthèses pour récupérer la valeur de ce dernier. Nous pouvons ensuite effectuer des opérations telles que des appels à l'API en fournissant ce paramètre.

Figure 24 : Utilisation d'un "router navigate" et d'une récupération de paramètre

```
displayMatch(fixture: Match) {  
  this.router.navigate(["/match", fixture.id]);  
}  
  
ngOnInit() {  
  const idMatch: string | null = this.router.snapshot.paramMap.get("id");  
  console.log(this.fixture);  
  if (idMatch && this.fixture == undefined) {  
    this.getFixtureById(Number(idMatch));  
  }  
}
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, nous pouvons observer le fonctionnement du passage entre la page contenant tous les matchs du jour et celle contenant le détail d'un seul match sur lequel l'utilisateur aurait cliqué.

Sur la partie du haut de l'image, on peut y voir le « router navigate » auquel je fournis un paramètre qui est l'identifiant du match. Puis, sur la partie du bas de l'image, je récupère le paramètre lors de l'initialisation du composant, je vérifie s'il est correctement récupéré et je le passe en paramètre de la fonction « getFixtureById » qui effectuera un appel à l'API pour récupérer les détails du match en question.

5.3.7 L'organisation de nos routes

Pour une petite application, le fichier « app-routing.module.ts » suffit à renseigner toutes les routes nécessaires au bon fonctionnement de l'application. Cependant, lorsque l'application est d'une taille plus conséquente et qu'elle est composée de plusieurs modules, les routes doivent être organisées de façon à rendre le code plus facile à maintenir.

En effet, nous avons la possibilité de renseigner nos routes spécifiques directement dans un module plus spécifique qui a pour objectif de gérer une partie précise de l'application. Cela nous permet d'être mieux organisé et de rendre le code bien plus lisible et maintenable.

Pour cela, il suffit de créer dans le module spécifique une liste de route similaire à celle présente dans « app-routing.module.ts ». Cette liste contiendra uniquement les routes permettant d'accéder aux composants gérés par le module spécifique. Ensuite, dans la partie « imports » de notre module, il faut ajouter l'élément « RouterModule.forChild(nomListe) » en lui fournissant la liste de nos routes. Cela permettra de lier ces dernières aux routes racines.

5.4 Les pipes

5.4.1 Présentation des pipes

Durant le développement de notre application, il se peut que nous tombions sur des données ayant un format « trop brut » pour être directement affichées. Les dates par exemple, sont très souvent fournies dans un format qui ne convient pas à tout le monde voire à personne. Par exemple, en JavaScript, les dates nous sont fournies dans un format contenant la date, l'heure et le fuseau horaire. Nous n'avons pas forcément besoin de toutes ces informations et c'est ici qu'interviennent les pipes.

Les pipes permettent d'effectuer des transformations sur nos chaînes de caractères, nous les utilisons pour les transformations fréquentes que nous serons amenés à effectuer plusieurs fois au long de la conception de notre application. Angular propose plusieurs pipes nativement :

- **DatePipe** : Permet de formater une date en utilisant les règles de notre région.
- **UpperCasePipe** : Met le texte en majuscule.
- **LowerCasePipe** : Met le texte en minuscule.
- **CurrencyPipe** : Permet de formater un nombre en « monnaie » en utilisant le symbole monétaire de notre région.
- **PercentPipe** : Permet de transformer un nombre en pourcentage.

En plus de ceux fournis par Angular, nous avons la possibilité de créer nos pipes personnalisés.

5.4.2 Utilisation des pipes

L'utilisation des pipes est plutôt simple et intuitive, il suffit d'ajouter l'opérateur « | » à côté de la chaîne de caractères que nous souhaitons transformer, suivi du nom du pipe que nous souhaitons utiliser. Si nous souhaitons utiliser plusieurs pipes en même temps pour la même chaîne de caractères, il suffit d'ajouter à nouveau l'opérateur « | » après le premier pipe suivi du nom du deuxième pipe que nous souhaitons utiliser.

Afin de pousser un peu plus loin l'utilisation des pipes, nous pouvons paramétrer ces derniers. Imaginons que nous utilisons le pipe natif « date », Angular formatera notre date en utilisant un type de format prédéfini pour notre région. Cependant, il se peut que nous ne soyons pas satisfaits et que nous souhaitons personnaliser le format. Pour cela, il suffit de mettre l'opérateur « : » après le nom de notre pipe, suivi du format que nous souhaitons entre guillemets.

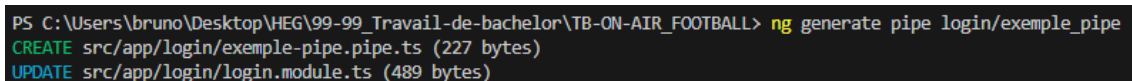
Pour finir, l'utilisation d'un pipe personnalisé se fait exactement de la même façon que les pipes natifs avec l'opérateur « | ».

5.4.3 Création d'un pipe personnalisé

Angular nous laisse la possibilité de créer des pipes personnalisés afin de répondre à des besoins spécifiques de notre application.

Afin de créer ce dernier, nous pouvons utiliser Angular CLI qui nous facilitera la tâche grâce à « ng generate ». Pour cela, il suffit d'écrire dans le terminal « ng generate pipe » suivi du nom que nous souhaitons donner à notre pipe, le chemin d'un répertoire peut précéder le nom du pipe si nous souhaitons le ranger dans une partie précise de l'application. Si aucun répertoire n'est précisé, le pipe sera créé dans le dossier de l'application et le module racine sera mis à jour. Tandis que si nous le rangeons dans une partie spécifique de l'application en fournissant le chemin d'un répertoire, le pipe sera créé là où on l'a souhaité et le module spécifique à cette partie de l'application sera mis à jour.

Figure 25 : Exemple de génération d'un pipe



```
PS C:\Users\bruno\Desktop\HEG\99-99_Travail-de-bachelor\TB-ON-AIR_FOOTBALL> ng generate pipe login/exemple_pipe
CREATE src/app/login/exemple-pipe.pipe.ts (227 bytes)
UPDATE src/app/login/login.module.ts (489 bytes)
```

(Mon CMD)

Dans l'exemple de la figure ci-dessus, nous pouvons voir que le pipe a été généré dans un répertoire spécifique nommé login et contenant son propre module. Le pipe a donc été créé dans ce dernier et c'est le module login qui a été mis à jour et non pas le module racine.

Lorsque le pipe est généré, nous avons un fichier contenant plusieurs parties. Premièrement, comme dans chaque fichier, nous avons les imports. De base, les seules importations sont « Pipe » qui est le décorateur du fichier et « PipeTransform » qui est l'interface nous permettant d'implémenter la méthode « transform ».

Ensuite, Nous avons notre décorateur « @Pipe » qui contient les propriétés du pipe, de base, il ne contient que le nom de ce dernier. Pour finir, nous avons la « class » qui contient la logique du pipe, dans cette classe nous n'avons qu'une seule méthode déjà implémentée nommée « transform ».

Cette méthode est le centre de notre pipe, c'est ici que nous allons effectuer les opérations de transformation de notre chaîne de caractères. Pour cela, nous pouvons commencer par supprimer ce qu'il y a dans les paramètres de notre méthode « transform » et remplacer par un simple paramètre de type string nommé comme nous

le souhaitons. Ensuite, dans le corps de la méthode, nous pouvons y coder les différentes opérations de transformations.

Figure 26 : Exemple de pipe

```
import { Pipe, PipeTransform } from "@angular/core";
...
@Pipe({
  name: "europeanQualification",
})
export class EuropeanQualificationPipe implements PipeTransform {
  transform(desc: string): { [klass: string]: any } {
    if (desc == null || desc == undefined) {
      return { border: "none" };
    }
    You, 4 days ago • test
    if (desc.includes("Promotion - Champions League (Group Stage: )")) {
      return { "border-left": "2px solid green" };
    } else if (
      desc.includes("Promotion - Champions League (Qualification: )")
    ) {
      return { "border-left": "2px solid yellow" };
    } else if (desc.includes("Promotion - Europa League (Group Stage: )")) {
      return { "border-left": "2px solid orange" };
    } else if (
      desc.includes("Promotion - Europa Conference League (Qualification: )")
    ) {
      return { "border-left": "2px solid blue" };
    } else if (desc.includes("Relegation - ")) {
      return { "border-left": "2px solid red" };
    } else if (desc.includes("(Relegation)")) {
      return { "border-left": "2px solid pink" };
    }
    return { border: "none" };
  }
}
```

(Code tiré de mon application LiveBall)

Dans ce pipe, la « description du classement d'un club » est récupérée. Elle est ensuite comparée à différentes chaînes de caractères afin de voir si oui ou non l'équipe est qualifiée à une compétition européenne. Si l'équipe est qualifiée, une bordure d'une couleur spécifique s'ajoute au texte sur son côté gauche. La relégation d'une équipe est aussi signalée à l'aide d'une bordure.

Le résultat de ce pipe est à retrouver en [annexe](#).

5.5 Les directives

5.5.1 Présentation d'une directive

Une directive, est une classe Angular fortement similaire à un composant, mais elle n'a pas de template. Il faut savoir qu'au sein d'Angular, la classe `Component` hérite de la classe `Directive`. Cette dernière permet d'interagir avec un élément du DOM et de lui attacher un comportement ou une apparence spécifique. La directive utilise le décorateur « `@Directive` » et possède un sélecteur CSS permettant d'indiquer à Angular où utiliser la directive dans nos templates.

Lorsque dans notre template Angular trouve une directive, le framework va instancier la classe de cette dernière et lui donner le contrôle sur les éléments du DOM qui lui reviennent.

Il existe plusieurs types de directives :

- **Les composants** : C'est le type de directive le plus courant. Un composant est simplement une directive avec un template. Par exemple, notre composant racine est une directive.
- **Les directives structurelles** : Ce type de directive, modifie notre DOM en ajoutant, supprimant ou remplaçant des éléments de ce dernier. Par exemple, « `ngIf` » et « `ngFor` » sont des directives structurelles.
- **Les directives d'attributs** : Ce type de directive permet la modification du comportement et de l'apparence des éléments du DOM, des attributs, des propriétés, etc. On retrouve généralement ces attributs directement dans les balises HTML.

Nous avons la possibilité de créer des directives d'attributs personnalisées.

5.5.2 `ngIf` et `ngFor`

Ces deux directives structurelles sont énormément utiles pour rendre nos templates dynamiques.

Premièrement, « `ngIf` » permet de conditionner l'affichage dans nos templates, nous allons donc pouvoir afficher ou supprimer des éléments du DOM uniquement à certaines conditions. Par exemple, cette directive est fortement utile dans un header, car nous pouvons cacher et afficher des éléments selon le type d'utilisateur actuellement connecté. Pour utiliser « `ngIf` » il suffit d'ajouter à l'intérieur d'une balise HTML « `div` », « `ng-container` » ou autre « `*ngIf= « condition »` »

Ensuite, « `ngFor` » permet de parcourir différents types de collections afin d'afficher tout le contenu de ces dernières dans notre vue. Par exemple, lorsque nous effectuons un appel à une API et qu'elle nous retourne une liste. Nous pouvons grâce à cette directive

fonctionnelle parcourir tout ce que nous avons récupéré afin de tout afficher dans notre template.

Sur LiveBall, « ngFor » est notamment utilisé pour afficher tous les matchs du jour en parcourant une liste reçue de la part d'une API. Pour utiliser « ngFor » il suffit d'ajouter à l'intérieur d'une balise HTML « div », « ng-container » ou autre « ngFor= « informations de parcours » ».

5.5.3 Création d'une directive d'attribut

Une directive d'attribut permet de changer le comportement ou l'apparence de nos éléments du DOM. Nous avons la possibilité d'en créer afin qu'elles répondent à des besoins spécifiques pour notre application.

Premièrement, nous pouvons générer notre directive grâce à Angular CLI. Pour cela, il nous suffit d'utiliser la commande « ng generate directive » suivi du nom de cette dernière. Comme pour tout le reste, la directive va être créée et le module concerné va être mis à jour.

Figure 27 : Exemple de création d'une directive

```
PS C:\Users\bruno\Desktop\HEG\99-99_Travail-de-bachelor\TB-ON-AIR_FOOTBALL> ng generate directive fixtures/border-card-select
CREATE src/app/fixtures/border-card-select.directive.ts (161 bytes)
UPDATE src/app/fixtures/fixtures.module.ts (1287 bytes)
```

(mon CMD)

Dans la figure ci-dessus, une directive a été générée dans le répertoire « fixtures » et le module spécifique gérant cette partie de l'application a été mis à jour.

Lorsque nous générons notre fichier de directive, nous avons plusieurs parties dans celui-ci. Premièrement, nous avons les imports. De base seule le décorateur « Directive » est importé.

Ensuite, nous avons notre décorateur avec à l'intérieur notre sélecteur CSS nous permettant d'identifier notre directive dans nos templates.

Pour finir, nous avons la classe contenant la logique de cette dernière nous permettant d'effectuer les opérations que nous souhaitons sur la partie du DOM qu'elle contrôlera.

Par exemple, sur l'application LiveBall, une directive a été créée, permettant de mettre en surbrillance la carte du match sur lequel l'utilisateur passe sa souris.

Figure 28 : Directive permettant la surbrillance d'une carte d'un match lorsque l'utilisateur passe sa souris dessus

```
1 import { Directive, ElementRef, HostListener } from "@angular/core";
2
3 You, 2 months ago | 1 author (You)
4 @Directive({
5   selector: "[appStlSelectGame]",
6 })
7 export class StlSelectGameDirective {
8   constructor(private el: ElementRef) {
9     this.setBackgroundColor("#373737");
10  }
11
12  @HostListener("mouseenter") onMouseEnter() {
13    this.setBackgroundColor("#DBDBDB");
14  }
15
16  @HostListener("mouseleave") onMouseLeave() {
17    this.setBackgroundColor("#373737");
18  }
19
20  setBackgroundColor(color: string) {
21    this.el.nativeElement.style.backgroundColor = color;
22  }
23
24 You, 2 months ago * ajout détail
```

(Code tiré de mon application LiveBall)

Dans la directive présente dans la figure ci-dessus, nous pouvons y voir quelques modifications par rapport au fichier généré directement par Angular. Premièrement, j'ai effectué de nouveaux imports. Le premier est « ElementRef » qui permet de récupérer l'élément HTML sur lequel j'ai appelé la directive afin d'y effectuer quelques modifications. Ensuite, j'ai utilisé « HostListener » qui permet de capturer un événement particulier de l'utilisateur.

Dans la logique de la directive, nous pouvons visualiser l'utilisation de « ElementRef » dans les paramètres du constructeur. Je fournis directement une couleur dans ce dernier à l'aide de la méthode « setBackgroundColor » afin qu'il l'attribue à l'élément dès l'initialisation de la classe de la directive.

Ensuite, j'utilise plusieurs « HostListener » qui me permettent de définir l'événement qui doit se produire pour changer la couleur de la carte. Nous pouvons observer deux événements, un qui se déclenche lorsque l'utilisateur met sa souris sur la carte, puis un qui remet la couleur d'origine lorsque l'utilisateur enlève sa souris.

Pour finir, nous avons la méthode « setBackgroundColor » qui est utilisée dans toute la directive. Elle utilise « el » qui est la variable de type « ElementRef » qui permet l'accès à certaines propriétés de l'élément du DOM qui appelle la directive. Dans ce cas, c'est

une propriété « `style.backgroundColor` » qui permet la modification de l'arrière-plan de la carte selon la couleur donnée en paramètre.

5.5.4 Utilisation de notre directive

Lorsque nous avons terminé le développement de notre directive, nous pouvons l'utiliser dans notre template. Pour cela, il suffit d'utiliser le sélecteur CSS attribué à notre directive en l'appelant dans la balise que l'on souhaite contrôler.

Dans le cas de la directive vu précédemment dans la figure, nous pouvons appeler notre directive directement dans la carte contenant les informations de matchs.

Figure 29 : Exemple d'utilisation d'une directive

```
<div class="card grey darken-3" style="margin-bottom: 0px">
  <div
    class="card-content white-text"
    appStlSelectGame
    (click)="displayMatch(fixture)" | You, 2 months ago • a
  >
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, nous pouvons observer qu'un simple appel du sélecteur CSS de la directive « `appStlSelectGame` » dans la balise qui nous intéresse, suffit à utiliser cette dernière.

5.5.5 Les directives paramétrées

Nous avons la possibilité d'ajouter un paramètre à notre directive. Imaginons que nous utilisons la même directive dans deux templates différents. Dans l'un des deux, nous souhaitons que lorsque l'utilisateur passe sa souris sur la carte, elle devienne bleue. Alors que dans l'autre, nous souhaitons qu'elle devienne grise.

Pour cela, nous avons besoin de passer un paramètre à notre directive lors de son appel afin qu'elle sache quelle couleur utiliser lorsqu'elle va changer l'apparence de nos éléments du DOM.

Figure 30 : Exemple de directive paramétrée

```
import { Directive, ElementRef, HostListener, Input } from "@angular/core";

You, 11 seconds ago | 1 author (You)
@Directive({
  selector: "[appStlSelectGame]",
})
export class StlSelectGameDirective {
  constructor(private el: ElementRef) {
    this.setBackgroundColor("#373737");
  }

  @Input("appStlSelectGame") backgroundColor: string;

  @HostListener("mouseenter") onMouseEnter() {
    this.setBackgroundColor(this.backgroundColor || "#DBDBDB");
  }
}
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, on peut voir qu'il faut premièrement importer « Input » du cœur d'Angular.

Ensuite, je crée mon « input » à l'aide de « @Input » en lui fournissant entre parenthèses le nom du sélecteur CSS de la directive. Puis, je lui donne un nom et un type.

Pour conclure, cet « input » aura comme valeur le nom d'une couleur ou le code hexadécimal de cette dernière et il est passé en paramètre de la méthode « setBackgroundColor ».

Dans certains cas, il se peut que je ne fournisse aucun paramètre à ma directive lors de son appel, la valeur de mon « Input » sera donc « undefined » ce qui peut poser un problème. Pour régler cela, j'ajoute l'opérateur « || » suivi d'une couleur en brut qui permettra de faire en sorte que si l'input est égal à « undefined », j'utiliserais la couleur se trouvant à droite de l'opérateur « || ».

5.6 Les services

5.6.1 Présentation des services

Un service est une classe qui va permettre de centraliser certaines fonctionnalités et opérations afin de pouvoir les utiliser dans plusieurs parties de notre application.

Généralement, un service aura une fonction bien précise. Par exemple, nous pouvons en créer un qui sera chargé de communiquer avec un serveur distant, ce dernier sera une classe composée de plusieurs méthodes accessibles par plusieurs composants afin qu'ils aient accès aux données.

Un service utilise le décorateur « @injectable », car c'est une classe dites « injectable » ce qui veut dire que nous pouvons utiliser le service d'injection de dépendances dans le

constructeur de n'importe quel composant afin d'avoir accès aux méthodes que le service propose. Le fait d'utiliser l'injection de dépendance permet à tous les composants utilisant le service, d'avoir accès à la même instance de ce dernier. Aucun composant ne devra instancier ce dernier ce qui fait qu'ils auront tous accès à la même version de la ou les données récupérées.

Nous avons aussi la possibilité d'accéder à un service dans un autre service. Par exemple, pour effectuer des appels à une API, un service est fourni par Angular et doit être injecté dans notre service personnalisé.

Pour conclure, ils nous permettent d'éviter fortement la redondance, car ils nous donnent la possibilité de centraliser plusieurs méthodes utilisées par plusieurs composants différents, ils nous permettent de fournir à tous les composants les mêmes versions des données, car comme un « singleton⁹ » une seule instance sera accessible grâce à l'injection de données.

5.6.2 Création d'un service

Lorsque nous sommes dans le cas où nous aurions besoin d'accéder aux mêmes méthodes dans plusieurs composants, nous pouvons créer un service.

Premièrement, nous aurons besoin d'Angular CLI pour la génération de ce dernier. Pour cela, il suffit d'utiliser « `ng generate service` » suivi du nom que nous souhaitons donner à ce dernier. Nous avons la possibilité de précéder le nom du service d'un nom de répertoire si nous souhaitons y créer notre service.

Figure 31 : Exemple de création d'un service

```
PS C:\Users\bruno\Desktop\HEG\99-99_Travail-de-bachelor\TB-ON-AIR_FOOTBALL> ng generate service leagues/league-data  
CREATE src/app/leagues/league-data.service.ts (139 bytes)
```

(Mon CMD)

Dans la figure ci-dessus, un service a été créé dans le répertoire « leagues ».

Lorsque notre service est généré, nous pouvons observer trois parties dans ce dernier. Premièrement, les imports, au début seul le décorateur « injectable » est importé. Ensuite, nous avons le décorateur « @injectable », avec à l'intérieur un élément nommé « providedIn » avec comme valeur « root ». Cet élément nous permet de définir le service comme accessible dans toute l'application. Puis, nous avons la classe de notre service qui représente la logique de ce dernier et qui contiendra toutes les méthodes et opérations.

⁹ Design pattern permettant d'obliger la classe à n'avoir qu'une seule instance.

Par exemple, sur l'application LiveBall, les services sont utilisés pour tous les appels à l'API. Nous y retrouvons des méthodes permettant de récupérer les matchs par date, par identifiant, etc.

Une image du service est disponible en [annexe](#).

5.6.3 Utilisation d'un service

Lorsque nous souhaitons utiliser un de nos services, nous ne devons surtout pas l'instancier, car cela enlèverait tout l'intérêt de ce dernier qui consiste à n'avoir qu'une seule instance partagée par tous ceux qui l'utilisent.

Pour utiliser correctement un service, il faut utiliser le système d'injection de dépendance d'Angular comme on a déjà pu le faire avec le « Router ». Pour cela, sur la page où nous souhaitons utiliser notre service, nous devons premièrement l'importer tout en haut du fichier. Une fois ceci fait, il faut injecter le service en l'ajoutant dans les parenthèses du constructeur. Il faut commencer par lui donner un nom comme une simple variable, puis lui donner comme type le nom du service après l'opérateur « : ».

Figure 32 : Exemple d'injection de service dans un composant

```
constructor(private fds: FootballDataService, private router: Router) {}
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, nous pouvons y voir un exemple d'injection de service nommé « fds » et de type « FootballDataService » dans le constructeur d'un composant. Ce service permet l'accès à des méthodes permettant d'effectuer des appels à une API. Pour l'utiliser dans la suite du code, il suffit d'utiliser ce type d'appel « this.fds.getFixturesByDate(date) ». Dans ce cas, j'utilise la méthode « getFixturesByDate » de notre service qui se chargera de me retourner tous les matchs pour une date donnée.

Grâce aux services, nous avons maintenant centralisé l'accès à nos données. Ce qui veut dire que si nous avons une modification à faire (par exemple, une modification de l'endpoint¹⁰), il n'y aura qu'un seul changement à faire du côté du service.

5.6.4 Niveau d'accès au service

De base, lors de la génération de notre service, dans le décorateur « @Injectable », nous retrouvons une propriété « providedIn : « root » ». Cette propriété rend le service

¹⁰ URL permettant l'accès aux données d'une API

accessible par une seule et même instance dans absolument toute l'application. Nous pourrions donc avoir accès à ce service dans n'importe quel composant.

Dans certains cas, nous allons vouloir limiter l'accès à ce service à une partie définie de l'application. Pour cela, nous pouvons le limiter à un module qui lui est chargé de gérer une partie de l'application. Premièrement, il faut supprimer la propriété « `providedIn` » de notre service. Ensuite, il faut ajouter une propriété « `providers` » à notre module spécifique. Pour finir, après l'opérateur « `:` », il faut ajouter une liste et y mettre le service concerné. Nous avons maintenant accès au service uniquement dans la partie de l'application gérée par ce module spécifique. Il est aussi possible de limiter l'utilisation d'un service à un composant, mais cela enlève l'intérêt principal de ce dernier.

5.7 Les modules

5.7.1 La modularité sur Angular

Angular est un framework permettant la création d'applications modulaires, de plus il possède son propre système de modularité qui se nomme « `NgModule` ». Le but d'un module, est de regrouper un ensemble de fonctionnalités liées entre elles.

Lors de la génération de notre application, nous avons d'office notre module racine dans lequel nous faisons référence au composant racine ainsi qu'à d'autres modules nécessaires au bon fonctionnement de l'application.

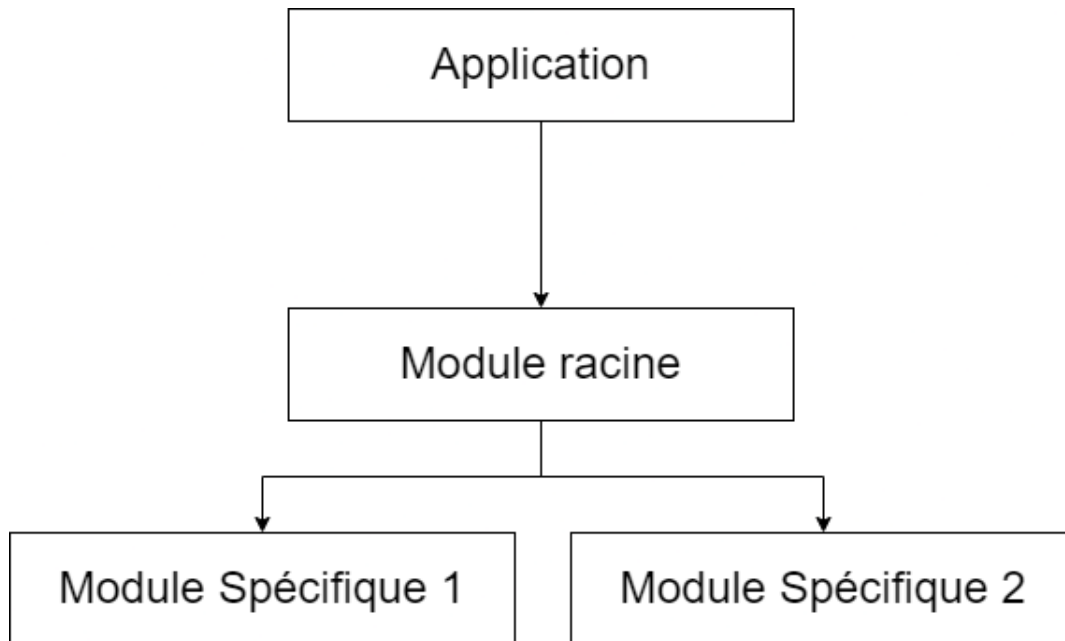
Cependant, quand l'application s'agrandit et que nous avons de plus en plus de fonctionnalités, il peut être difficile de gérer l'application avec le module racine uniquement. C'est pourquoi nous pouvons créer des modules spécifiques pour des ensembles de fonctionnalités, ce qui rend l'application bien plus organisée et plus facile à maintenir.

Tous les modules spécifiques créés durant le développement de l'application doivent être référencés dans le module racine. Cela nous permet de n'avoir qu'une seule ligne à ajouter dans ce dernier au lieu de devoir ajouter tous les composants ainsi que leurs services, pipes et autres dépendances.

Pour mieux comprendre l'utilité d'un module, nous pouvons imaginer un grand sac qui représente le module racine contenant toutes sortes d'objets qui ne sont pas forcément triés. Dès que nous souhaitons chercher quelque chose, nous aurons beaucoup de mal à le trouver, car tout est mélangé et en désordre. Du coup, nous avons la possibilité de regrouper tous les objets du même type dans un plus petit sac étiqueté qui représente un module spécifique. Ce plus petit sac, peut ensuite être rangé dans le gros sac et cela nous permettra d'être bien plus organisés.

Si nous prenons maintenant un exemple concret avec un système d'authentification contenant les fonctionnalités permettant de se connecter et de s'inscrire, nous pouvons regrouper le composant pour la connexion ainsi que celui pour l'inscription dans un module spécifique nommé « authentification ». De plus, nous incluons également tous les pipes et services uniquement utilisés par ces fonctionnalités. Pour finir, nous référencerions le module spécifique dans le module racine.

Figure 33 : Représentation de la modularité dans une application Angular



(Reconstitution inspirée de : <https://www.udemy.com/course/angular-developper-tutorial-application-typescript/learn/lecture/31653906#overview>)

5.7.2 Contenu d'un module

Tout comme le module racine, un module spécifique est composé de plusieurs points :

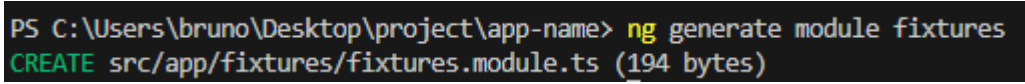
- « **declarations** » : C'est dans cet élément que l'on déclare la liste de nos composants, de nos pipes, etc.
- « **imports** » : Dans cet élément, nous pouvons lui fournir tous les autres modules dont nous avons besoin. Ce dernier n'accepte que des modules, nous ne pouvons pas lui fournir de composant ici. Pour résumer, il faut mettre tout ce qui n'est pas un module dans l'onglet « declarations » et tous les modules dans « imports ».
- « **providers** » : Permet d'utiliser le « système d'injection de dépendance » d'Angular.

On peut voir que l'on retrouve tous les points du module racine, sauf « bootstrap », car ce point doit être uniquement présent dans ce dernier.

5.7.3 Création d'un module

Dès que nous souhaitons créer un module afin de regrouper un ensemble de fonctionnalités liées, nous devons utiliser la commande « `ng generate module` » suivi du nom que nous souhaitons donner au module par exemple « `fixtures` ». Cela créera un module permettant la gestion de toutes les fonctionnalités ayant un rapport avec les matchs.

Figure 34 : Génération d'un module avec Angular CLI



```
PS C:\Users\bruno\Desktop\project\app-name> ng generate module fixtures
CREATE src/app/fixtures/fixtures.module.ts (194 bytes)
```

(Screen personnel)

Une fois le module généré, un répertoire est automatiquement créé avec le module à l'intérieur. Par la suite, il faudra lui fournir toutes les dépendances dont il aura besoin pour gérer sa « partie » de l'application. Pour des raisons d'organisation, il est fortement conseillé de mettre tous les composants, pipes et autres types de dépendances directement dans le répertoire du module.

Si un composant ou un autre type de dépendance est directement généré dans le répertoire du module, Angular CLI mettra automatiquement à jour le bon module spécifique, si aucun répertoire n'est indiqué lors de la génération de ces derniers, c'est le module racine qui sera mis à jour.

Il y a bien évidemment toujours la possibilité de réorganiser une application qui ne fonctionnait qu'avec le module racine. Il suffit de déplacer les différents fichiers dans le dossier du module concerné, lui attribuer les dépendances, enlever les dépendances du module racine et lui fournir la référence du module spécifique dans « `imports` ».

Figure 35 : Exemple de module pour l'ensemble de fonctionnalités "fixtures"

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { ListMatchesComponent } from "../list-matches/list-matches.component";
import { DetailMatchComponent } from "../detail-match/detail-match.component";
import { StlSelectGameDirective } from "../stl-select-game.directive";
import { Routes, RouterModule } from "@angular/router";
import { NgxPaginationModule } from "ngx-pagination";
import { MatTabsModule } from "@angular/material/tabs";
import { FixtureDataService } from "../fixture-data.service";
import { DetailLeagueComponent } from "../leagues/detail-league/detail-league.component";

const fixturesRoutes: Routes = [
  { path: "league/:id", component: DetailLeagueComponent },
  { path: "matches/", component: ListMatchesComponent },
  { path: "match/:id", component: DetailMatchComponent },
];

You, last month | 1 author (You)
@NgModule({
  declarations: [
    ListMatchesComponent,
    DetailMatchComponent,
    StlSelectGameDirective,
  ],
  imports: [
    CommonModule,
    RouterModule.forChild(fixturesRoutes),
    NgxPaginationModule,
    MatTabsModule,
  ],
  providers: [FixtureDataService],
})
export class FixturesModule {}
```

(Code tiré de mon application LiveBall)

5.8 Les requêtes http

5.8.1 Présentation du HttpClientModule

Lorsque nous développons une application web, nous avons de grandes chances d'avoir à communiquer avec un serveur distant. Dans le cas où nous aurions besoin de communiquer avec une API, Angular fournit un client prêt à l'emploi permettant la communication avec cette dernière. Ce client se nomme le « HttpClientModule ».

Nous avons uniquement besoin de l'importer dans notre module racine afin qu'il puisse être utilisable dans toute l'application. Une fois ceci fait, nous aurons la possibilité de communiquer avec notre API.

5.8.2 Mise en place du « HttpClientModule »

Afin de mettre correctement en place notre « client http » fourni par Angular, nous aurons plusieurs étapes à suivre.

Premièrement, nous devons importer ce dernier dans notre module racine. Pour cela, il suffit de l'importer directement depuis le paquet « @angular/common/http » puis d'ajouter l'élément « HttpClientModule » à la section « imports » de notre module racine.

Une fois ceci fait, nous aurons la possibilité d'utiliser le client dans nos services ou nos composants. Pour cela, nous devons utiliser le service d'injection de dépendance d'Angular en injectant « HttpClient » et non pas « HttpClientModule » dans notre service ou notre composant. Une fois ceci fait, nous avons accès aux différentes méthodes.

Voici quelques exemples, en imaginant que notre injection de « HttpClient » se nomme « http » :

- « **this.http.get(url)** » : requête permettant de récupérer des données.
- « **this.http.post(url)** » : requête permettant d'envoyer des données afin de les stocker.
- « **this.http.delete(url)** » : requête permettant de supprimer des données.

Ces méthodes demanderont chacune des paramètres tels que « l'endpoint » que nous souhaitons requêter ainsi que l'en-tête et le paramètre si besoin.

5.8.3 Récupérer des données

Lorsque nous récupérons des données directement dans une liste en TypeScript, nous n'avons pas de délai de réponse, car nous recevons des données « synchrone ». Mais lorsque nous souhaitons récupérer des données depuis un serveur distant, les données récupérées auront un délai qui peut poser un problème si nous utilisons des méthodes traditionnelles pour les gérer.

Nous devons maintenant être capables de coder la récupération et l'utilisation de ces données en prenant en compte que nous ne les recevons pas à la milliseconde.

La première chose à prendre en compte est que nous ne pouvons pas savoir à l'avance le type des données que nous récupérerons. Avec une liste « TypeScript » classique, nous pouvions directement la « typer » ce qui fait que nous n'avons pas à gérer ce cas. Cependant, le type de listes ou d'objets que nous recevons de notre serveur distant n'est pas prévisible et nous devons donc utiliser le type « any » qui lui acceptera tous types de données.

Ensuite, étant donné que nous travaillons avec des données « asynchrones », il est conseillé d'utiliser un « observable ». Étant donné qu'il y a un délai avant l'arrivée des données, ce dernier s'occupera de les récupérer et de nous les retourner lorsqu'elles seront arrivées. Cela fonctionne avec un système d'abonnement, il faudra donc

« s'abonner » à « l'observable » afin qu'il nous notifie et nous fournisse les données lorsqu'elles nous ont été envoyées par le serveur distant.

Imaginons que nous avons une méthode « getMatchesById » qui récupère, traite et retourne un match selon son identifiant. Lorsque nous choisissons le type que l'on souhaite donner à notre retour, nous ne pouvons pas lui donner le type « Match », car il ne comprendra pas directement le lien entre le type « Match » et les données reçues depuis le serveur distant.

Le fait d'utiliser comme type de retour « Observable<any[]> » fera en sorte que la méthode retourne un « Observable » qui lui retournera l'objet de type « any » quand il le recevra depuis le serveur distant.

Figure 36 : Exemple de méthode retournant un observable

```
getFixturesById(id: number): Observable<any[]> {  
  const headers = new HttpHeaders({  
    "x-rapidapi-key": environment.apiToken,  
    "x-rapidapi-host": environment.apiHost,  
  });  
  
  const params = {  
    id: id,  
  };  
  
  return this.http.get<any[]>(this.apiUrlFixtures, { headers, params }).pipe(  
    tap((res) => console.log("OK match !")),  
    catchError((err) => {  
      console.log(err);  
      return of([]);  
    })  
  );  
}
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, nous pouvons observer que je retourne une liste de type « any[] » à l'aide de la méthode « get » présente dans mon injection de dépendance nommée « http ». Je lui donne « l'endpoint », les « headers » et les « paramètres » afin qu'il fasse la requête « GET » souhaitée.

J'utilise ensuite l'opérateur « pipe » qui permet de définir ce que je souhaite faire lorsque la requête a été effectuée. Le « tap » me permet de définir ce que je souhaite effectuer lorsque cette dernière s'est bien déroulée tandis que « catchError » permet de définir quoi faire en cas d'erreur.

5.8.4 Utiliser nos données asynchrones

Lorsque tout cela a été implémenté, nous sommes prêts à « consommer » nos données dans notre composant.

Imaginons maintenant que nous souhaitons récupérer les matchs par leurs identifiants directement dans notre composant en utilisant la méthode « getFixturesById » créée dans notre service.

Premièrement, il faut prendre en compte que notre méthode « getFixturesById » ne retourne pas directement notre liste, mais un « Observable ».

Figure 37 : Exemple de consommation de données asynchrones

```
getFixtureById(id: number) {
  this.fds.getFixturesById(id).subscribe((res: any[]) => {
    this.fixtureAny = (res as any).response[0];
    console.log(this.fixtureAny);
    this.events = this.fixtureAny.events;
    this.lineups = this.fixtureAny.lineups;
    this.statistics = this.fixtureAny.statistics;
    this.players = this.fixtureAny.players;
    let fDateTotal = this.fixtureAny.fixture.date;
    let fDateMatch = fDateTotal.split("T")[0];
    let Hour = fDateTotal.split("T")[1];
    let d = new Date(Date.parse(fDateTotal));
    let utcHours = d.getUTCHours();
    let utcMinutes = d.getUTCMinutes();
    let offsetHours = 2; // Heure de Paris
    let localHours = utcHours + offsetHours;
  });
}
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, une méthode « getFixturesById » a été créée. Dans celle-ci, j'appelle une méthode du même nom qui est présente dans l'injection de dépendance « fds », qui est mon service Angular.

Lorsque j'appelle cette méthode, présente dans mon service, j'utilise l'opérateur « subscribe » qui comme son nom l'indique, me permet de m'abonner à mon « Observable » afin de recevoir les données une fois qu'elles ont été reçues.

Ensuite, « res » représente ce que me fournit mon « Observable », qui dans ce cas représente mon match ayant l'identifiant donné en paramètre. Puis, entre crochets, nous retrouvons tous les traitements effectués aux données afin qu'elles correspondent à ce que je souhaite.

Une fois tous les traitements effectués, nous avons la possibilité de créer une instance de l'objet souhaité en lui fournissant toutes les données. Cela permettra d'obtenir finalement, un objet qui sera forcément plus facile à utiliser étant donné que nous l'avons personnellement implémenté. Une figure représentant la création de l'instance est disponible en [annexe](#).

5.9 Les formulaires

5.9.1 Les formulaires dans le développement web

Les formulaires sont une partie très importante du développement web, ils sont présents dans une grande partie des applications actuellement en ligne et permettent de répondre à beaucoup de besoin.

Ils peuvent être difficiles à mettre en place due au nombre élevé de règles de gestion auxquelles il faut réfléchir et implémenter. Angular offre des solutions pour faciliter la conception de ces derniers principalement avec les modules « FormsModule » et « ReactiveFormsModule » provenant de la librairie « @angular/forms ».

Avant de mettre en place quoi que ce soit, il faut premièrement importer « FormsModule ». Pour cela, il suffit de l'ajouter dans la partie « imports » du module concerné. Si nous souhaitons qu'il soit disponible dans toute l'application, il faut l'ajouter dans le module racine tandis que si nous souhaitons l'implémenter uniquement dans une partie de l'application, il faut le fournir dans le module spécifique concerné.

5.9.2 NgForm

Lorsque nous importons le module « FormsModule », toutes les balises « form » du module concerné utiliseront « NgForm » sans que nous n'ayons besoin d'effectuer une quelconque opération supplémentaire.

La directive « NgForm » va créer une instance nommée « FormGroup » pour chaque balise « form ». Une référence à cette directive nous permettra de contrôler si le formulaire est valide tout au long du remplissage de ce dernier et elle nous permettra d'être notifiés lorsque l'utilisateur soumettra le formulaire.

Figure 38 : Balise HTML « form » avec ngForm

```
<form
  #registerForm="ngForm"
  (ngSubmit)="onSubmitRegister()"
  class="col s12"
  style="margin-top: 120px; margin-bottom: 200px"
>
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, nous pouvons observer une balise « form » utilisant une variable référencée dans le « template » qui est égale à « ngForm ». Cela me permet de stocker dans cette variable nommée « registerForm » toutes les informations du formulaire.

Par exemple, nous avons la possibilité de vérifier si le formulaire est totalement valide avant de rendre un bouton cliquable.

Figure 39 : Bouton vérifiant si le formulaire est valide avant d'être cliquable

```
<button
  class="btn waves-effect waves-light"
  type="submit"
  [disabled]="
    !registerForm.form.valid || user.password != confirmPasswordUser
">
```

(Code tiré de mon application LiveBall)

Comme nous pouvons le voir sur cette figure, le bouton sera désactivé tant que le formulaire n'est pas valide. Pour cela, j'utilise la propriété « [disabled] », ensuite dans les conditions lui permettant de savoir quand il doit se désactiver, j'utilise ma variable référencée qui est capable de me fournir « l'état » du formulaire.

5.9.3 NgModel

La directive NgModel, doit être appliquée sur chacun des champs de notre formulaire afin d'avoir notre liaison bidirectionnelle avec la logique de notre composant. Cela nous permettra, si nous le lions avec « NgForm », d'avoir un contrôle sur la qualité des données saisies, afin de prévenir l'utilisateur en cas de non-validité de ces dernières.

Cette directive, si elle est référencée dans nos champs, créera une instance nommée « FormControl ». Tout cela, nous permettra d'accéder aux entrées de l'utilisateur directement dans notre logique.

Chaque instance de « FormControl » doit avoir un nom qui lui est attribué afin de pouvoir y accéder ensuite. Pour cela, il suffit d'utiliser la propriété « name » dans nos balises HTML.

Grâce à cette directive, nous pouvons avoir un suivi sur toutes les interactions effectuées par l'utilisateur. Nous aurons directement accès à l'état de notre champ, s'il a été modifié ou si une case a été cochée par exemple. Grâce à cela nous pourrions par exemple, afficher dynamiquement une couleur ou un message d'erreur à l'utilisateur pour le prévenir d'un éventuel problème.

Figure 40 : Exemple d'utilisation de ngModel

```
<input
  id="registerEmail"
  type="email"
  name="email"
  [(ngModel)]="user.email"
  #email="ngModel"
  required
  (change)="log()"
/>
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, Nous pouvons observer un exemple d'utilisation de « ngModel ». Premièrement, je fournis à « ngModel » sa liaison avec la propriété « email » de mon objet « user » qui lui est présent dans la partie logique de mon composant.

Cela va me permettre d'effectuer la liaison bidirectionnelle et va faire en sorte que « user.email » soit toujours égal à la valeur de « l'input » de notre « template ». Cela me permettra d'avoir un contrôle en direct et en continu sur l'entrée fournie par l'utilisateur.

Ensuite, je donne un nom à mon « ngModel » en utilisant une variable référencée dans le « template » qui est elle aussi égale à « ngModel ». Cette variable nommée « email » me permettra d'avoir accès à toutes les informations de mon « input » comme son état ou sa valeur par exemple.

5.9.4 Les règles de validation

Il est essentiel de mettre en place des règles de validation afin d'obtenir des données de qualités provenant de notre formulaire. Angular nous facilite fortement la tâche, il est capable de se lier directement aux règles de validation fournie par l'HTML et de les utiliser afin d'afficher un message d'erreur à l'utilisateur par exemple.

Figure 41 : Exemple d'implémentation d'un message d'erreur en cas de champ non valide

```
<input
  id="registerEmail"
  type="email"
  name="email"
  [(ngModel)]="user.email"
  #email="ngModel"
  required
  (change)="log()"
/>
<label for="registerEmail">Email</label>
<div
  [hidden]="email.valid || email.pristine"
  class="card-panel red accent-1"
>
  Email is required
</div>
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, nous pouvons observer un champ qui doit être obligatoirement rempli. Pour cela, j'utilise simplement la propriété HTML « required ».

Ensuite, nous pouvons observer en dessous une « div » utilisant la propriété « [hidden] » permettant de cacher un élément ou non selon une ou des conditions. Les conditions mise en place dans cet exemple vérifient simplement si « l'input » nommé « email » est valide.

La puissance d'Angular dans ce cas est sa capacité à prendre en compte toutes les règles de validations HTML mise en place pour le champ et de faire apparaître un message d'erreur tant que toutes ses contraintes ne sont pas remplies.

De plus le « email.pristine » permet simplement d'éviter que lorsque l'utilisateur arrive sur la page tous les messages d'erreur soit en place. Cela oblige au minimum une interaction avec le champ avant de faire apparaître le message

En conclusion, cela nous économise beaucoup de temps, car nous n'avons pas besoin de coder ces contraintes nous-mêmes.

5.10 Les tests sur Angular

5.10.1 Pourquoi tester son application ?

Le but d'un test est de vérifier qu'une partie de son application fonctionne correctement. Cela permet de garantir la qualité de la solution.

Bien que nous puissions effectuer ces tests manuellement cela peut s'avérer être une grosse perte de temps. C'est pourquoi il est essentiel de mettre en place des tests automatisés sous forme de code. Ces tests vont vérifier si les différentes parties de notre application fonctionnent correctement lorsqu'on les exécute.

L'automatisation des tests nous permet de n'avoir qu'un simple script à exécuter pour vérifier que toute notre application va bien lorsque nous avons ajouté une nouvelle fonctionnalité par exemple. De cette façon, nous pouvons nous assurer que la nouvelle fonctionnalité n'a pas introduit de bugs dans des parties existantes de l'application.

5.10.2 Les différents outils pour tester son application Angular

Il existe plusieurs outils permettant de tester une application Angular :

- Jasmine testing
- Karma Jasmine HTML reporter
- Cypress
- Selenium

Tous ces outils ont leurs propres qualités et défauts et proposent différentes fonctionnalités qui peuvent être plus ou moins intéressantes selon ce que nous cherchons à effectuer.

Il est important de savoir que « Jasmine testing » et « Karma Jasmine HTML reporter » sont automatiquement installés dans notre projet Angular lorsque nous le générons sauf si celui-ci a été généré avec l'option « --minimal ». Il peut donc être intéressant d'utiliser ces solutions.

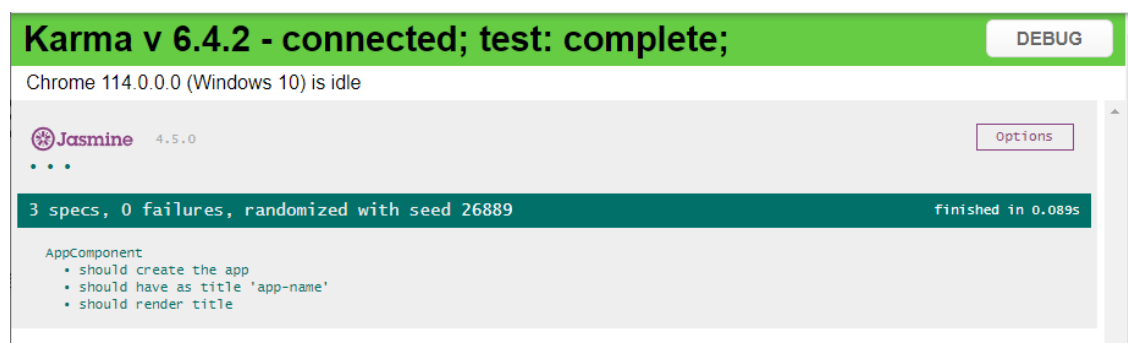
Premièrement, lors de la génération du projet, nous pouvons voir un fichier nommé « app.component.spec.ts » présent avec tous les autres fichiers dans le répertoire « app ». Ce fichier contiendra certains tests, mais « Karma » prendra tous les fichiers se terminant par « spec.ts » comme des fichiers de test. Il est donc possible de séparer ces tests en plusieurs fichiers, cela permet de mieux organiser son application et d'avoir des fichiers qui contiendront chacun le code chargé de tester une partie de l'application.

Le fichier « app.component.spec.ts » peut être considéré comme le fichier de test « racine ». Il contient de base, quelques tests permettant par exemple de vérifier si le composant racine se crée correctement.

Nous pouvons générer le fichier de configuration de « Karma » avec Angular CLI en utilisant la commande « ng generate config karma » si nous souhaitons personnaliser notre expérience.

Une fois, tout cela mis en place, il suffit d'utiliser la commande « ng test » pour les lancer. Cela ouvrira une page en « localhost » contenant les résultats de nos tests.

Figure 42 : Résultat de tests avec Karma



(Screen personnel)

Dans la figure ci-dessus, nous pouvons observer que trois tests ont été effectués et qu'ils sont tous correctement passés.

6. Le déploiement de notre solution Angular

6.1 Où déployer mon application Angular

Il existe une multitude d'hébergeurs capable d'héberger notre application Angular afin qu'elle soit disponible en ligne.

Pour cela, il faut s'intéresser aux hébergeurs compatibles avec du Node.js, ces derniers seront capables de télécharger Angular CLI ainsi que toutes les dépendances, dont le projet à besoin.

Voici quelques hébergeurs permettant le déploiement d'une application Angular :

- Hostinger
- Netlify
- PlanetHoster
- Hidora
- Firebase hosting

Ce n'est qu'un simple exemple de différents hébergeurs permettant le déploiement de solution Node.js.

6.2 Mise en place du déploiement

Avant de pouvoir déployer correctement son application, il faut pouvoir la construire (build), c'est à cette étape que notre code « TypeScript » va être transcompilé en « JavaScript » afin de pouvoir être correctement interprété par le navigateur.

Le but principal de cette étape, est de transformer notre code afin d'obtenir des fichiers statiques pour qu'ils puissent être utilisés par un serveur. Plusieurs étapes se dérouleront, dont l'utilisation du « TypeScript compiler » qui va transcompiler notre « TypeScript » en « JavaScript ».

Sur Angular, la commande permettant le « build » est « ng build », cette commande va faire passer une série de tests bien plus stricts que « ng serve ». Une fois que toutes les erreurs détectées par le « build » sont corrigées et que ce dernier s'exécute correctement, un dossier « dist » est créé contenant ce qui doit être déployé. Ensuite, nous sommes prêts au déploiement. Il suffira ensuite de trouver l'hébergeur qui nous convient et de suivre la marche à suivre de ce dernier.

6.3 Exemple de déploiement sur Firebase Hosting

Lorsque notre « build » s'est correctement déroulé, et que nous avons notre dossier « dist » prêt à l'emploi, nous pouvons effectuer un déploiement afin de retrouver notre solution en ligne.

Avec Firebase Hosting, tout peut être fait directement dans l'invite de commande de notre projet. La première étape est de créer un projet sur Firebase, une fois ceci fait nous avons accès à un tableau de bord nous permettant plusieurs possibilités. Ce qui nous intéresse est uniquement la partie hosting disponible dans la section créée.

Nous allons maintenant devoir installer le CLI de FireBase. Pour cela, nous avons besoin de notre installateur de paquet « npm ». Il suffit d'exécuter la commande suivant afin d'installer ce dernier de façon global utilisable pour tous nos projets « npm install -g firebase-tools ». Une fois installé, nous pouvons vérifier que tout s'est bien déroulé avec la commande « firebase --version » et nous connecter à notre compte google avec la commande « firebase login ».

La prochaine étape consiste à lier notre projet local à notre projet Firebase avec quelques configurations au préalable, le but est que notre livrable généré avec « ng build » soit déployé.

Premièrement, il faut utiliser la commande « firebase init » afin de commencer la liaison, une série de questions vont être posé auxquelles il faut répondre. Pour la première question, il faut choisir la réponse « Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys » qui est l'utilisation la plus basique de Firebase Hosting. Ensuite, il nous demande si nous souhaitons utiliser un nouveau projet ou un projet existant. Il faut choisir un « projet existant », car nous l'avons créé au préalable. Une liste de choix apparaît, il faut maintenant choisir le projet concerné. Pour finir, il va nous demander plusieurs petites informations sur notre application. Tout d'abord, il souhaite savoir ce que nous souhaitons utiliser comme « public directory », dans notre cas nous utilisons « dist/nom-application » car c'est notre livrable. Puis, il nous interroge afin de savoir si notre application est une SPA, bien évidemment, il faut répondre « yes ». Pour la question concernant Github, c'est au choix de la personne déployant l'application. Pour finir, Firebase nous demande si nous souhaitons écraser notre « index.html » afin qu'il s'occupe d'en créer un lui-même, il faut répondre « no », car Angular s'en charge.

Une fois l'initialisation terminée, deux fichiers sont générés dans notre projet local « .firebaserc » et « firebase.json ». Sur le fichier « firebase.json » nous retrouverons toute la configuration pour Firebase.

Figure 43 : Fichier de configuration Firebase

```
{
  "hosting": {
    "public": "dist/on-air-football",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ]
  }
}
```

(Code tiré de mon application LiveBall)

Dans la figure ci-dessus, le premier point « public » représente l'emplacement de mon « build ». Ensuite, « ignore » permet juste de référencer les dossiers que je ne souhaite pas envoyer dans mon déploiement. Pour finir, « rewrite » est une configuration mise en place pour le bon fonctionnement d'une SPA.

Une fois toutes ces étapes complétées, il nous manque plus qu'à déployer. Pour cela, il suffit d'utiliser la commande « firebase deploy » qui saura quoi faire grâce au fichier de configuration. Lorsque nous obtenons le message de réussite « Deploy complete », Firebase nous fournit directement dans notre invite de commande les URL permettant d'accéder à notre application. L'URL « Project console » nous redirige sur notre projet Firebase tandis que « Hosting URL » nous permet d'accéder à notre application en ligne.

Figure 44 :Exemple de réussite de déploiement

```
=== Deploying to 'liveball-9298f'...

i  deploying hosting
i  hosting[liveball-9298f]: beginning deploy...
i  hosting[liveball-9298f]: found 10 files in dist/on-air-football
+  hosting[liveball-9298f]: file upload complete
i  hosting[liveball-9298f]: finalizing version...
+  hosting[liveball-9298f]: version finalized
i  hosting[liveball-9298f]: releasing new version...
+  hosting[liveball-9298f]: release complete

+  Deploy complete!

Project Console: https://console.firebase.google.com/project/liveball-9298f/overview
Hosting URL: https://liveball-9298f.web.app
```

(Mon CMD)

7. Conclusion

Dans le monde actuel, l'importance du développement web est immense et demande des technologies rigoureusement efficaces.

Après ce travail, je suis d'avis qu'Angular répond de façon optimale aux besoins des développeurs web actuels. C'est un Framework fortement complet avec énormément de concepts offrant aux concepteurs d'applications une grande flexibilité. Il demande une certaine rigueur pour la prise en main et surtout beaucoup de temps, mais une fois ceci fait une panoplie d'outils s'offre à nous, nous permettant de concevoir des solutions efficaces, modernes et stables.

J'ai personnellement beaucoup apprécié apprendre ce Framework, j'ai pu effectuer d'autres projets avec React.js et Vue.js mais je me suis senti bien plus à l'aise avec l'utilisations d'Angular. Le Framework est très bien structuré et surtout fortement bien documenté permettant ainsi un apprentissage dans les meilleures conditions possibles. Les différents concepts principaux d'Angular offrent une liberté que certains autres Framework n'étaient pas capables de m'offrir jusqu'ici, j'ai donc découvert un côté de la conception d'applications qui m'était encore inconnue.

Bien évidemment, le fait qu'Angular soit un Framework fortement complet et utilisant « TypeScript » fait de lui un Framework que j'ai pris énormément de temps à maîtriser. Il m'a demandé bien plus de rigueur et d'engagement que ses concurrents, mais cela en valait la peine.

Angular est pour moi une solution incontournable de la conception d'applications web et continuera de l'être pendant fort longtemps encore. Il a totalement sa place dans le haut du classement des solutions web. Étant open-source et ayant une communauté très active, je pense que son évolution va continuer d'être incroyablement croissante et qu'il va continuer d'être un premier choix pendant encore longtemps.

8. Bibliographie

ABOUELNASR, Mohamed, 2022. Angular 15 CLI does not create environments folder when creating an angular project via ng new. *Stack Overflow* [en ligne]. 24 novembre 2022. Disponible à l'adresse : <https://stackoverflow.com/q/74558182> [consulté le 1 avril 2023].

ALEXANDER, Will, [sans date]. Passez en SPA avec le routing. *OpenClassrooms* [en ligne]. Disponible à l'adresse : <https://openclassrooms.com/fr/courses/7471261-debutez-avec-angular/7549406-passez-en-spa-avec-le-routing> [consulté le 26 juin 2023].

ALLEN, Andrew, 2022. Answer to « Angular 15 CLI does not create environments folder when creating an angular project via ng new ». *Stack Overflow* [en ligne]. 24 novembre 2022. Disponible à l'adresse : <https://stackoverflow.com/a/74558518> [consulté le 1 avril 2023].

ANGULAR, [sans date]. Angular - @angular/router. [en ligne]. Disponible à l'adresse : <https://angular.io/api/router> [consulté le 26 juin 2023 a].

ANGULAR, [sans date]. Angular - CLI Overview and Command Reference. [en ligne]. Disponible à l'adresse : <https://angular.io/cli> [consulté le 1 avril 2023 b].

ANGULAR, [sans date]. Angular - Introduction to forms in Angular. [en ligne]. Disponible à l'adresse : <https://angular.io/guide/forms-overview> [consulté le 12 juillet 2023 c].

ANGULAR, [sans date]. Angular - Introduction to modules. [en ligne]. Disponible à l'adresse : <https://angular.io/guide/architecture-modules> [consulté le 21 juin 2023 d].

ANGULAR, [sans date]. Angular - NgForm. [en ligne]. Disponible à l'adresse : <https://angular.io/api/forms/NgForm> [consulté le 12 juillet 2023 e].

ANGULAR, [sans date]. Angular - NgModel. [en ligne]. Disponible à l'adresse : <https://angular.io/api/forms/NgModel> [consulté le 12 juillet 2023 f].

ANGULAR, [sans date]. Angular - Testing. [en ligne]. Disponible à l'adresse : <https://angular.io/guide/testing> [consulté le 12 juillet 2023 g].

ANGULAR, [sans date]. Angular - Transforming Data Using Pipes. [en ligne]. Disponible à l'adresse : <https://angular.io/guide/pipes> [consulté le 21 juin 2023 h].

ANGULAR, [sans date]. Angular - Understanding templates. [en ligne]. Disponible à l'adresse : <https://angular.io/guide/template-overview> [consulté le 4 juin 2023 i].

Angular - Get data from a server, [sans date] [en ligne]. Disponible à l'adresse : <https://angular.io/tutorial/tour-of-heroes/toh-pt6> [consulté le 3 mai 2023].

Angular - HttpClientModule, [sans date] [en ligne]. Disponible à l'adresse : <https://angular.io/api/common/http/HttpClientModule> [consulté le 7 juillet 2023].

Angular - Introduction to services and dependency injection, [sans date] [en ligne]. Disponible à l'adresse : <https://angular.io/guide/architecture-services> [consulté le 7 juillet 2023].

ANGULAR, 2022. Angular - NgModules. [en ligne]. 28 février 2022. Disponible à l'adresse : <https://angular.io/guide/ngmodules> [consulté le 3 avril 2023].

ANGULAR-TRAINING, [sans date]. Understanding the File Structure. [en ligne]. Disponible à l'adresse : https://angular-training-guide.rangle.io/bootstrapping/file_structure [consulté le 1 avril 2023].

DJORDJEVIC, Daniel, 2017. Le cycle de vie d'un composant Angular. [en ligne]. 6 septembre 2017. Disponible à l'adresse : <https://blogs.infinitiesquare.com/posts/web/blogs.infinitiesquare.com/posts/web/le-cycle-de-vie-d-un-composant-angular> [consulté le 5 avril 2023].

GUIDE-ANGULAR, [sans date]. Décorateurs. [en ligne]. Disponible à l'adresse : <https://guide-angular.wishtack.io/typescript/decorateurs> [consulté le 1 avril 2023].

Hébergement Node.js : quel est le meilleur hébergeur en 2023 ?, [sans date] *Presse-citron* [en ligne]. Disponible à l'adresse : <https://www.presse-citron.net/hebergeur/nodejs/> [consulté le 8 juillet 2023].

[HTTPS://WWW.FACEBOOK.COM/ANGULAR.UNIVERSITY](https://www.facebook.com/angular.university), 2016. Angular Router: A Complete Guide. *Angular University* [en ligne]. 9 octobre 2016. Disponible à l'adresse : <https://blog.angular-university.io/angular-2-router-nested-routes-and-nested-auxiliary-routes-build-a-menu-navigation-system/> [consulté le 3 mai 2023].

IONOS, 2022. Single Page Applications : définition, fonctionnement et utilité. *IONOS Digital Guide* [en ligne]. 28 février 2022. Disponible à l'adresse : <https://www.ionos.fr/digitalguide/sites-internet/creation-de-sites-internet/single-page-application/> [consulté le 1 avril 2023].

JAVATPOINT, [sans date]. Angular Template - javatpoint. [en ligne]. Disponible à l'adresse : <https://www.javatpoint.com/angular-template> [consulté le 4 juin 2023].

JESUISUNDEV, 2020. Comprendre Typescript en 5 minutes. *Je suis un dev* [en ligne]. 27 avril 2020. Disponible à l'adresse : <https://www.jesuisundev.com/comprendre-typescript-en-5-minutes/> [consulté le 1 avril 2023].

LA RÉDACTION, JDN, 2019. Framework ou infrastructure logicielle : définition et traduction. [en ligne]. 20 janvier 2019. Disponible à l'adresse : <https://www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1203355-framework/> [consulté le 1 avril 2023].

LOUISTITI, 2020. *Apprendre Node.js #9 - npm : le gestionnaire de paquets* [en ligne]. 28 juin 2020. Disponible à l'adresse : <https://www.youtube.com/watch?v=3n1OZzLO3uE> [consulté le 1 avril 2023].

Microsoft TypeScript, [sans date] [en ligne]. Disponible à l'adresse : <https://encyclopedia.pub/entry/30160> [consulté le 15 mai 2023].

Mise en Place du Routing, [sans date] [en ligne]. Disponible à l'adresse : <https://guide-angular.wishtack.io/angular/routing/mise-en-place-du-routing> [consulté le 29 juin 2023].

MOHIT, Joshi, [sans date]. Angular vs React vs Vue: Core Differences. *BrowserStack* [en ligne]. Disponible à l'adresse : <https://browserstack.wpengine.com/guide/angular-vs-react-vs-vue/> [consulté le 15 juin 2023].

MOZILLA, 2022. Application monopage (SPA) - Glossaire MDN : définitions des termes du Web | MDN. [en ligne]. 21 septembre 2022. Disponible à l'adresse : <https://developer.mozilla.org/fr/docs/Glossary/SPA> [consulté le 1 avril 2023].

MOZILLA, 2023. Web Components | MDN. [en ligne]. 12 mars 2023. Disponible à l'adresse : https://developer.mozilla.org/en-US/docs/Web/Web_Components [consulté le 4 avril 2023].

Pourquoi tester son application ?, [sans date] *OpenClassrooms* [en ligne]. Disponible à l'adresse : <https://openclassrooms.com/fr/courses/5641591-testez-votre-application-c/5641598-pourquoi-tester-son-application> [consulté le 12 juillet 2023 a].

Pourquoi tester son application ?, [sans date] *OpenClassrooms* [en ligne]. Disponible à l'adresse : <https://openclassrooms.com/fr/courses/5641591-testez-votre-application-c/5641598-pourquoi-tester-son-application> [consulté le 12 juillet 2023 b].

SEBASTIEN, 2022. TypeScript vs JavaScript : quelles différences ? - Codeur Blog. *Codeur.com* [en ligne]. 21 novembre 2022. Disponible à l'adresse : <https://www.codeur.com/blog/differences-typescript-vs-javascript/> [consulté le 15 mai 2023].

SIMON, Dieny, [sans date]. Développer Votre Première Application avec Angular (2022). *Udemy* [en ligne]. Disponible à l'adresse : <https://www.udemy.com/course/angular-developper-tutoriel-application-typescript/> [consulté le 1 avril 2023 a].

SIMON, Dieny, [sans date]. NodeJS tutoriel français : Développer facilement votre première API Rest (Express.js, Sequelize et MySQL inclus) - YouTube. [en ligne]. Disponible à l'adresse : https://www.youtube.com/playlist?list=PLhVogk7htzNiO_cbnPqa7fkVzzRhUqQLj [consulté le 1 avril 2023 b].

VAN DER FEER, Julien, 2020. Quelles sont les différentes catégories de framework ? *https://fiches-pratiques.chefdentreprise.com/* [en ligne]. 21 août 2020. Disponible à l'adresse : https://fiches-pratiques.chefdentreprise.com/Thematique/technologie-1109/FichePratique/Quelles-sont-les-differentes-categorie-de-framework--352263.htm#&utm_source=social_share&utm_medium=share_button&utm_campaign=share_button [consulté le 1 avril 2023].

WIKIPÉDIA, 2022a. Environnement de développement. *Wikipédia* [en ligne]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Environnement_de_d%C3%A9veloppement&oldid=198999051 [consulté le 1 avril 2023]. Page Version ID: 198999051

WIKIPÉDIA, 2022b. Plateforme client riche. *Wikipédia* [en ligne]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Plateforme_client_riche&oldid=199880551 [consulté le 1 avril 2023]. Page Version ID: 199880551

WIKIPÉDIA, 2023a. Angular. *Wikipédia* [en ligne]. Disponible à l'adresse : <https://fr.wikipedia.org/w/index.php?title=Angular&oldid=202061809> [consulté le 1 avril 2023]. Page Version ID: 202061809

WIKIPÉDIA, 2023b. Application web monopage. *Wikipédia* [en ligne]. Disponible à l'adresse :

https://fr.wikipedia.org/w/index.php?title=Application_web_monopage&oldid=200363944 [consulté le 1 avril 2023]. Page Version ID: 200363944

WIKIPÉDIA, 2023c. TypeScript. *Wikipédia* [en ligne]. Disponible à l'adresse : <https://fr.wikipedia.org/w/index.php?title=TypeScript&oldid=201696168> [consulté le 1 avril 2023]. Page Version ID: 201696168

Annexe 1 : Composant racine généré














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

You, il y a 6 jours | 1 author (You)
@Component({
  You, il y a 6 jours • initial commit
  selector: 'app-root',
  template: `
    <!--The content below is only a placeholder and can be replaced.-->
    <div style="text-align:center" class="content">
      <h1>
        Welcome to {{title}}!
      </h1>
      <span style="display: block">{{ title }} app is running!</span>
      Tour of Heroes</a></h2>
      </li>
      <li>
        <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Documentation</a></h2>
      </li>
      <li>
        <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
      </li>
    </ul>
    <router-outlet></router-outlet>
  `,
  styles: []
})
export class AppComponent {
  title = 'app-training';
}
```


Annexe 2 : fichier de configuration TypeScript

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "strictPropertyInitialization": false,
    "noImplicitOverride": true,
    "noPropertyAccessFromIndexSignature": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "sourceMap": true,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "ES2022",
    "module": "ES2022",
    "useDefineForClassFields": false,
    "lib": [
      "ES2022",
      "dom"
    ]
  },
  "angularCompilerOptions": {
    "enableI18nLegacyMessageIdFormat": false,
    "strictInjectionParameters": true,
    "strictInputAccessModifiers": true,
    "strictTemplates": true
  }
}
```

Annexe 3 : Pipe permettant d'afficher les équipes qualifiées à une compétition européenne

<div>   </div> <h3>Italy - Serie A</h3>										
Table				Next matches				Résumé		
Rank	Logo	Team	MP	W	D	L	GF	GA	GD	Points
1		Napoli	38	28	6	4	77	28	49	90
2		Lazio	38	22	8	8	60	30	30	74
3		Inter	38	23	3	12	71	42	29	72
4		AC Milan	38	20	10	8	64	43	21	70
5		Atalanta	38	19	7	12	66	48	18	64
6		AS Roma	38	18	9	11	50	38	12	63
7		Juventus	38	22	6	10	56	33	23	62
8		Fiorentina	38	15	11	12	53	43	10	56
9		Bologna	38	14	12	12	53	49	4	54
19		Cremonese	38	5	12	21	36	69	-33	27
20		Sampdoria	38	3	10	25	24	71	-47	19

Champions League (Group)
Champions League (qualification)
Europa league (Group)
Conference league (qualification)
Relegation (qualification)
Relegation

Annexe 4 : Service pour la récupération des matchs

```
import { Injectable } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Observable, catchError, of, tap } from "rxjs";
import { environment } from "../../environnements/environnement";

You, 2 mon
You, 1 second ago | 1 author (You)

@Injectable()
export class FixtureDataService {
  private apiUrlFixtures = environment.apiUrl + "fixtures";

  constructor(private http: HttpClient) {}

  getFixturesById(id: number): Observable<any[]> {
    const headers = new HttpHeaders({
      "x-rapidapi-key": environment.apiToken,
      "x-rapidapi-host": environment.apiHost,
    });

    const params = {
      id: id,
    };

    return this.http.get<any[]>(this.apiUrlFixtures, { headers, params }).pipe(
      tap((res) => console.log("OK match !")),
      catchError((err) => {
        console.log(err);
        return of([]);
      })
    );
  }

  getFixturesByDate(myDate: string): Observable<any[]> {
    const headers = new HttpHeaders({
      "x-rapidapi-key": environment.apiToken,
      "x-rapidapi-host": environment.apiHost,
    });

    const params = {
      date: myDate,
    };

    return this.http.get<any[]>(this.apiUrlFixtures, { headers, params }).pipe(
      tap((res) => console.log("OK match !")),
      catchError((err) => {
        console.log(err);
        return of([]);
      })
    );
  }
}
```

Annexe 5 : Création d'une instance et retour de cette dernière

```
this.fixture = new Match(  
  this.fixtureAny.fixture.id,  
  fDateMatch,  
  Hour,  
  this.fixtureAny.fixture.timezone,  
  this.fixtureAny.fixture.status.long,  
  this.fixtureAny.fixture.status.elapsed,  
  this.fixtureAny.teams.home.id,  
  this.fixtureAny.teams.away.id,  
  this.fixtureAny.teams.home.name,  
  this.fixtureAny.teams.away.name,  
  this.fixtureAny.teams.home.logo,  
  this.fixtureAny.teams.away.logo,  
  homeTeamCode,  
  awayTeamCode,  
  this.fixtureAny.league.id,  
  this.fixtureAny.goals.home,  
  this.fixtureAny.goals.away,  
  this.fixtureAny.fixture.venue.name,  
  this.fixtureAny.fixture.referee,  
  round  
);  
});  
  
console.log(this.events);  
}  
  
getFixture() {  
  return this.fixture;  
}
```