

APPRENDRE ANGULAR

Développez facilement votre première
application Angular avec Typescript



SIMON DIENY

Apprendre Angular

Développez facilement votre première application Angular avec TypeScript

Simon DIENY

Table des matières

<u>Remerciements</u>	4
<u>Avant propos</u>	5
<u>Chapitre 1 : Présentation de Angular</u>	10
<u>Chapitre 2 : ECMAScript 6</u>	9
<u>Chapitre 3 : Découvrir TypeScript</u>	
<u>Chapitre 4 : Les Web Components</u>	
<u>Chapitre 5 : Premiers pas avec Angular</u>	
<u>Chapitre 6 : Les Composants</u>	
<u>Chapitre 7 : Les Templates</u>	
<u>Chapitre 8 : Les Directives</u>	1
<u>Chapitre 9 : Les Pipes</u>	1
<u>Chapitre 10 : Les Routes</u>	1
<u>Chapitre 11 : Les Modules</u>	1
<u>Chapitre 12 : Les services et l'injection de dépendances</u>	1
<u>Chapitre 13 : Formulaires pilotés par le template</u>	1
<u>Chapitre 14 : La programmation réactive</u>	2
<u>Chapitre 15 : Effectuer des requêtes HTTP standards</u>	2

Chapitre 16 : Effectuer des traitements asynchrones avec RxJS **2**

Chapitre 17 : Authentification **2**

Chapitre 18 : Déployer votre application **262**

Remerciements

Si vous avez des suggestions ou des remarques sur le cours (manque de commentaires dans les extraits de code, présence de coquilles, etc), n'hésitez pas à m'en parler dans la classe virtuelle. On pourra en discuter, et si votre proposition permet d'améliorer le cours, c'est avec plaisir que je mettrai à jour le cours avec votre suggestion, et que je vous ajouterai cette la liste des remerciements. Vous serez alors co-auteur de ce cours, en quelque sorte !

Thibault : Pour avoir relu le cours en entier, et corrigé de trop nombreuses coquilles.

Jérôme : Pour avoir proposé des suggestions que se sont avérées pertinentes pour la suite.

Pierre-Yves, Jonathan et Brice : Pour avoir proposé des idées d'améliorations sur la toute première version de ce cours, alors que ce n'étais encore qu'un embryon dans le CourseLab.

Avant-propos

Bienvenue à tous, et à toutes, dans ce cours consacré au développement de votre première application avec Angular !

Ce cours s'adresse aux développeurs web qui souhaiteraient créer des applications web réactives : les développeurs novices, comme les développeurs plus expérimentés qui avaient l'habitude d'utiliser AngularJS.

Ce cours va vous permettre de prendre en main rapidement Angular. Vous verrez dans ce cours : *Pourquoi choisir Angular ? Comment mettre en place un environnement de développement ? Comment récupérer des données depuis un serveur distant ? Comment concevoir un site réactif avec plusieurs pages ?*

Sachez qu'il n'y a pas besoin de connaître AngularJS v.1 pour suivre ce cours, nous partons de zéro !

Cependant, il y a quelques prérequis nécessaires. Mais pas d'inquiétude, il s'agit de connaissances élémentaires sur lesquelles vous pouvez vous former, et il y a des cours très bien faits là-dessus sur Internet.

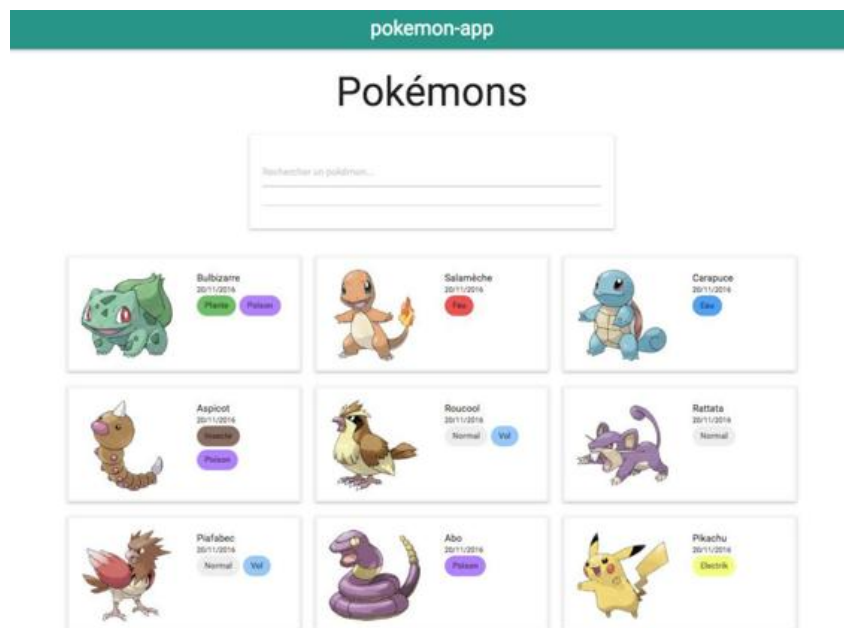
Je vous conseille donc de voir (ou de revoir) les cours suivants si vous en ressentez le besoin :

- Connaître le HTML et le CSS.
- Avoir déjà entendu parler de Javascript, car c'est le langage que nous allons utiliser tout le long de ce cours.
- Connaître un peu la programmation orientée objet (savoir ce qu'est une classe, une méthode, une propriété...)

Si vous êtes trop impatients, vous pouvez commencer à suivre le cours dès maintenant, car les premiers chapitres sont assez

théoriques, mais vous sentirez rapidement le besoin de vous mettre à niveau par la suite !

Pendant ce cours, je vous propose de développer une application pour gérer des Pokémons. La démonstration de l'application finale que vous réaliserez est disponible en ligne. Je vous donne l'adresse juste après, dans la section concernant les ressources du cours. Voici un aperçu :



1. Les versions utilisées

Dans ce cours, le terme AngularJS désigne la version 1.x d'Angular, et les versions supérieures du framework sont appelées simplement Angular. C'est l'appellation qui est recommandée par la documentation officielle : « *Angular is the name for the Angular of today and tomorrow. AngularJS is the name for all v1.x versions of Angular.* » J'utiliserai donc le terme Angular pour désigner les versions d'Angular supérieures ou égales à la version 2.

Je vous donne ces informations à titre informatif, il n'y a pas nécessité de s'inquiéter. Nous verrons comment installer et mettre à jour les outils dont nous aurons besoin pour développer sereinement avec Angular pendant le cours.

Est-ce que ce cours sera mis à jour ?

Alors en tant que livre papier ou PDF, ce n'est techniquement pas possible. Mais dans sa version numérique depuis La Bibliothèque d'Alexandrie, la réponse est : "Oui, bien sûr, le cours est toujours à jour" !

En effet, le cours est mis à jour à chaque fois qu'une nouvelle version d'Angular est disponible, c'est-à-dire environ tous les 6 mois. Vous recevrez les informations sur les dernières nouveautés d'Angular par email.

Sachez que l'équipe en charge du développement d'Angular assure le support de chaque version pendant un an et demi, à partir de sa date de sortie. Cela vous laisse donc un peu de temps devant vous.

2. Comment est structuré le cours ?

Ce cours est structuré en trois parties distinctes :

Partie 1, Une vue d'ensemble d'Angular : Cette section théorique vous permet de savoir où vous mettez les pieds, et comment réaliser un magnifique « Hello, World ! » avec Angular.

Partie 2, Acquérir les bases d'Angular : Nous verrons comment maîtriser les éléments de base d'une application Angular avec les composants, les templates, la gestion des routes...

Partie 3, Aller plus loin avec Angular : Nous lèverons le voile sur les formulaires, les requêtes HTTP, l'authentification, le déploiement...

3. Accéder aux ressources du cours

Les ressources nécessaires pour suivre ce cours sont disponibles à l'adresse suivante :

www.alexandria-library.co/ressources-angular

Je vous recommande de vous y rendre tout de suite, si vous avez votre ordinateur ou votre téléphone à portée de main. Cela vous permettra de vous familiariser tout de suite avec cette page, sur laquelle vous allez passer beaucoup de temps. Vous y trouverez :

- **Le code source de l'application que nous réaliserons** : vous pouvez vous en servir comme correction, n'hésitez pas à vous y référer en cas de doute !
- **Les extraits de code trop long** : Parfois, il y a beaucoup de code à recopier manuellement, ce qui n'a pas vraiment d'intérêt pédagogique. À la place, je vous propose de copier-coller le code en question directement depuis la page des ressources du cours. Par contre, je ne donne pas le code correspondant de la solution, il s'agit du socle de départ, pour vous faire économiser du temps. Ensuite, c'est à vous de bosser.
- **Les exercices proposés tout au long de la formation**. Les corrections sont soit disponibles directement dans le cours, soit avec les ressources de ce cours.

Attention : Les extraits de code sont disponibles correspondant à l'état du fichier lorsque vous le rencontrerez pour la première fois dans le chapitre, sans les modifications ultérieures qui peuvent être apportées plus loin. Il s'agit de vous faire économiser le temps de recopie d'un fichier long, ensuite c'est à vous de travailler dessus en suivant le cours !

Partie 1

Découvrir Angular

Chapitre 1 : Présentation de Angular

De quoi parle-t-on exactement ?



Alors comme ça vous souhaitez vous former au développement d'applications Web avec la nouvelle version d'Angular ? Vous aussi vous rêvez de construire des sites dynamiques, qui réagissent immédiatement aux moindres interactions de vos utilisateurs, avec une performance optimale ? Eh, ça tombe bien, vous êtes au bon endroit !

Nous vivons une époque excitante pour le développement Web avec JavaScript. Il y a une multitude de nouveaux Frameworks disponibles, et encore une autre multitude qui éclot jour après jour. Nous allons voir pourquoi vous devez faire le pari de vous lancer avec Angular, et ce que vous allez pouvoir faire avec ce petit bijou, sorti tout droit de la tête des ingénieurs de Google.

Cette nouvelle version d'Angular est une réécriture complète de la première version d'Angular, nommée AngularJS. C'est donc une bonne nouvelle pour ceux qui ne connaîtraient pas cette version d'Angular, ou qui en auraient juste entendu parler : **pas besoin de connaître AngularJS, vous pouvez vous lancer dans l'apprentissage d'Angular dès maintenant !**

Pour rappel et pour être tout à fait clair : Angular désigne le « nouvel » Angular (version 2 et supérieure), et AngularJS désigne la version 1 de cet outil. Ce sont ces appellations que j'utiliserai dans le cours.

1. Introduction

Alors commençons par le commencement, qu'est-ce que c'est Angular, au fait ? Et bien c'est un Framework.

Et c'est quoi, un frame... machin ?

Un Framework est un mot Anglais qui signifie « Cadre de travail ». En gros c'est un outil qui permet aux développeurs (c'est-à-dire vous) de travailler de manière plus efficace et de façon plus organisée. Vous avez sûrement remarqué que vous avez souvent besoin de faire les mêmes choses dans vos applications : valider des formulaires, gérer la navigation, lever des erreurs... Souvent les développeurs récupèrent des fonctions qu'ils ont développées pour un projet, puis les réutilisent dans d'autres projets. Et bien dans ce cas-là, on peut dire que vous avez développé une sorte de mini-Framework personnel !

L'avantage d'un Framework professionnel, est qu'il permet à plusieurs développeurs de travailler sur le même projet, sans se perdre dans l'organisation du code source. En effet, lorsque vous développez des fonctions « maison », vous êtes le seul à les connaître, et si un jour vous devez travailler avec un autre développeur, il devra d'abord prendre connaissance de toutes ces fonctions. En revanche, un développeur qui rejoint un projet qui utilise un Framework, connaît déjà les conventions et les outils à sa disposition pour pouvoir se mettre au travail.

Oui d'accord, c'est sympa d'avoir un cadre de travail commun, mais pour travailler sur quoi exactement ?

Effectivement, quels genres d'applications peuvent être développés avec Angular ? Et bien Angular permet de développer des applications web, de manière robuste et efficace. Nous allons voir la différence entre une application web, et un site web, car cette

distinction est très importante pour bien comprendre dans quoi vous mettez les pieds.

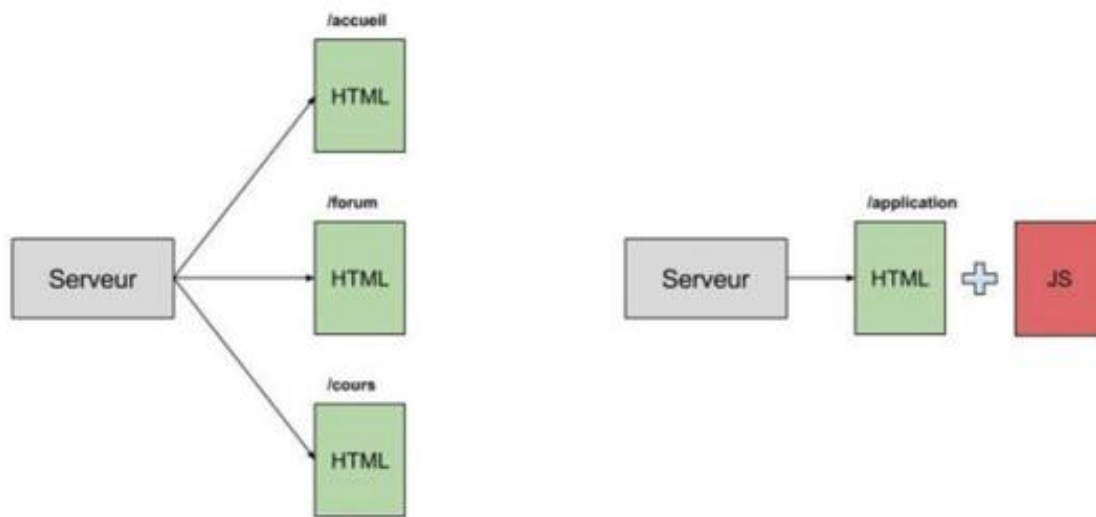
1.1. Site Web versus Application Web

La tendance actuelle en développement web est de vouloir séparer complètement :

- La partie client du site : c'est-à-dire les fichiers HTML, CSS et JavaScript, qui sont interprétés par le navigateur.
- La partie serveur du site : les fichiers PHP, si vous développez votre site en PHP, les fichiers Java si vous utilisez Java... qui sont interprétés côté serveur. C'est dans cette partie que l'on fait des requêtes aux bases de données, par exemple.

Un « site web » au sens traditionnel du terme, est donc une application serveur qui envoie des pages HTML dans le navigateur du client à chaque fois que celui-ci le demande. Quand l'utilisateur navigue sur votre site et change de page, il faut faire une requête au serveur. Quand l'utilisateur remplit et soumet un formulaire, il faut faire une requête au serveur... bref, tout cela est long, coûteux, et pourrait être fait plus rapidement en JavaScript.

Une « application web » est une simple page HTML, qui contient suffisamment de JavaScript pour pouvoir fonctionner en autonomie une fois que le serveur l'a envoyé au client. Je vous ai fait un petit schéma d'explication :



L'architecture d'un site web et d'une application web.

À votre avis, quel est le schéma représentant un site web, et celui représentant une application Web ?

Le schéma de gauche représente un site web : à chaque fois que l'utilisateur demande une page, le serveur s'occupe de la renvoyer : /accueil, /forum, etc... Dans le cas d'une application web, le serveur ne renvoie qu'une page pour l'ensemble du site, puis le JavaScript prend le relais pour gérer la navigation, en affichant ou masquant les éléments HTML nécessaires, pour donner l'impression à l'internaute qu'il navigue sur un site traditionnel !

L'avantage de développer un site de cette façon, c'est qu'il est incroyablement plus réactif. Imaginez, vous remplacez le délai d'une requête au serveur par un traitement JavaScript ! De plus, comme vous n'avez pas à recharger toute la page lors de la navigation, vous pouvez permettre à l'utilisateur de naviguer sur votre site tout en discutant avec ses amis par exemple ! (Comme la version web de Facebook).

Vous avez remarqué que l'ensemble d'une application web est contenu sur une seule page ? Eh bien, on appelle ce genre d'architecture une architecture SPA, ce qui signifie simplement Single Page Application.

Même si Angular ne vous plait pas au final (ce dont je doute fort !), vous serez sûr d'avoir appris beaucoup de choses sur l'écosystème bourgeonnant du développement d'applications web. La première partie de ce cours contient essentiellement des chapitres théoriques qui permettront de poser le décor avant d'attaquer le vif du sujet. Nous terminerons la première partie de ce cours avec le fameux « Hello, World ! » et un questionnaire pour vérifier que vous n'avez pas fait semblant de suivre ce cours.

Le « Hello, World ! » est un grand classique en programmation. Dans chaque langage que vous aborderez, la première chose à faire est de développer une petite application qui affiche un message de bienvenue à l'utilisateur pour vérifier que tout marche bien ! On appelle cette petite application un « Hello, World ! ».

2. Différences entre Angular et AngularJS

C'était quoi, AngularJS ?

AngularJS était très populaire et a été utilisé pour développer des applications clients complexes, exactement comme son grand-frère Angular. Il permettait de réaliser de grosses applications et de les tester avec efficacité. Il a été développé dans les locaux de Google en 2009 et était utilisé pour quelques-unes de ces applications en interne (je ne pourrai pas vous dire lesquelles par contre).

Pour les développeurs d'AngularJS, je vous ai préparé une petite liste ci-dessous des changements majeurs entre la version un et la deux. Voici résumé en six points les changements qui me paraissent les plus importants :

1. **Les contrôleurs** : L'architecture traditionnelle MVC est remplacée par une architecture réactive à base de composants web. L'architecture MVC est une architecture classique que l'on retrouve dans beaucoup de Framework (Symfony, Django, Spring) permettant de découper le Modèle, la Vue et le Contrôleur.
2. **Les directives** : La définition existante de l'objet Directive est retirée, et remplacée par trois nouveaux types de directives à la place : les composants, les directives d'attributs et les directives structurelles.
3. **Le \$scope** : Les scopes et l'héritage de scope sont simplifiés et la nécessité d'injecter des \$scopes est retirée.
4. **Les modules** : Les modules AngularJS sont remplacés par les modules natifs d'ES6.
5. **jQuery** : Cette version plus légère de jQuery était utilisée dans AngularJS. Elle est retirée dans Angular, principalement pour des raisons de performance.

6. **Le two-way data-binding** : Pour les même raisons de performances, cette fonctionnalité n'est pas disponible de base. Cependant, et il est toujours possible d'implémenter ce mécanisme avec Angular.

Pourquoi Angular ?

L'équipe de Google qui travaille sur AngularJS a officiellement annoncé Angular à la conférence européenne Ng-Conf en Octobre 2014 (Une conférence consacrée spécialement à Angular). Ils ont annoncé que cette nouvelle version ne serait pas une mise à jour, mais plutôt une réécriture de l'ensemble du Framework, ce qui causera des changements de ruptures important. D'ailleurs, il a été annoncé que AngularJS ne serait plus maintenu à partir de 2018. On dirait qu'ils veulent faire passer un message, vous ne trouvez pas ?

Les motivations pour développer un nouveau Framework, tout en sachant que cette version ne serait pas rétro-compatible, étaient les suivantes :

1. **Les standards du web** ont évolué, en particulier l'apparition des Web Components (nous y reviendrons) qui ont été développés après la sortie d'AngularJS, et qui fournissent de meilleures solutions de manière native que celles qui existent actuellement dans les implémentations spécifiques d'AngularJS. Angular a été l'un des premiers Frameworks conçu pour intégrer sérieusement avec les Web Components.
2. **JavaScript ES6** – la plupart des standards d'ES6 ont été finalisés et le nouveau standard a été adopté mi-2015 (Nous y reviendrons également). ES6 fournit des fonctionnalités qui peuvent remplacer les implémentations existantes d'AngularJS et qui améliorent leurs performances. Pourquoi s'en priver ?
3. **Performance** – AngularJS peut être amélioré avec de nouvelles approches et les fonctionnalités d'ES6. Différents

modules du noyau d'Angular ont été également retirés, notamment jqLite (une version de JQuery allégée pour Angular) ce qui résulte en de meilleures performances. Ces nouvelles performances font d'Angular un outil parfaitement approprié pour développer des applications web mobiles.

3. La philosophie d'Angular

Angular est un Framework orienté composants. Lors du développement de nos applications, nous allons coder une multitude de petits composants, qui une fois assemblés tous ensemble, constitueront une application à part entière. Un composant est l'assemblage d'un morceau de code HTML, et d'une classe JavaScript dédiée à une tâche particulière.

Hé mais attends, ... depuis quand il y a des classes en JavaScript ?

Oui, je sais, les classes n'existent pas en JavaScript... mais je vous dis qu'on va quand même en utiliser dans nos développements, soyez patient !

Ce qu'il faut bien comprendre, c'est que ces composants reposent sur le standard des Web Components, que nous verrons dans un chapitre dédié. Ce standard n'est pas encore supporté par tous les navigateurs, mais ça pourrait le devenir un jour. Il a été pensé pour découper votre page web en fonction de leur rôle : barre de navigation, boîte de dialogue pour tchatter, contenu principal d'une page... Un composant est censé être une partie qui fonctionne de manière autonome dans une application.

Angular n'est pas le seul à s'intéresser à ce nouveau standard, mais c'est le premier (enfin, je n'en connais aucun autre avant lui) à considérer sérieusement l'intégration des Web Components.

4. Conclusion

Il est important de noter que Angular est une technologie récente, et que par conséquent elle évolue rapidement. C'est pourquoi savoir s'adapter au fur et à mesure et ne pas baisser les bras est important. Si à un moment ou à un autre vous bloquez sur le cours, ne paniquez pas. Pensez à prendre une grande respiration, et à reprendre lentement la partie qui vous bloque, depuis le début.

Pour les curieux, sachez que la documentation officielle d'Angular se trouve sur le site angular.io. Par contre, cette documentation n'est disponible qu'en anglais. Qui a dit que l'anglais était partout en informatique ?

En résumé

- Les sites web deviennent de plus en plus de véritables applications, et une utilisation intensive du langage JavaScript devient nécessaire.
- Angular est un Framework orienté composant, votre application entière est un assemblage de composants.
- Les quatre éléments à la base de toute application sont : le module, le composant, le Template et les annotations.
- Nous utiliserons SystemJS pour charger les modules de nos applications.
- Angular est conçu pour le web de demain et intègre déjà la norme ECMAScript6 (ES6) et les Web Components.
- Enfin, retenez que tout cet écosystème est bourgeonnant et change très vite. Soyez persévérant lors de votre apprentissage et ne vous découragez pas, et vous prendrez vite plaisir à utiliser Angular !

Chapitre 2 : ECMAScript 6

Le nouveau visage de JavaScript...

Si vous n'avez jamais entendu parler de ECMAScript 6 (ou ES6), c'est que vous n'avez pas encore percé tous les mystères de JavaScript ! Vous avez sans doute remarqué que JavaScript est un langage un peu à part : un système d'héritage dit « prototypal », des fonctions « anonymes », ... Bref, le besoin d'une nouvelle standardisation s'est fait sentir pour fournir à JavaScript les moyens de développer des applications web robustes.

1. C'est quoi, ECMAScript 6 ?

ECMAScript 6 est le nom de la dernière version standardisé de JavaScript. Ce standard a été approuvé par l'organisme de normalisation en Juin 2015 : cela signifie que ce standard va être supporté de plus en plus par les navigateurs dans les temps à venir. ECMAScript 6 a été annoncé pour la première fois en 2008, et bientôt il deviendra inévitable (Il est important de noter toutes les versions futures de ECMAScript ne prendront pas autant de temps).

Mais ça fait quelque temps que je développe en JavaScript, je n'ai jamais entendu parler de tout ça !

En fait, sans le savoir, vous deviez surement développer en ES5 car c'est le standard le plus courant utilisé depuis quelques années.

Pour votre information, ECMAScript6, ECMAScript 2015 et ES6 désignent la même chose. Nous utiliserons comme petit nom ES6, car c'est son surnom le plus populaire.

Même si toutes les nouveautés d'ES6 ne fonctionnent pas encore dans certains navigateurs, beaucoup de développeurs ont commencé à développer avec ES6 (Pourquoi ne pas prendre un peu d'avance ?) et utilisent un transpileur pour convertir leur code ES6 en du code ES5. Ainsi leur code peut être interprété par tous les navigateurs.

Un transpileur ?

Le transpileur est un outil qui permet de publier son code pour les navigateurs qui ne supportent pas encore l'ES6 : le rôle du transpileur est de traduire le code ES6 en code ES5. Comme certaines fonctionnalités d'ES6 ne sont pas disponibles dans ES5, leur comportement est simulé.

Dans ce chapitre nous n'allons pas voir comment utiliser un transpileur : nous allons juste nous consacrer à la découverte

théorique d'ES6, et vous verrez qu'il y a déjà pas mal de choses à voir. Mais dès le prochain chapitre, nous utiliserons un transpileur ! Patience...

La bonne nouvelle c'est que nous allons coder en ES6, et être ainsi à la pointe de la technologie !

Sachez qu'ECMAScript 6 est une spécification standardisée qui ne concerne pas seulement JavaScript, mais également le langage Swift d'Apple, entre autres ! JavaScript est une des implémentations de la spécification standardisée ECMAScript 6.

2. Le nouveau monde d'ES6

Commençons tout de suite avec une des nouveautés les plus intéressantes d'ES6 : il est désormais possible d'utiliser des classes en Javascript ! Pour créer une simple classe *Vehicule* avec ES5, nous aurions dû faire comme ceci :

```
function Vehicle(color, drivingWheel) {  
  this.color = color;  
  this.drivingWheel = drivingWheel;  
  this.isEngineStart = false;  
}  
Vehicle.prototype.start = function start(){  
  this.isEngineStart = true;  
}  
Vehicle.prototype.stop = function stop(){  
  this.isEngineStart = false;  
}
```

Le code ci-dessus était le moyen détourné utilisé pour créer un objet, en utilisant la mécanique des prototypes, propre à JavaScript. Mais ES6 introduit une nouvelle syntaxe : *class*. C'est le même mot clé que dans d'autres langages, mais sachez que c'est toujours de l'héritage par prototype qui tourne derrière, mais vous n'avez plus à vous en soucier.

```
class Vehicle {  
  constructor(color, drivingWheel) {  
    this.color = color;  
    this.drivingWheel = drivingWheel;  
    this.isEngineStart = false;  
  }  
  start() {  
    this.isEngineStart = true;  
  }  
  stop() {  
    this.isEngineStart = false;  
  }  
}
```

Voilà une classe JavaScript, bien différente de ce que l'on avait précédemment. Si vous avez bien remarqué à la ligne 2, on a même droit à un constructeur !
C'est merveilleux !

2.1. L'héritage

En plus d'avoir ajouté les classes en JavaScript, ES6 continue avec l'héritage. Plus besoin de l'héritage prototypal.

Avant, il fallait appeler la méthode *call* pour hériter du constructeur. Par exemple, pour développer une classe *Voiture* et *Moto*, héritant de la classe *Vehicule*, on écrivait quelque chose comme ça :

```
function Vehicle (color, drivingWheel) {  
  this.color = color;  
  this.drivingWheel = drivingWheel;  
  this.isEngineStart = false;  
}  
Vehicle.prototype.start = function start() {  
  this.isEngineStart = true;  
}  
Vehicle.prototype.stop = function stop() {  
  this.isEngineStart = false;  
};  
  
// Une voiture est un véhicule.  
function Car(color, drivingWheel, seatings) {  
  Vehicle.call(this, color, drivingWheel);  
  this.seatings = seatings;  
}  
Vehicle.prototype = Object.create(Vehicle.prototype);  
  
// Une moto est un véhicule également.  
function Motorbike (color, drivingWheel, unleash) {  
  Vehicle.call(this, color, drivingWheel);  
  this.unleash = unleash;  
}  
Motorbike.prototype = Object.create(Vehicle.prototype);
```

Maintenant on peut utiliser le mot clé *extends* en JavaScript (non, non ce n'est pas une blague), et le mot-clé *super*, pour rattacher le tout à la « superclasse », ou « classe-mère » si vous préférez.

```
class Vehicle {  
  constructor(color, drivingWheel, isEngineStart = false) {  
    this.color = color;  
    this.drivingWheel = drivingWheel;  
    this.isEngineStart = isEngineStart;  
  }  
  start() {  
    this.isEngineStart = true;  
  }  
}
```



```
stop() {  
  this.isEngineStart = false;  
}  
}  
class Car extends Vehicle {  
  constructor(color, drivingWheel, isEngineStart = false, seatings) {  
    super(color, drivingWheel, isEngineStart);  
    this.seatings = seatings;  
  }  
}  
class Moto extends Vehicle {  
  constructor(color, drivingWheel, isEngineStart = false, unleash) {  
    super(color, drivingWheel, isEngineStart);  
    this.unleash = unleash;  
  }  
}
```

Le code est quand même plus clair et concis avec cette nouvelle syntaxe.

2.2. Les paramètres par défaut

En JavaScript, on ne peut pas restreindre le nombre d'arguments attendus par une fonction, ni définir des paramètres comme facultatifs. Voici l'implémentation traditionnelle de la fonction *Somme* en JavaScript (*Sum* en Anglais), qui prend un nombre quelconque d'arguments en paramètre, les additionne, puis retourne le résultat :

```
function sum(){  
  var result = 0;  
  for (var i = 0; i < arguments.length; i++) {  
    result += arguments[i];  
  }  
  return result;  
}
```

Comme vous pouvez le constater, il n'y a pas d'arguments dans la signature de la fonction, mais le mot-clé *arguments* permet de récupérer le tableau des paramètres passés à la fonction et ainsi de le traiter dans le code de la fonction. Par contre, ce n'est pas très pratique si l'on veut définir un nombre déterminé de paramètres.

Et pour définir des valeurs par défaut ? Les développeurs avaient l'habitude de bricoler pour faire comme si les variables par défaut étaient supportées :

```
function someFunction (defaultValue) {  
  defaultValue = defaultValue || undefined;  
  return defaultValue;  
}
```

Dans le code ci-dessus, si aucun paramètre n'est passé en paramètre, la fonction renverra *undefined*, sinon elle renverra la valeur passée en paramètre.

Imaginons une fonction qui multiplie deux nombres passés en paramètres. Mais le deuxième paramètre est facultatif, et vaut un par défaut :

```
function multiply (a, b) {  
  // b est facultatif  
  var b = typeof b !== 'undefined' ? b : 1;  
  return a*b;  
}
```

```
}  
multiply (2, 5); // 10  
multiply (1, 5); // 5  
multiply (5); // 5
```

À la ligne 3, nous utilisons un opérateur ternaire pour attribuer la valeur 1 à la variable b si aucun nombre n'a été passé en deuxième paramètre. Ce code semble très compliqué pour réaliser quelque chose de très simple. Heureusement, ES6 nous permet d'utiliser une syntaxe plus élégante :

```
function multiply (a, b = 1) {  
  return a*b;  
}  
multiply (5); // 5
```

Avec ES6, il suffit de définir une valeur par défaut dans la signature même de la fonction.

C'est quand même beaucoup plus pratique, non ?

2.3. Les nouveaux mots clés : « **let** » & « **const** »

Le mot-clé *let* permet de déclarer une variable locale dans le contexte où elle a été assignée (Un *contexte* est le terme français pour désigner un *scope* en Anglais).

Par exemple, les instructions que vous écrivez dans le corps d'une fonction ou à l'extérieur n'ont pas le même contexte. Normalement une instruction *if* n'a pas de contexte en soi, mais maintenant si, avec le mot clé *let*. Cela peut être utile pour effectuer beaucoup d'opérations sur une variable, sans polluer d'autres contextes avec des variables qui ne sont pas nécessaires :

```
var x = 1;
if(x < 10) {
  let v = 1;
  v = v + 21;
  console.log(v);
}
// v n'est pas définie, car déclarée avec 'let' et non 'var'.
console.log(v);
```

Et cela fonctionne pour les boucles également !

```
for (let i = 0; i < 10; i++) {
  console.log(i); // 0, 1, 2, 3, 4 ... 9
}
// i n'est pas défini hors du contexte de la boucle
console.log(i);
```

En général, garder un contexte global propre est vivement conseillé et c'est pourquoi ce mot clé est vraiment le bienvenu ! Sachez que *let* a été pensé pour remplacer *var*, alors nous n'allons pas nous en priver dans ce cours.

Le mot clé *const* quant à lui, permet de déclarer ... des constantes ! Voyons comment cela fonctionne :

```
const PI = 3.141592;
```

Une déclaration de constante ne peut se faire qu'une fois, une fois définie, vous ne pouvez plus changer sa valeur :

```
PI = 3.14 // Erreur : la valeur de PI ne peut plus être modifiée
```

C'est bien pratique si vous avez besoin de définir des valeurs une bonne fois pour toutes dans votre application.

Cependant, le comportement est un peu différent pour une constante de tableau ou d'objet. Vous ne pouvez pas modifier la *référence* vers le tableau ou l'objet, mais vous pouvez continuer à modifier les *valeurs* du tableau, ou les propriétés de l'objet :

```
const MATHEMATICAL_CONSTANTS = {PI: 3.141592, e: 2.718281};  
MATHEMATICAL_CONSTANTS.PI = 3.142; // Aucun problème.
```

Une dernière chose. En JavaScript, comme dans beaucoup d'autres langages, il existe des mots-clés réservés. Ce sont des mots que vous ne devez pas utiliser comme noms de variables, de fonctions ou de méthodes, car JavaScript a une utilité spéciale pour eux. Voici quelques exemples de mots-clés : *if*, *else*, *new*, *var*, *for* ... mais également *class* ou *super* !

2.4. Un raccourci pour créer des objets

ES6 apporte un raccourci sympathique pour créer des objets. Observez le code suivant :

```
function createCar() {  
  let color = 'green';  
  let wheel = 4;  
  return { color, wheel };  
}
```

Cela vous semble étrange ? Vous avez l'impression qu'il manque quelque chose à la ligne 4 ? Et bien vous avez raison, normalement nous aurions dû écrire :

```
function createCar() {  
  let color = 'green';  
  let wheel = 4;  
  return { color: color, wheel: wheel };  
}
```

Comme vous pouvez le constater, à la ligne 4, si la propriété de l'objet a le même nom que la variable utilisée comme valeur pour l'attribut, nous pouvons utiliser le raccourci d'ES6 comme ci-dessus.

2.5. Les promesses

Les promesses, ou « promises » en anglais, étaient déjà présentes dans AngularJS. Pour ceux qui ne voient pas de quoi on parle, nous allons voir ça tout de suite.

L'objectif des promesses est de simplifier la programmation asynchrone. En général, on utilise les fonctions de callbacks (des fonctions anonymes qui sont appelées à certains moments dans votre application), mais les Promesses sont plus pratiques que les Callbacks. Voici par exemple un code avec des callbacks, pour afficher la liste d'amis d'un utilisateur quelconque :

```
getUser(userId, function(user) {  
  getFriendsList(user, function(friends) {  
    showFriends(friends);  
  });  
});
```

L'exemple ci-dessus permet de récupérer un objet **user** à partir de son identifiant, puis de récupérer la liste de ses amis, et enfin d'afficher cette liste. On constate que ce code est très verbeux, et qu'il sera vite compliqué de le maintenir. Les promesses nous proposent un code plus efficace et plus élégant :

```
getUser(userId)  
  .then(function(user) {  
    getFriendsList(user);  
  })  
  .then(function(friends) {  
    showFriends(friends);  
  });
```

Il n'y a même pas besoin d'explications j'espère : le code est assez clair, il parle de lui-même !

Heu, oui peut-être, mais c'est quoi cette méthode then ?

Vous avez raison, je ne vous ai pas encore tout dit. En fait, lorsque vous créez une promesse (avec la classe `Promise`), vous lui associez implicitement une méthode `then`, et cette méthode prend deux arguments : une fonction en cas de succès et une fonction en cas d'erreur. Ainsi, lorsque la promesse est réalisée, c'est la fonction de succès qui est appelée, et en cas d'erreur, c'est la fonction d'erreur qui est invoquée.

Voici un exemple d'une promesse qui récupère un utilisateur depuis un serveur distant, à partir de son identifiant :

```
let getUser = function(userId) {  
  return new Promise(function(resolve, reject) {  
    // Appel asynchrone au serveur...  
    // pour récupérer les infos d'un utilisateur...  
    // À partir de la réponse du serveur,  
    // j'extrais les données de l'utilisateur :  
    let user = response.data.user;  
    if(response.status === 200) {  
      resolve(user);  
    } else {  
      reject('Cet utilisateur n'existe pas !');  
    }  
  })  
}
```

Et voilà, vous venez de créer une promesse ! Vous pouvez ensuite l'utiliser avec la méthode *then* dans votre application :

```
getUser(userId)  
  
  .then(function (user) {  
  
    console.log(user); // en cas de succès  
  
  }, function (error) {  
  
    console.log(error); // en cas d'erreur  
  
  });
```

Voilà, vous en savez déjà un peu plus sur les Promesses. Je voudrais justement vous montrer autre chose qui pourrait vous intéresser, les *arrow functions*, qui vous permettent de simplifier

l'écriture des fonctions anonymes : cela se combine parfaitement avec l'utilisation des promesses.

2.6. Les fonctions fléchées, ou « Arrow Functions »

Les *arrows functions* (ou fonctions 'fléchées' en français) sont une nouveauté d'ES6. Le premier cas d'utilisation des *arrows functions* est avec *this*. L'utilisation de *this* peut être compliquée, surtout dans le contexte de fonctions à l'intérieur d'autres fonctions. Prenons l'exemple ci-dessous :

```
class Person {
  constructor(firstName, email, button) {
    this.firstName = firstName;
    this.email = email;
    button.onclick = function() {
      // ce 'this' fait référence au bouton,
      // et non à une instance de Personne.
      sendEmail(this.email);
    }
  }
}
```

Comme vous le voyez en commentaire, le *this* de la ligne 8 ne fait pas référence à l'instance de *Person* mais au bouton sur lequel l'utilisateur a cliqué. Or, c'est le contraire que nous souhaitons, nous voulons avoir accès au mail de la personne, pas au bouton ! Les développeurs JavaScript ont donc pensé à utiliser une variable intermédiaire à l'extérieur du contexte de la fonction, souvent nommé *that*, comme ceci :

```
class Person {
  constructor(firstName, email, button) {
    this.firstName = firstName;
    this.email = email;
    // 'this' fait référence ici à l'instance de Personne
    var that = this;

    button.onclick = function() {
      // 'that' fait référence à la même instance de Personne
      sendEmail(that.email);
    }
  }
}
```

Cette fois-ci, nous obtenons le comportement souhaité, mais avouez que ce n'est pas très élégant.

C'est là que les *arrows functions* entrent en scène. Nous pouvons écrire la fonction *onclick* comme ceci :

```
button.onclick = () => { sendEmail(this.email); }
```

Les *arrow functions* ne sont donc pas tout à fait des fonctions classiques, parce qu'elles ne définissent pas un nouveau contexte comme les fonctions traditionnelles. Nous les utiliserons beaucoup dans le cadre de la programmation asynchrone qui nécessite beaucoup de fonctions anonymes.

C'est également pratique pour les fonctions qui tiennent sur une seule ligne :

```
someArray = [1, 2, 3, 4, 5];  
let doubleArray = someArray.map((n) => n*2);  
console.log(doubleArray); // [2, 4, 6, 8, 10]
```

La fonction `map` de JavaScript permet d'appliquer un traitement à chaque élément d'un tableau grâce à une fonction passée en paramètre.

Pour reprendre l'exemple avec les promesses, on peut écrire ça :

```
// Un traitement asynchrone en quelques lignes !  
getUser(userId)  
  .then(user => getFriendsList(user))  
  .then(friends => showFriends(friends));
```

2.7. Les ‘Template Strings’

Avant de terminer avec les nouveautés d'ES6, je voulais vous expliquer comment utiliser les *templates strings*. C'est une petite amélioration d'ES6 bien sympathique !

Jusqu'à maintenant, concaténer des chaînes de caractères était pénible en JavaScript, il fallait ajouter des symboles « + » les uns à la suite des autres :

```
let someText = "duTexte";
let someOtherText = "unAutreTexte";
let embarrassingString = someText;

embarrassingString += " blabla";
embarrassingString += someText;
embarrassingString += "blabla";
embarrassingString += someOtherText;

return embarrassingString;
```

Comme vous pouvez le constater, c'était assez pénible, et cela augmente les erreurs d'inattention.

Mais avec ES6, on peut utiliser des *templates strings*, qui commencent et se terminent par un *backtick* (```). Il s'agit d'un symbole particulier qui permet d'écrire des chaînes de caractères sur plusieurs lignes.

```
// On écrit des strings sur plusieurs ligne grâce au backtick !
let severalLinesString = `bla
blablablablablablab
balblablablabla
b
ablablabbbl`;

// .. mais pas avec des guillemets !
// Ce code va lever une erreur !
let severalLinesStringWithError = "bla
blba
blbla
blabla"
```

On peut même insérer des variables dans la chaîne de caractères avec `${}`, comme ceci :

```
return `${this.name} a pour email : ${this.email}`;
```

Bref, il est bien pratique ce *backtick*, pas vrai ?

3. Conclusion

Nous avons vu dans ce chapitre plusieurs changements majeurs apportés par ES6 à JavaScript, et qui nous serviront avec Angular. Sachez que ES6 est rétro-compatible, donc vous pouvez toujours développer de la façon dont vous le faites actuellement, puis migrer petit à petit vers la syntaxe d'ES6. Je vous recommande fortement d'adopter ES6 assez rapidement, vous gagnerez en productivité et en lisibilité avec cette nouvelle syntaxe. Vous vous demandez maintenant quand est-ce que vous allez pouvoir mettre en pratique tout ce que vous venez d'apprendre ? Eh bien, direction le prochain chapitre, où nous allons voir le transpileur TypeScript !

En résumé

- JavaScript profite de la nouvelle spécification standardisée ECMAScript 6, également nommée ES6.
- On peut développer en ES6 dès aujourd'hui, et utiliser un transpileur pour convertir notre code d'ES6 vers ES5, afin qu'il soit compréhensible par tous les navigateurs.
- ES6 nous permet d'utiliser les classes et l'héritage en JavaScript.
- ES6 introduit deux nouveaux mots-clés : *let* et *const*. Le premier permet de déclarer des variables et tend à remplacer *var*, et *const* permet de déclarer des constantes.
- Les Promesses offrent une syntaxe plus efficace que les callbacks, et tendent à les remplacer, surtout pour les développements relatifs à la programmation asynchrone.

Chapitre 3 : Découvrir TypeScript

Le JavaScript sous stéroïdes

Dans cette troisième partie nous allons aborder un des piliers d'Angular : TypeScript. Certains d'entre vous en ont peut-être déjà entendu parler, mais les autres ne vous en faites pas, nous allons démystifier tout cela immédiatement !

Avant de voir plus en détail ce qu'on va faire avec ce langage, et parce que je ne suis pas très inspiré, je vous propose la définition de Wikipédia :

"TypeScript est un langage de programmation libre et open-source développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript. C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript. Il a été co-crée par Anders Hejlsberg, principal inventeur de C#)"

Je n'aurais pas été plus clair. Mais je vous rassure on va creuser ça tout de suite, je ne vais pas vous laisser avec une simple définition de Wikipédia.

1. C'est quoi, « TypeScript » ?

Le langage JavaScript a toujours présenté les mêmes faiblesses : pas de typage des variables, absence de classes comme dans les autres langages... Ces faiblesses font que lorsque l'on commençait à écrire beaucoup de code, on arrivait vite à un moment où on s'emmêle les pinceaux ! Notre code devient redondant, perd en élégance, et devient de moins en moins lisible : on parle de code spaghetti. (Vous comprendrez tous seuls la référence aux célèbres pâtes italiennes !)

La communauté des développeurs, ainsi que certaines entreprises, se sont donc mises à développer des métalangages pour JavaScript : CoffeeJS, Dart, et TypeScript en sont les exemples les plus célèbres. Ces outils apportent également de nouvelles fonctionnalités qui font défaut au JavaScript natif, avec une syntaxe moins verbeuse. Par exemple, TypeScript permet de typer vos variables, ce qui permet d'écrire du code plus robuste. Une fois que vous avez développé votre application avec le métalangage de votre choix, vous devez le compiler, c'est-à-dire le transformer en du code JavaScript que le navigateur pourra interpréter.

En effet, votre navigateur ne sait pas interpréter le CoffeeJS, ni le Dart, ni le TypeScript. Vous devez d'abord compiler ce code en JavaScript pour qu'il soit lisible par votre navigateur.

Puisque ces méta-langages produisent du JavaScript au final, pourquoi je ne développerais pas directement en JavaScript ? (Pourquoi s'embêter avec cette étape intermédiaire ?)

En effet, c'est une excellente question. En théorie, on pourrait effectivement se passer de métalangages et réécrire nous-même les éléments dont nous avons besoin. En pratique, c'est très différent :

- D'abord vous n'allez pas réécrire tout ce que le métalangage vous apporte en JavaScript, vous en auriez pour plusieurs mois ou même plusieurs années pour réécrire ce que les équipes de développement de ces outils ont réalisé.
- Et vous devriez recommencer à chaque nouveau projet !

Autant vous dire que vous en auriez pour trop longtemps pour que ça en vaille la peine. Ne perdons plus de temps et voyons tout de suite le lien entre TypeScript et ce qui nous intéresse : Angular.

2. Angular et TypeScript

Ce que vous devez savoir, c'est qu'Angular vous recommande de développer vos applications avec TypeScript. Si vous allez sur la documentation officielle, vous verrez que tous les exemples de code et les explications sont données en TypeScript.

Il est donc obligatoire d'utiliser TypeScript avec Angular, vous n'avez pas vraiment le choix lorsque vous souhaitez vous lancer dans le développement d'applications Angular.

En bonus, Angular lui-même est écrit avec TypeScript. C'est pourquoi nous utiliserons ce langage, qui est d'ailleurs proposé par défaut sur la documentation officielle.

3. « Hello, TypeScript ! »

Pourquoi ne pas se familiariser avec TypeScript à travers une petite initiation ? Je ne vous propose rien d'original, nous allons commencer par le traditionnel « Hello, World ! » avec TypeScript.

C'est parti, je vous propose de créer un dossier où vous le souhaitez sur votre ordinateur, l'emplacement n'a pas beaucoup d'importance. A la racine de ce dossier, créez un fichier *test.ts*, et ajoutez-y le code suivant :

```
// Fonction qui retourne un message de bienvenue
function sayHello(person) {
  return "Hello, " + person;
}
// Création d'une variable "Jean"
var Jean = "Jean";
// On ajoute dans notre la page HTML un message
// pour affiche "Hello, Jean".
document.body.innerHTML = sayHello(Jean);
```

L'extension des fichiers TypeScript est *.ts*, et non *.js* . Pensez-y lorsque vous nommez vos fichiers !

Rien de passionnant vous me direz, et pourtant regardez tout ce qu'implique ce petit fichier :

- Vous avez créé un fichier TypeScript. Si, si, même si vous pensez que c'est du JavaScript, il s'agit bien de TypeScript ! (regardez l'extension de votre fichier, il s'agit bien de *test.ts* et non de *test.js*). Et oui, le TypeScript sera compilé en JavaScript : donc si vous écrivez directement du JavaScript, cela ne pose pas de problème pour TypeScript. C'est l'avantage, **TypeScript ne vous impose pas d'abandonner le JavaScript**, il vient juste l'améliorer, et cette souplesse est très agréable !
- Le revers de la médaille, c'est que le navigateur est bien incapable de lire du TypeScript, tant que celui-ci n'a pas été compilé en JavaScript !

Pour remédier à ce problème, nous allons installer le nécessaire pour transformer notre TypeScript en JavaScript. Toujours motivé ? Alors c'est parti !

Pour commencer, ouvrez un terminal et placez-vous à la racine de votre projet. Assurez-vous d'abord que Node.js et Npm sont installés, et vérifiez leurs versions respectives grâce aux commandes suivantes :

```
node -v  
v10.15.3
```

Puis vérifiez aussi que vous avez aussi Npm d'installé :

```
npm -v  
6.4.1
```

Si vous n'avez pas exactement les mêmes versions que moi, ce n'est pas grave. Par contre vous devez avoir au moins la version 10 pour Node. Si les deux commandes ci-dessus vous affichent une erreur, c'est probablement que vous n'avez pas installé Node: <https://nodejs.org/fr>. Vous allez en avoir besoin pour la suite, revenez à ce chapitre une fois que vous l'aurez installé.

Bon, ce n'est pas tout, mais nous devons installer le compilateur TypeScript. Il suffit d'une seule commande pour cela :

```
npm install -g typescript
```

Ensuite, vérifiez rapidement que l'installation a bien fonctionné :

```
tsc -v  
version x.y.z; // ce seront des chiffres chez vous !
```

Voilà, c'est aussi simple que ça !

Là aussi, la version a de l'importance. Essayez d'avoir au moins la version 3.4, mais pas en-dessous, ou vous serez bloqués dans la suite du cours. Je vous avais prévenu que c'était un écosystème bourgeonnant !

Mais le meilleur arrive, nous allons transformer notre code TypeScript en JavaScript grâce à la commande suivante :

```
tsc test.ts
```

Si vous regardez le contenu du dossier de votre application, vous verrez qu'un nouveau fichier est apparu : *test.js* !

Qu'est-ce que nous attendons pour ouvrir ce fichier qui a été créé spécialement pour nous ? Ouvrez donc le fichier *test.js* avec votre éditeur de texte, voici ce qu'il devrait contenir :

```
// Fonction qui retourne un message de bienvenu
function sayHello(person) {
  return "Hello, " + person;
}
// Création d'une variable "Jean"
var Jean = "Jean";
// On ajoute dans notre la page HTML un message
// pour afficher "Hello, Jean".
document.body.innerHTML = sayHello(Jean);
```

Mais rien n'a changé dans ce fichier, à part son extension ts en js !

Ben oui, c'est du JavaScript ! La seule chose que le compilateur a fait, c'est de supprimer les espaces entre les lignes ! Qu'est-ce que vous vouliez qu'il fasse, notre fichier initial ne contenait que du JavaScript.

Bon je vous rassure, on va faire des choses un peu plus intéressantes dans la suite de ce chapitre, c'était surtout pour vous montrer le fonctionnement !

*A quoi servent les fichiers de définition dans TypeScript ? Les fichiers avec l'extension *.d.ts ?*

Si vous voulez utiliser des bibliothèques externes écrites en JavaScript, vous ne pouvez pas savoir le type des paramètres attendus par telle ou telle fonction d'une bibliothèque. C'est pourquoi la communauté TypeScript a créé des interfaces pour les types et les fonctions exposés par les bibliothèques JavaScript les plus populaires (comme jQuery par exemple). Les fichiers contenant ces interfaces ont une extension spéciale : *.d.ts. Ils contiennent une liste de toutes les fonctions publiques des librairies utilisées.

Mais concernant votre code, sachez que depuis TypeScript 1.6, le compilateur est capable de trouver tout seul ces interfaces avec les dépendances de notre répertoire *node_modules*. On n'a donc plus à le faire par nous-même !

4. Pas de limites avec TypeScript

Bon, nous allons maintenant voir ce que TypeScript a dans le ventre, et ce qu'il va nous apporter avec Angular. Et quoi de plus normal pour commencer, que de voir les **types** de **TypeScript**. La syntaxe pour déclarer une variable typée avec TypeScript est la suivante :

```
// On déclare une variable typée en TypeScript !  
var variable: type;
```

Vous êtes libre de choisir le type que vous voulez. Vous pouvez utiliser des types basiques ou alors créer vos propres types :

```
// On déclare un nombre :  
var lifePoint: number = 100;  
  
// On déclare une chaîne de caractère :  
var name: string = 'Green Lantern';  
  
// On déclare une variable qui correspond à une classe de notre application !  
var greenLantern: Hero = new Hero(lifePoint, name);  
  
// Je peux même créer un tableau de héros, qui contient tous les héros de mon application !  
var heros: Array<Hero> = [greenLantern, superMan];
```

Le typage de nos variables est très pratique, puisque cela vous permet d'être sûrs du type des variables que vous utilisez. Pouvoir être sûr que notre tableau héros ne contient que des héros et pas des nombres ou autres choses, est plutôt confortable. Notre code devient plus robuste et plus élégant.

Pour vous prouver ce que je viens de vous dire, essayons d'ajouter une chaîne de caractères à la variable point de vie. Redéfinissez le code de votre fichier *test.ts* comme ceci :

```
var lifePoint: number = 100;  
lifePoint = 'some-string';
```

Ensuite essayez de compiler ce code :

```
tsc test.ts
```


Vous obtiendrez alors cette magnifique erreur dans votre console :

```
Test.ts(7,1): error TS2322: Type 'string' is not assignable to type 'number'.
```

L'erreur est explicite, à la ligne 2 de votre fichier *test.js*, vous avez essayé d'assigner une string à une variable que vous aviez déclarée comme étant un nombre, et TypeScript vous l'interdit bien sûr.

En revanche si vous faites :

```
// Ce code ne causera pas d'erreur car lifePoint est bien de type number.  
lifePoint = 50;
```

Et sachez que TypeScript s'occupe de vérifier le type de toutes les variables typées pour nous. Si dans notre tableau de héros nous essayons d'ajouter un nouvel élément qui ne soit pas un héros, TypeScript nous retournerait une erreur. Bref, c'est magique non ?

Sachez que TypeScript propose également un type nommé "any" qui signifie 'tous les types'.

4.1. TypeScript et les fonctions

TypeScript permet de spécifier un type de retour pour nos fonctions. Imaginons que nous voulons créer une fonction pour générer des *Heros* :

```
// Un constructeur pour notre classe Hero  
// On spécifie le type de retour après les ':', ici Hero.  
function createHero(lifePoint: number, name: string): Hero {  
    var hero = new Hero();  
    hero.lifePoint = lifePoint;  
    hero.name = name;  
    return hero;  
}
```

Cette fonction doit retourner une instance de la classe *Hero*, comme indiqué après les ':' à la ligne 3. Vous avez également remarqué qu'on a pu typer les paramètres de notre fonction ? Là-aussi, il s'agit d'un gros plus qu'apporte TypeScript, et qui nous permet de développer un code plus sérieux qu'avec le JavaScript natif !

Vous pouvez également ajouter des paramètres optionnels à vos fonctions. Par exemple, ajoutons à notre constructeur précédent un paramètre facultatif pour indiquer la planète d'origine d'un héros, grâce à l'opérateur '?' :

```
// Le '?' indique que le paramètre 'planet' est facultatif :  
function createHero(lifePoint: number, name: string, planet?: string):  
    Hero {  
    var hero = new Hero();  
    hero.lifePoint = lifePoint;  
    hero.name = name;  
    if(planet) hero.planet = planet;  
  
    return hero;  
}
```

4.2. Les classes avec TypeScript

Nous allons faire une petite expérience intéressante. Nous allons créer une classe vide, et nous allons la transcompiler vers ES5, puis vers ES6. Créer donc un fichier *test.ts* quelque part sur votre ordinateur, et ajoutez-y le code TypeScript suivant :

```
// test.ts
class Test {}
```

Maintenant, nous allons compiler ce code vers du JavaScript ES5, c'est-à-dire vers une spécification de JavaScript qui ne supporte pas encore le mot-clé *class*. Pour cela, exécutons la commande TypeScript que nous avons déjà vu, mais en rajoutant une option pour demander explicitement à TypeScript de compiler vers du ES5 :

```
tsc --t ES5 test.ts
```

Maintenant, si vous allez jeter un coup d'œil au fichier *test.js* qui vient d'être généré par TypeScript, voici ce que vous trouverez à l'intérieur :

```
// Ma classe en ES5 (test.js)
var Test = (function () {
  function Test() {
  }
  return Test;
})();
```

Mais, c'est une classe ça ???

Et bien d'une certaine manière, oui ! Vous avez sous les yeux une classe, mais avec la syntaxe d'ES5. Vous y reconnaitrez une IIFE.

IIFE : Immediately-invoked Function Expression. Ce sont des fonctions qui s'exécutent immédiatement, et dans un contexte privé.

Essayons maintenant de compiler le même code, mais vers ES6 :

```
tsc --t ES6 test.ts
```

Si vous affichez maintenant le code généré dans *test.js* :

```
// Ma classe en ES6 (test.js)
class Test {
}
```

Mais, on est revenu au point de départ, non ? C'est exactement le code TypeScript qu'on avait au début !

Oui, vous avez raison, et c'est parfaitement normal. Vous savez pourquoi ? Parce qu'ES6 supporte parfaitement les classes, comme nous l'avons vu dans le chapitre précédent. Comme TypeScript **et** ES6 supportent les classes, et bien vous ne voyez pas de différences !

4.3. Les Décorateurs

Vous vous rappelez des métadonnées que nous avons vues dans le premier chapitre ? Non ? Tant pis, rappelez-vous simplement qu'il s'agissait d'ajouter des informations à nos classes via des annotations. TypeScript propose un moyen d'implémenter ces métadonnées grâce aux décorateurs.

Prenons un cas simple, vous avez développé une classe, et vous souhaitez indiquer à Angular qu'il s'agit bien d'une classe de composant, et pas d'une classe lambda, et bien pour cela vous utiliserez les décorateurs TypeScript :

```
// Exemple d'utilisation des décorateurs  
@Component({  
  selector: 'my-component',  
  template: 'my-template.html'  
})  
export class MyComponent {}
```

En Angular, on utilisera régulièrement des décorateurs, qui sont fournis par le Framework. Retenez également que tous les décorateurs sont préfixés par @.

Les Décorateurs sont assez récents dans TypeScript, depuis la version 1.5 exactement. Voilà pourquoi il est important de veiller à avoir une version à jour.

5. Conclusion

TypeScript est un méta-langage qui est surtout connu pour apporter le typage à JavaScript, mais il s'agit également d'un transpileur, capable de générer du code vers ES5 ou ES6. TypeScript apporte évidemment d'autres fonctionnalités : les valeurs énumérées, les interfaces, les décorateurs, etc.

L'objectif de ce chapitre était de vous initier à TypeScript, pour voir ensuite son fonctionnement avec Angular. Si vous voulez pousser plus loin vos connaissances en TypeScript, je vous invite à regarder la documentation officielle (www.typescriptlang.org/docs). La documentation est disponible uniquement en anglais, mais il s'agit d'anglais technique et la majorité de la documentation est composée d'exemples de code, vous devriez largement vous y retrouver même si vous n'êtes pas très à l'aise avec cette langue.

En résumé

- Angular a été construit pour tirer le meilleur parti d'ES6 et de TypeScript.
- Il faut utiliser TypeScript pour ses développements avec Angular.
- Angular a été développé avec TypeScript. Il est fortement recommandé d'adopter TypeScript pour vos développements avec Angular.
- L'extension des fichiers TypeScript est `*.ts` et non `*.js`.
- Le navigateur ne peut pas interpréter le TypeScript, il faut donc compiler le TypeScript vers du code JavaScript.
- TypeScript apporte beaucoup de fonctionnalités complémentaires à JavaScript comme le typage des variables, la signature des fonctions, les classes, la généricité, les annotations, ...
- Les annotations TypeScript permettent d'ajouter des informations sur nos classes, pour indiquer par exemple que telle classe est un composant de l'application, ou telle autre un service.

Chapitre 4 : Les Web Components

L'avenir du web ?

Avant de commencer à développer notre toute première application Angular, où nous allons réaliser un splendide « *Hello, World !* », je tenais à vous présenter les Web Components, ou Composant Web. Les Composants Web sont indépendants d'Angular. Cependant les concepteurs d'Angular ont fait un effort particulier pour les intégrer dans leur Framework, et je pense que ce serait une bonne chose que vous sachiez de quoi il s'agit. Ce chapitre est très théorique, vous pouvez le lire en diagonale si vous êtes pressé de passer à la pratique, et revenir lire ce chapitre plus tard. C'est comme vous voulez !

1. Introduction aux Web Components

Les *Web Components* désignent un standard qui permet aux développeurs de créer des sections complètement autonomes au sein de leurs pages web.

On peut par exemple créer un composant web qui gère l'affichage d'articles dans notre application : ce composant web fonctionnera indépendamment du reste de la page, il possèdera son propre code HTML, son propre code CSS et son propre code JavaScript, encapsulé dans le composant web. Il faudra ensuite insérer ce composant web dans la page principale de notre application pour indiquer que, à tel endroit, nous voulons afficher des articles grâce à ce composant web.

On peut voir les Web Components comme des widgets réutilisables.

Les Web Components utilisent des capacités standards, nouvelles ou en cours de développement des navigateurs. Il s'agit d'une technologie récente qui n'est pas encore supportée par tous les navigateurs. Pour les utiliser dès aujourd'hui, nous devons utiliser des *polyfills* pour combler les lacunes de couverture des navigateurs.

Un *polyfill* est un ensemble de fonctions, souvent sous forme de scripts JavaScript, permettant de simuler sur un navigateur web ancien des fonctionnalités qui ne sont pas nativement disponibles.

Les Web Components sont composés de quatre technologies différentes, qui peuvent chacune être utilisées séparément, mais qui une fois assemblées forment le standard des Web Components :

- Les éléments personnalisés (*Custom Elements*) ;
- Le DOM de l'ombre (*Shadow DOM*) ;

- Les templates HTML (*HTML Templates*) ;
- Les imports HTML (*HTML Imports*).

Pour ceux qui ne s'en souviennent plus, le DOM est une représentation de votre code HTML sous forme d'arbre. Ainsi un élément `` a des éléments fils ``. L'élément racine du DOM est donc la balise `<html>`.

Je vais vous présenter chacune de ces technologies : le plus important est que vous compreniez pourquoi telle ou telle technologie a été inventée et à quoi elle sert.

Par ailleurs, je vous conseille vivement de lire ce chapitre de haut en bas, il y a un certain enchaînement logique entre tous ces éléments !

2. Les éléments personnalisés

Les éléments personnalisés permettent de créer des balises HTML personnalisées dans vos pages web, selon les besoins de votre application.

Vous vous demandez sûrement quel est l'intérêt de ces éléments personnalisés, étant donné que vous pouvez déjà mettre dans votre code une balise du type `<balise-inventée>`, et ensuite appliquer du JavaScript dessus, ou même du CSS :

```
<!-- Je crée une balise au hasard -->
<une-balise-inventee></une-balise-inventee>
<style>
/* Je rajoute un peu de style à ma nouvelle balise */
une-balise-inventee {
  width: 200px;
  height: 50px;
  border: 1px solid orange;
}
</style>
<script>
// Ce script permet d'afficher le nombre de balises personnalisées sur la page :
var elementPerso = document.getElementsByTagName("une-balise-inventee");
var quantite = elementPerso.length;
alert("Il y a " + quantite + " éléments <une-balise-inventee> dans ce document.");
</script>
```

Alors je vous recommande tout de suite de ne jamais faire ça, et ce pour plusieurs raisons :

- Votre code n'est pas valide, et développer du code invalide n'est pas très bien vu pour un développeur !
- Si tout le monde faisait comme vous, il serait impossible de s'y retrouver : imaginez qu'une personne récupère votre code, et qu'à la place des balises `<p>`, ``, `<h1>`, ... elle se retrouve face à des balises `<une-balise-inventee>`, `<element-important>`... cette personne ne saurait plus où donner de la tête ! Même si l'idée de créer ses propres balises peut être intéressante, vous êtes d'accord sur l'idée qu'il faut que tout le

monde respecte les mêmes standards pour pouvoir s'y retrouver, non ?

- Vous ne profitez pas des spécifications des Web Components : en respectant ces standards, vous pourriez profiter d'un coup d'un code valide, d'un standard respecté par tous les développeurs, et également des *lifecycle callbacks*. Bref, ça vaut le coup !

Heu, que signifie exactement 'lifecycle callbacks' ?

Il n'y a rien de compliqué ! Prenons un mot après l'autre :

- *Lifecycle* ou cycle de vie : désigne le fait que vos éléments personnalisés vont passer par plusieurs étapes au cours de leur existence (création, modification, suppression).
- *Callbacks* : désigne simplement une fonction qui sera appelé lorsque votre élément passera par une des étapes de son cycle de vie.

Par exemple, vous pouvez attacher une fonction sur un de vos éléments, qui ne sera appelée que lorsque l'élément sera créé. Ainsi vous pouvez attacher des comportements à différentes étapes du cycle de vie d'un composant.

Il y a quatre *lifecycle callback* à avoir en tête si vous voulez vous en servir :

1. **createdCallback** – Le comportement à définir quand l'élément est enregistré auprès du navigateur via *Document.registerElement* (Nous allons voir de quoi il s'agit ci-dessous).
2. **attachedCallback** – La fonction qui est appelée quand l'élément est inséré dans le DOM.

3. **detachedCallback** – La fonction qui est appelée quand l'élément est retiré du DOM.
4. **attributeChangedCallback** – Le comportement de l'élément quand un de ses attributs est ajouté, modifié ou retiré.

Bon ok, tu m'as convaincu d'utiliser les standards du Web Component ! Mais comment on fait, tu ne nous as rien dit !

Alors, la clé pour utiliser les éléments personnalisés, c'est la méthode *Document.registerElement*. On utilise cette méthode pour enregistrer de nouveaux éléments, en lui passant le nom de notre élément ainsi qu'un tableau d'options, notamment un prototype pour notre nouvel élément.

Ainsi le navigateur sait que l'élément que nous venons de lui déclarer est un élément personnalisé à part entière, et qu'il ne s'agit pas juste d'une balise malformée. En effet, si vous utilisez cette méthode, elle ajoutera à votre élément personnalisé l'interface *HTMLElement*. Cette interface représente n'importe quel élément HTML, ce qui signifie que votre code sera valide !

En revanche, si vous n'utilisez pas cette méthode et que vous faites votre élément personnalisé dans votre coin, alors votre code ne sera pas valide, et implémentera probablement l'interface *HTMLUnknownElement* !

Vous pouvez aussi construire votre propre élément personnalisé à partir d'un élément natif. Prenons `<button>` par exemple. Alors vous ne pourrez pas utiliser un nom personnalisé comme `<my-button>`, vous devrez utiliser une syntaxe comme `<button is='my-button'>`.

Bon je sens que vous êtes impatient de voir du code, et d'arrêter de m'écouter parler tout seul !

Rassurez-vous, voici un exemple de qualité pour me rattraper. Nous allons créer un petit élément personnalisé pour afficher la vignette et quelques informations sur des super-héros.

Créons tout de suite un élément personnalisé nommé `<super-heros>`. Pour cela nous avons besoin d'un nom et d'un prototype. C'est parti !

Faites attention, le nom de votre élément personnalisé doit contenir un tiret pour indiquer au navigateur qu'il ne s'agit pas d'un élément natif. Généralement on préfixe notre élément par un diminutif du nom de notre application, comme : 'monApp-monElement'.

Vous devez également associer un template à votre élément, sinon le navigateur ne saura pas ce qu'il doit afficher.

Créer donc une page *index.html* avec le code suivant :

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Test</title>
</head>
<body>
  <script>
    // On crée une classe pour notre élément personnalisé,
    // et on lui ajoute un template avec la méthode 'innerHTML'.
    class SuperHero extends HTMLElement {
      constructor() {
        super();
        this.innerHTML = '<h1>Superman</h1>';
      }
    }

    // On définit notre élément personnalisé avec la méthode 'define'.
    customElements.define('super-hero', SuperHero);
    // On ajoute notre élément sur la page web !
    document.body.appendChild(new SuperHero());
  </script>
```

```
</body>  
</html>
```

Affichez ensuite cette page dans votre navigateur préféré : vous verrez affiché *Superman* en titre sur votre page !

Si vous inspectez votre code, vous verrez que le DOM de votre page contient notre nouvelle balise :

```
▶ <head>...</head>  
▼ <body>  
  ▶ <script>...</script>  
  ▼ <super-heros>  
    <h1>Superman</h1>  
  </super-heros>  
</body>  
</html>
```

La console du navigateur lorsque j'inspecte mon titre « Superman ».

Pour vous prouver que notre code est bien valide, essayez de faire valider cette page HTML par le W3C Validator : validator.w3.org (en copiant et collant le code ci-dessus), vous obtiendrez alors ce magnifique message :

Document checking completed. No errors or warnings to show.

Notre code est valide, en voici la preuve !

Vous voyez bien que nous ne sommes pas fous, nous avons bien créé une nouvelle balise HTML valide !

Qu'attendons-nous pour conquérir le monde !?!

*Oui mais on aurait pu faire ça avec
<h1>Superman</h1>, non ?*

Ah, la question qui fâche ! Bon, oui, sur la forme vous avez raison, mais vous auriez perdu les avantages des éléments personnalisés :

- **Réutilisabilité** : Maintenant que nous avons défini notre élément personnalisé, nous pouvons le réutiliser autant de fois que vous le voulez. Imaginons que vous ayez développé un élément permettant d'afficher la présentation d'un héros, et bien il vous suffit d'appeler cet élément autant de fois que vous avez de héros !
- **Personnalisation** : Je ne vous l'ai pas encore montré, mais nous pouvons ajouter des propriétés et des méthodes personnalisées encapsulées dans notre élément, ce que nous ne pouvons pas faire avec une simple balise.
- **Encapsulation** : Tout le code que nous développons au sein de notre élément peut être encapsulé dedans, et ainsi nous pouvons développer nos pages HTML comme un assemblage d'éléments personnalisés !

Attend, tu viens de dire que l'on peut encapsuler notre code HTML, avec le code Javascript et CSS associés, mais comment éviter les conflits entre les éléments alors ?

En effet, que-ce passe-t-il si vous avez sur votre page un élément A qui indique que tous ces paragraphes doivent être écrit en jaune, et un élément B qui spécifie que tous les paragraphes doivent être écrit en rouge ? Lequel l'emporte ? Et que deviennent les paragraphes qui n'appartiennent à aucun élément, et qui sont simplement présents sur la page ?

Je vous rassure, la spécification des composants Web prend en compte ce genre de problématique, et c'est pour ça qu'on va tout de suite voir ce qu'est le DOM de l'ombre. Il permettra d'éviter les conflits entre les éléments personnalisés.

Pour rappel, les éléments personnalisés que nous venons de voir sont un des quatre standards des composants web, mais vous pouvez tout à fait les utiliser pour eux-mêmes, indépendamment des composants web.

3. Le DOM de l'ombre

Ce nom est génial, avouons-le. Imaginez la tête que vont faire vos camarades ou vos collègues quand vous leur direz que vous utilisez le DOM de l'ombre pour vos développements !

Dans la suite du cours j'emploierai le terme Anglais : le Shadow DOM, c'est encore plus impressionnant !

L'objectif du *shadow DOM* est de permettre d'encapsuler nos éléments personnalisés de manière sûre (c'est-à-dire de les isoler du reste de la page), afin d'éviter tout conflit avec les éléments de la page. Retenez ce principe, et vous avez déjà fait la moitié du travail !

Le Shadow DOM représente donc un ensemble de spécifications qui est supporté par les standards récents du Web, afin d'isoler le JavaScript, le CSS et le HTML d'un élément personnalisé du reste de la page.

Cela permet donc de créer un nouveau DOM qui n'est pas rattaché au DOM principal, donc le DOM principal ne peut plus accéder à nos shadow DOM, et c'est là que ça devient intéressant.

Pourquoi tu dis 'nos shadow DOM', on peut en créer plusieurs ?

Oui, vous aurez exactement un nouveau Shadow DOM pour chaque élément personnalisé, afin que chacun d'eux puisse profiter du système d'encapsulation. Vous voulez voir en pratique comment cela fonctionne ? Pas de soucis, nous allons créer un Shadow DOM.

Un Shadow DOM doit toujours être rattaché à un élément existant, que ce soit un élément présent dans votre fichier HTML, ou un élément créé depuis un script. Et il est bien sûr possible de rattacher un Shadow DOM à un élément personnalisé, comme *<mon-element>* par exemple !

Créons simplement un paragraphe avec un identifiant unique :

```
<!-- Un simple paragraphe -->  
<p id="paragraphe-shadow"><p>
```

Et maintenant ajoutons un Shadow DOM sur ce paragraphe :

```
// On crée un nouveau Shadow DOM sur un élément de notre page,  
// qui possède l'identifiant 'paragraphe-shadow' :  
var shadow = document.querySelector('#paragraphe-shadow').attachShadow({mode:
```

```
'open'}});  
  
// Pour l'instant notre Shadow DOM est vide, mais nous pouvons lui ajouter du contenu.  
shadow.innerHTML = "<p id='shadow'>Salut, Shadow DOM !</p>";  
  
// Si on recherche notre contenu caché depuis la console du navigateur,  
// alors la commande suivante ne retourne rien, car le DOM n'a pas accès au Shadow  
DOM !  
document.querySelectorAll('#paragraphe-shadow');
```

Comme vous le constatez, il suffit d'utiliser la méthode *Element.attachShadow* pour instancier un nouveau *Shadow DOM*. Cette méthode prend un paramètre qui permet d'indiquer si on souhaite accéder au DOM de l'ombre en utilisant du code JavaScript écrit dans le contexte principal de la page.

C'est le cas pour nous car nous avons définis ce paramètre à *'open'*. La méthode *attachShadow* retourne ensuite un nouveau *Shadow DOM* si vous avez défini le paramètre à *open*, et retourne *null* si vous avez défini le paramètre à *closed*.

Comme promis, vous pouvez ajouter du CSS qui ne sera appliqué qu'aux éléments de ce Shadow DOM, en l'occurrence votre élément *'paragraphe-shadow'* :

```
// On ajoute un peu de style à notre Shadow DOM  
shadow.innerHTML += '<style>p {color: red;}</style>';
```

Si vous testez ce code, vous vous apercevrez que seuls les paragraphes de notre *Shadow DOM* ont un texte rouge !

3.1. Angular et le Shadow DOM

Vous devez savoir que Angular n'utilise pas directement les spécifications du Shadow DOM que nous venons de voir.

QUOI ? On apprend des trucs qui ne servent à rien depuis le début ?!

Non, je ne vous ai pas exactement menti, disons que tout cela était nécessaire.

Derrière les spécifications du Shadow DOM, vous avez déjà compris le mécanisme d'encapsulation sous-jacent. Et bien Angular utilise lui aussi son propre mécanisme d'encapsulation en interne, qui simule un comportement d'encapsulation, et qui a l'avantage de fonctionner sur tous les navigateurs, contrairement au Shadow DOM qui est assez récent et qui n'est pas supporté par les navigateurs plus anciens. Du coup, c'est bénéfique pour nous !

En fait, Angular peut quand même utiliser les spécifications du Shadow DOM si on le lui demande explicitement. Pour être exact, Angular propose trois techniques d'encapsulation des éléments personnalisés. Pour chacun de nos composants que nous créons avec Angular, nous avons le choix entre les systèmes d'encapsulation suivants :

- **Emulated** : C'est la méthode par défaut qu'utilise Angular, qui simule le fonctionnement du Shadow DOM mais qui a l'avantage de fonctionner sur tous les navigateurs. En interne, Angular simule le Shadow DOM en suffixant les sélecteurs utilisés pour cibler les composants. C'est cette méthode que nous utiliserons dans ce cours. Comme il s'agit de la méthode par défaut, le comportement de notre application sera le même sans ajouter de code particulier, Angular s'occupe de tout pour nous ! Elle n'est pas belle la vie ?

- **Native** : Cette méthode d'encapsulation supporte les spécifications du Shadow DOM, nous obtenons donc le même comportement d'encapsulation mais avec une portabilité moindre sur les anciens navigateurs.
- **None** : Pour une raison quelconque, vous pouvez vouloir ne pas utiliser le mécanisme d'encapsulation des vues, à vos risques et périls !

4. Les templates HTML

Un template HTML permet de déclarer des petits morceaux de HTML qui ne seront pas mis en forme lors du chargement de la page, mais qui pourront être copiés, complétés, puis insérés dans le document grâce au JavaScript : on pourrait parler de *module HTML*. Ce concept de découpe en module existe déjà beaucoup côté serveur (JavaEE, .NET, Symfony, Django...) mais HTML propose désormais un mécanisme similaire. On peut donc créer de petits éléments de contenu HTML pouvant être réutilisés dans différents endroits de notre application.

Le contenu d'un template HTML est tout de même parsé pendant le chargement de la page, afin de s'assurer qu'il soit valide.

Créons sans plus tarder un template HTML, qui a pour objectif d'afficher un héros. Pour cela nous utilisons la balise `<template>` dédiée :

```
<template id="super-heros">
<style>
h1 { color: green; };
</style>
<h1>Green Lantern</h1>
</template>
```

Vous voyez, il n'y a rien de bien compliqué, nous utilisons simplement la balise dédiée à la création de template HTML : `<template>`. Pour l'instant, évidemment ce code ne fait rien qui soit visible à l'écran pour l'utilisateur, puisqu'il est destiné à être appelé en JavaScript. Ajoutons donc un script sur notre page afin d'interagir avec notre template HTML :

```
// On reprend notre élément personnalisé 'super-heros' précédent
customElements.define('super-hero',
class extends HTMLElement {
  constructor() {
    super();
    let template = document.getElementById('super-hero');
    // On récupère le contenu de notre template :
```

```
let templateContent = template.content;
// On ajoute le contenu de notre template au DOM de l'ombre
const shadowRoot = this.attachShadow({mode:'open'})
.appendChild(templateContent.cloneNode(true));
}
})
```

Désormais, dans votre page, vous pouvez utiliser : `<super-hero>`
`</super-hero>`.

Comme vous pouvez le voir, on commence à avoir un système d'encapsulation solide en HTML : *composants personnalisés*, *Shadow DOM* et *templates HTML*. Si seulement on pouvait déclarer tout ça dans un fichier à part, et qu'il suffirait ensuite d'importer dans notre page web principale, ça serait vraiment la classe !

Attendez, mais... je crois bien que c'est possible avec les *imports HTML* !

5. Les imports HTML

Les *imports HTML* sont le dernier standard qu'il nous manque pour pouvoir développer de véritables composants web dans un fichier dédié, fonctionnant indépendamment du reste de la page.

Ils rendent désormais possible d'importer un fichier HTML en utilisant la balise `<link>` dans un autre document HTML :

```
<link rel='import' href='un-autre-fichier.html'>
```

En fait, **on peut importer du HTML dans du HTML**, et le fichier importé fonctionne en autonomie, il contient son propre HTML, CSS et JavaScript et ne vient pas interférer avec votre DOM !

Certains sites proposent même des composants prêts à l'emploi ! Il y a également des bibliothèques connues qui ont été conçues pour faciliter l'intégration des Web Composants, comme la librairie Polymer de Google, et qui vous permet de faire ça :

```
<!-- Polyfill pour les Composants Web, afin de supporter les navigateurs plus anciens -->
<script src="components/webcomponentsjs/webcomponents-lite.min.js"></script>

<!-- Un import de HTML -->
<link rel="import" href="components/google-map/google-map.html">

<!-- On utilise l'élément personnalisé importé ! -->
<google-map latitude="37.790" longitude="-122.390"></google-map>
```

Le code parle de lui-même, et il vous affichera une *Google map* centrée sur San Francisco, rien que ça !

Bon je pense que je vous en ai déjà pas mal dit, n'hésitez pas à continuer à voir ce que sont les Composants Web si le sujet vous intéresse !

6. Conclusion

Voilà, c'est tout pour ce chapitre. J'espère que vous aurez appris plein de choses sur l'avenir du développement web. Les Web Components auront certainement un rôle à jouer dans le web de demain, et il serait dommage de passer à côté !

Retenez que les Web Components sont composés de quatre technologies différentes qui fonctionnent ensemble, pour nous permettre de réaliser des applications web avec du HTML modulaire.

En résumé

- Les Composants Web permettent d'encapsuler du code HTML, CSS et JavaScript qui n'interfère pas avec le DOM principal de la page web.
- Les Composants Web sont un assemblage de quatre standards indépendants : les éléments personnalisés, le Shadow DOM, les templates HTML et les imports HTML.
- Les éléments personnalisés permettent de créer ses propres éléments HTML valides.
- Les templates HTML permettent de développer des morceaux de code HTML qui ne sont pas interprétés au chargement de la page.
- Les imports HTML permettent d'importer du HTML dans du HTML.
- Les spécifications des Composants Web sont assez récentes et ne fonctionnent pas forcément sur tous les navigateurs, mais Angular propose de simuler en interne certaines de ces spécifications pour augmenter leur portabilité.

Chapitre 5 : Premiers pas avec Angular

Toujours commencer par « Hello, World ! »

Bon, je vous l'avais promis, et nous y arrivons !

Nous allons réaliser une superbe démonstration de « *Hello, World !* » avec Angular (enfin !). Il nous aura fallu quelques chapitres théoriques avant de commencer, mais je pense c'était nécessaire que vous ayez un aperçu global de cet écosystème.

Avant de se lancer tête baissée dans ce qui nous attend, je vous propose un petit plan de bataille :

- D'abord, installer un environnement de développement.
- Ecrire le composant racine de notre application : rappelez-vous que notre application Angular n'est qu'un assemblage de composants, et donc il faut au moins un composant pour faire fonctionner une application.
- Informer Angular du composant avec lequel nous souhaitons démarrer notre application. Pour nous ce sera facile, car nous n'aurons qu'un seul composant, le composant racine.
- Ecrire une simple page `index.html` qui contiendra notre application.

Je vous propose de ne pas traîner et de commencer tout de suite ! Allez, au boulot !

Par défaut, un navigateur affiche toujours le fichier nommé `index.html` d'un répertoire, c'est pourquoi nous aurons un fichier `index.html` à la racine de notre projet.

1. Les outils du développeur Angular

Pour développer avec Angular, il y a un certain nombre d'éléments à installer sur votre machine.

Nous allons réaliser un petit état des lieux avant de commencer, ça ne fera de mal à personne, et au moins nous serons sûr de tous partir sur le même pied d'égalité. Voyons tout de suite les éléments indispensables à installer ou à mettre à jour sur votre poste de travail.

2. Installer Node.js et Npm

Node.js et Npm (Node Module Packager, pour être exact) sont deux outils indispensables pour le développeur JavaScript moderne. Node.js permet d'exécuter du code JavaScript côté serveur, et Npm permet d'installer et gérer les dépendances de paquets JavaScript, comme Angular par exemple, qui est avant tout un simple paquet JavaScript.

Node.js ne nous intéressera pas en soi pendant ce cours, mais lors de son installation, Npm est automatiquement installé avec lui par défaut. Il va donc nous être utile pour pouvoir commencer à travailler.

Il faut donc commencer par installer Node.js : nodejs.org/fr.

Il s'agit d'un simple fichier à télécharger, et que vous exécutez ensuite. Une fois Node.js et Npm installé sur votre poste de travail, nous allons vérifier que l'installation s'est bien déroulée en affichant la version respective de chacun de ces outils, depuis un terminal.

Exécutez donc les commandes `node -v` et `npm -v`. Si des numéros de versions s'affichent, alors nous sommes prêt à passer à la suite.

Pour ouvrir un terminal sous Windows, tapez `Windows + R`, puis « `cmd` », et enfin la touche Entrée. Sur Mac, vous pouvez chercher le terme « `terminal` » avec Spotlight (`cmd + barre espace`).

3. Installer un éditeur de code qui supporte le TypeScript

Comme nous l'avons vu, le développement d'applications Angular se fait avec TypeScript, et il va donc falloir installer un éditeur de code qui supporte le TypeScript.

Heureusement, il existe plusieurs IDEs qui supportent TypeScript et qui pourront vous aider lors de vos développements :

- Visual Studio Code : Recommandé par Microsoft pour les développements Angular, il s'agit d'un IDE complet, populaire et gratuit. Je ne peux que vous conseiller cet IDE si vous ne savez pas lequel choisir, vous ne pouvez pas vous tromper.
- WebStorm (Payant mais licence de 1 an offerte si vous êtes étudiant).
- Sublime Text avec certains plugins à installer.
- Il y en a plein d'autres, d'ailleurs le site officiel de TypeScript a listé sur sa page d'accueil les IDEs recommandés pour TypeScript.

Retenez que ces outils vous simplifient la vie, mais qu'ils ne sont pas indispensables. Choisissez l'environnement de développement avec lequel vous êtes le plus à l'aise, et ne vous embêtez pas avec un nouvel IDE si vous êtes satisfait du vôtre.

Le terme IDE désigne un environnement de développement : il s'agit souvent d'un éditeur de texte amélioré, spécialisé dans le développement logiciel avec tel ou tel langage.

Pour ma part, j'ai prévu d'utiliser Visual Studio Code, mais vous êtes libres de travailler avec l'outil que préférez. Bon, on commence quand, pour de vrai ?

4. Démarrer un projet Angular

Il y a plusieurs moyens de démarrer un projet Angular, certains sont plus rapides que d'autres.

Mais il existe une façon standard de démarrer son projet Angular, c'est via l'outil Angular CLI (*cli.angular.io*), qui vous permet de mettre en place un nouveau projet Angular en une seule ligne de commande !

Le terme « CLI » signifie « Command Line Interface ». En français, cela veut dire « Interface en ligne de commande ». Pour être plus concret, cela signifie que Angular CLI est un outil que l'on utilise en exécutant des commandes dans un terminal, c'est tout.

Pour commencer, nous allons donc voir comment installer ce formidable outil, et ensuite nous l'utiliserons pour mettre en place notre première application Angular !

Aussi, sachez que ce que nous sommes entrain d'installer et de mettre en place maintenant vous servira de socle pour le reste du cours, mais également pour vos prochains projets Angular.

5. Installer Angular CLI

Alors, c'est parti, nous allons installer l'outil Angular CLI.

Angular CLI se présente comme un simple paquet Npm, et il peut donc s'installer sur votre poste de travail en une seule ligne de commande. La même commande permet d'installer ou de mettre à jour Angular CLI :

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

L'ajout du paramètre `-g` permet d'installer Angular CLI au niveau global sur votre ordinateur, et pas seulement au niveau d'un projet. En fait, voici le schéma de base de la commande permettant d'installer un paquet avec Npm :

```
npm install <nom_du_paquet>@<version>
```

Rien de très compliqué. Ensuite, lors de l'installation, Angular-CLI devrait installer une nouvelle commande nommée *ng* sur votre poste de travail. C'est cette commande que nous utiliserons pour demander à Angular CLI d'exécuter des tâches pour nous.

Pour vérifier que cette commande est bien installée, fermez votre terminal et ouvrez-en un nouveau. Exécutez ensuite la commande *ng version*.

6. Générer le socle de notre projet

Angular CLI dispose d'une commande permettant de générer le socle de votre application. Sur le site officiel d'Angular CLI, on trouve la commande suivante :

```
ng new ng-pokemons-app --minimal --routing=false --style=css --skipGit
```

Cette commande doit générer une architecture de dossiers, et certains fichiers pour votre nouveau projet, dans un dossier nommé **ng-pokemons-app**.

Pourquoi un tel nom ? Et bien parce que tout le long du cours je vous propose de développer une application qui vous permettent de gérer des Pokémons, ni plus ni moins. (Qui y a vu un lien avec le jeu Pokémon GO ?) Nous partirons d'un dossier vide jusqu'à obtenir une application finale prête pour la production ! Cela vous permettra d'avoir une vision globale du processus de réalisation d'une application Angular.

Vous pouvez nommer votre dossier comme vous le voulez, cela n'a pas beaucoup d'importance pour la suite.

Heu... pourquoi cette commande est aussi complexe, alors que l'on veut simplement démarrer un petit projet ?

Vous avez eu raison de poser la question. En fait, il existe plusieurs options que vous pouvez passer à la commande `ng new`, afin de paramétrer un minimum le socle de l'application qui sera générée. Voici les principales options que j'ai choisi d'appliquer à notre petit projet :

L'option `--minimal`: Cette option permet de générer une version très allégée du socle de notre projet. Cela nous évite de nous retrouver avec une quantité de fichiers de configuration astronomique, comme cela peut être le cas pour un projet plus conséquent.

L'option `--routing=false`: Cette option permet de dire à Angular CLI de ne pas générer de fichier de configuration pour les routes de notre application. Cela nous permettra de voir par nous même comment tout cela fonctionne.

L'option `--style=css`: Cette option permet d'indiquer que nous souhaitons utiliser du code CSS pour le style de notre application. En effet, il est possible d'utiliser un préprocesseur CSS, qui vous permet notamment de déclarer des variables dans du code CSS, afin de vous éviter des duplications dans vos feuilles de style. Mais ce n'est pas le sujet de ce cours, restons sur le CSS.

L'option `--skipGit`: Cette option permet ne pas ajouter le système de versionning Git dans notre projet. Si vous n'en avez jamais entendu parler, sachez que cela nous fait simplement des fichiers de configuration en moins. C'est toujours ça de pris.

Voilà, vous devriez avoir mieux compris le rôle de chacun de ces options.

Avant de continuer, pensez à bien exécuter cette commande :

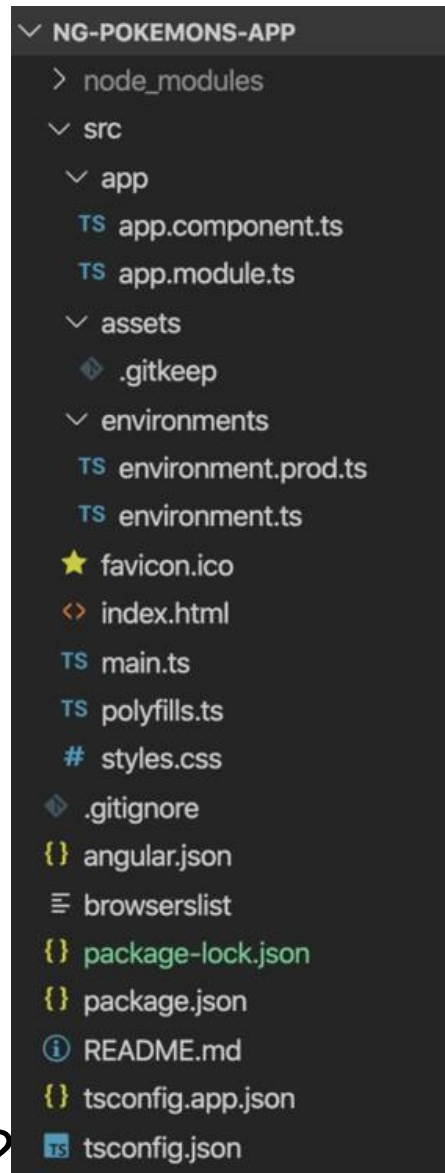
```
ng new ng-pokemons-app --minimal --routing=false --style=css --skipGit
```

La commande peut être un peu longue, car Angular CLI va installer toutes les dépendances dont nous aurons besoin pour notre projet.

7. Présentation du socle généré par Angular CLI

Jusqu'à maintenant, tout ce que nous avons fait avait l'air... « magique ». Exécuter une simple commande, et hop, on obtient un socle prêt pour notre application Angular. Pourtant, il n'y a rien de magique. Je vous invite donc à ouvrir votre éditeur de texte favori, et à ouvrir le socle que Angular-CLI a généré pour nous. Et là, c'est la panique.

Mais... Qu'est-ce que tous ces fichiers font là ? Une application devrait pouvoir fonctionner avec moins de fichiers non,



pourquoi ce superflu ?

Je vous propose de vous expliquer à quoi correspondent tous ces fichiers générés. Une fois que vous aurez compris, vous saurez pourquoi chaque fichier est là dans le projet. Je vous propose donc de voir à quoi correspondent tous ces éléments :

Le dossier `node_modules` : Il s'agit du dossier qui contient toutes les dépendances dont notre application a besoin pour fonctionner. Angular CLI a déjà installé ces dépendances pour nous.

Le dossier `src` : C'est dans ce dossier que se trouve l'essentiel du code source de notre projet. C'est donc ici que vous allez passer la plupart de votre temps en tant que développeur. Tous nos composants Angular, nos templates, nos styles, et tous les autres éléments de notre application se trouveront dans ce dossier. Tous les fichiers en dehors de ce dossier sont destinés à soutenir la création de votre application. Vous retrouvez dans ce dossier plusieurs éléments :

- **Le dossier `app`** : Il s'agit du dossier qui contient le code source de notre application, comme mentionné ci-dessus. On retrouve dans ce dossier notre premier composant Angular, *app.component.ts*, et notre premier module, *app.module.ts*. On aura l'occasion de revenir là-dessus très prochainement.
- **Le dossier `assets`** : C'est le dossier qui est censé contenir les images de votre application. Vous pouvez également ajouter d'autres fichiers dont vous pourrez avoir besoin. Un fichier PDF à télécharger pour vos utilisateurs par exemple.
- **Le dossier `environments`** : Ce dossier contient un fichier de configuration pour chacun de vos environnements de destination : développement sur votre ordinateur local, serveur de production, etc. Nous n'allons pas vraiment avoir besoin de ce dossier pour la suite.
- **Le fichier `browserlist`** : C'est un fichier de configuration utilisé par Angular CLI pour paramétrer certains outils en fonction de tel ou tel navigateur.
- **Le fichier `favicon.ico`** : Il s'agit de la fameuse icône qui s'affiche dans l'onglet de votre navigateur lorsque vous lancez votre application. Par défaut, il s'agit du logo d'Angular.

- **Le fichier `index.html`** : C'est l'unique page HTML de notre application ! C'est elle qui contiendra l'ensemble de notre application.
- **Le fichier `main.ts`** : C'est le point d'entrée principale de notre application. Il s'occupe de compiler notre application, et lance le module racine de celle-ci.
- **Le fichier `polyfill.ts`** : Les différents navigateurs existants n'ont pas le même niveau de prise en charge des normes du Web. Les polyfills servent à lisser ces différences en uniformisant le comportement de votre code entre tous les navigateurs.
- **Le fichier `style.css`** : Le fichier qui contient le style global de votre application. La plupart du temps, vous aurez besoin que vos styles soient attachés à un composant. Cependant, pour une maintenance plus facile, les règles de style qui affectent toutes votre application doivent être centralisées.
- **Le fichier `tsconfig.app.json`** : La configuration du compilateur TypeScript pour notre application Angular.
- **Le fichier `gitignore`** : Les fichiers à ignorer pour le versioning avec Git. Si vous n'utilisez pas Git ou que vous ne savez pas à quoi sert cet outil, vous pouvez supprimer ce fichier. Sinon gardez-le, il est déjà pré-configuré pour le projet.
- **Le fichier `angular.json`** : La configuration d'Angular CLI. Vous pouvez définir plusieurs valeurs par défaut et configurer également les fichiers inclus lors de la création de votre projet. Par exemple, c'est dans ce fichier que se trouve le préfixe que l'on a défini précédemment en ligne de commande.
- **Le fichier `package-lock.json`** : Il s'agit d'un fichier propre au fonctionnement du Node Package Manager, qui a à voir avec les dépendances de vos dépendances. En effet, vous déclarez vos dépendances dans le fichier `package.json`, mais

vos dépendances peuvent elles-mêmes avoir des dépendances qui sont en conflit. Je ne rentre pas plus dans les détails, car normalement, vous n'aurez jamais à modifier ce fichier. Dites-vous simplement que vous n'avez pas à vous en occuper.

- **Le fichier `package.json`** : Le fichier bien connu qui vous permet de déclarer toutes les dépendances de votre projet. Vous pouvez également ajouter vos propres scripts personnalisés dans ce fichier, en plus de ceux d'Angular CLI.
- **Le fichier `README.md`** : Un fichier vous permettant de renseigner des informations pour démarrer et développer votre projet. Si vous êtes motivés, vous pourrez essayer de modifier ce fichier, afin d'indiquer à toute personne récupérant votre projet comment démarrer l'application sur sa machine !
- **Le fichier `tsconfig.json`** : C'est le fichier principal de la configuration de TypeScript, celui que j'ai mentionné au-dessus est simplement une configuration supplémentaire qui étend celle-ci.

Bon, j'espère ne pas vous avoir endormis avec mes explications. Même si vous n'avez pas compris précisément le rôle de chacun des fichiers, retenez simplement que les fichiers sources sont dans le dossier "`src`", et que le reste concerne la configuration de votre application.

C'est grandement suffisant pour commencer avec Angular !

8. Implémenter notre premier composant Angular

Angular CLI a créé notre premier composant Angular, il est temps de voir à quoi cela ressemble concrètement, depuis le temps que vous en entendez parler !

Rendez-vous donc dans le fichier **app.component.ts** de votre projet.

Nous allons modifier un peu le code de ce composant, car il a été généré par Angular CLI et il y a pas mal de code inutile pour nous. Remplacez le code existant du fichier *app.component.ts* par ceci:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   template: `<h1>Hello, {{name}} !</h1>`,
6 })
7 export class AppComponent {
8   name = 'Angular';
9 }
```

Nous allons prendre un peu de temps pour décrire ce fichier, car malgré sa taille réduite, ce code est composé de trois parties différentes.

D'abord, on importe les éléments dont on va avoir besoin dans notre fichier. A la ligne 1, on importe l'annotation `Component` depuis le cœur de Angular : `@angular/core`. Retenez donc simplement la syntaxe suivante :

```
import { UnElement } from { quelque-part };
```

Un composant doit au minimum importer l'annotation *Component*, bien sûr.

Ensuite, de la ligne 3 à la ligne 6, on aperçoit l'annotation `@Component` qui nous permet de définir un composant. L'annotation

d'un composant doit au minimum comprendre deux éléments : *selector* et *template*.

- *Selector* permet de donner un nom à votre composant afin de l'identifier par la suite. Par exemple, notre composant se nomme ici *pokemon-app*, ce qui signifie que dans notre page web, c'est la balise qui sera insérée. Et ce code sera parfaitement valide. Vous vous rappelez du chapitre sur les Web Components ?
- Quant à l'instruction *template*, elle permet de définir le code HTML du component (On peut bien sûr définir notre template dans un fichier séparé avec l'instruction *templateUrl* à la place de *template*, afin d'avoir un code plus découpé et plus lisible). Ici, vous pouvez deviner que la syntaxe avec les doubles accolades permet d'afficher la variable déclarée à la ligne 7. Il s'agit de *l'interpolation*, que nous verrons dans le chapitre sur les templates. En attendant, retenez que ce template affiche « *Hello, Angular* » pour l'utilisateur.

Enfin, à la ligne 7, on retrouve le code de la classe de notre composant. C'est cette classe qui contiendra la **logique** de notre composant. Le mot-clef *export* permet de rendre le composant accessible pour d'autres fichiers.

Par convention, on suffixe le nom des composants par Component : la classe du composant app est donc AppComponent.

9. Présentation du module racine

Pour l'instant nous n'avons qu'un unique composant dans notre application, mais imaginez que notre application soit composée de 100 pages, avec 100 composants différents, comment ferions-nous pour nous y retrouver ?

L'idéal serait de pouvoir regrouper nos composants par fonctionnalité : par exemple regrouper tous les composants qui servent à l'authentification entre eux, tous ceux qui servent à construire un blog entre eux, etc.

Eh bien, Angular permet cela, grâce aux **modules** ! Tous nos composants seront regroupés au sein de modules.

Mais du coup ? ... Il nous faut au moins un module pour notre composant, non ?

Bravo ! Vous avez tout compris !

Au minimum, votre application doit contenir un module : le **module racine**. Au fur et à mesure que votre application se développera, vous ajouterez d'autres modules pour couvrir de nouvelles fonctionnalités.

Le module racine de notre application est *app.module.ts*, et à déjà été généré par Angular CLI :

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5
6 @NgModule({
7   declarations: [
8     AppComponent
```

```
9   ],
10  imports: [
11    BrowserModule
12  ],
13  providers: [],
14  bootstrap: [AppComponent]
15  })
16  export class AppModule { }
```

Je vais présenter ce code rapidement. D'abord, on retrouve des importations en haut du fichier. On importe en premier le *BrowserModule* à la ligne 1, qui est un module qui fournit des éléments essentiels pour le fonctionnement de l'application, comme les directive *ngIf* et *ngFor* dans tous nos templates, nous reviendrons dessus plus tard.

Ensuite, on retrouve l'annotation *NgModule* contenue dans le cœur d'Angular lui-même, à la ligne 2. Enfin, à la ligne 4, on importe le seul composant de notre application *AppComponent*, que nous allons rattacher au module racine juste après.

Mais le plus important ici est l'annotation *NgModule*, car c'est elle qui permet de déclarer un module :

- **declarations** : Une liste de tous les composants et directives qui appartiennent à ce module. Nous avons donc ajouté notre unique composant *AppComponent*.
- **imports** : Permet de déclarer tous les éléments que l'on a besoin d'importer dans notre module. Les modules racines ont besoin d'importer le *BrowserModule* (contrairement aux autres modules que nous ajouterons par la suite dans notre application).
- **bootstrap** : Permet d'identifier le composant racine, qu'Angular appelle au démarrage de l'application. Comme le module racine est lancé automatiquement par Angular au démarrage de l'application, et qu'*AppComponent* est le

composant racine du module racine, c'est donc *AppComponent* qui apparaîtra au démarrage de l'application. Ça va, rien de compliqué si on prend le temps de bien comprendre.

Lors de ce cours, nous commencerons à travailler avec une application mono-module, puis nous ajouterons d'autres modules pour que vous voyez comment se passe le développement d'une application plus complexe.

10. Démarrer l'application

Démarrer l'application va être un jeu d'enfant. Il y a déjà une commande disponible pour nous permettre de démarrer l'application. Ouvrez donc un terminal qui pointe vers le dossier de votre projet et taper la commande suivante :

```
ng serve --o
```

Vous devriez voir des choses qui s'affichent dans la console, puis après un délai de quelques secondes, votre navigateur s'ouvre tout seul, et vous voyez affiché le message « *Hello, Angular !* » dans votre navigateur :

Hello, Angular !

Ça y est, nous y sommes arrivés !

Pour couper la commande `npm start`, appuyer sur `CTRL + C`. En effet cette commande tourne en continu puisque qu'elle s'occupe de démarrer le serveur qui s'occupe d'envoyer l'application au navigateur !

11. Conclusion

Voilà, vous avez réalisé votre premier « *Hello, World !* » avec Angular ! Vous pouvez être fier de vous.

Mine de rien, ce modeste exemple a nécessité d'utiliser les Web Components, ES6 et TypeScript ! C'est pourquoi il n'était pas inutile de les étudier juste avant.

Je vous propose dans le prochain chapitre de commencer à construire notre application de Pokémons, par-dessus la base que nous venons de réaliser dans ce chapitre. Mais rassurez-vous, il y a encore beaucoup de choses à voir !

En résumé

- Angular CLI est l'outil par défaut choisie par Angular pour démarrer une nouvelle application.
- On a besoin au minimum d'un module racine et d'un composant racine par application.
- Le module racine se nomme par convention *AppModule*.
- Le composant racine se nomme par convention *AppComponent*.
- Angular CLI est fourni avec des commandes prêtes à l'emploi comme la commande *ng serve*, qui nous permet de démarrer notre application web dans le navigateur.

Partie 2

Acquérir les bases sur Angular

Chapitre 6 : Les Composants

Dans la section précédente, nous avons eu un aperçu du socle d'une application Angular, et de son écosystème. Il est temps maintenant de voir plus en profondeur le fonctionnement des composants.

Je vous propose de mettre en place un simple composant qui affiche une liste de Pokémons. Nous allons continuer à travailler à partir du socle mis en place lors du chapitre précédent, et vous allez voir que l'on peut déjà faire pas mal de choses.

Pour le moment, ne touchez pas au code de notre projet. Les débuts de chapitres sont généralement consacrés à la théorie, et je vous préviendrai quand on passe en mode pratique.

1. Qu'est-ce qu'un composant ?

Un composant est une classe, qui contrôle une portion de l'écran. Cette *portion de l'écran* contrôlée par le composant, on l'appelle une *vue* : une vue peut être l'ensemble de la page web, une fenêtre de tchat, une barre de navigation, etc... Cette vue est définie dans le *template* du composant.

On peut dire que : 1 composant = 1 classe + 1 vue.

On définit la logique d'un composant dans une classe ; c'est-à-dire ce qu'il faut pour faire fonctionner la vue. La classe d'un composant interagit avec la vue à travers des *propriétés* et des *méthodes* que nous allons voir.

Par exemple, imaginons un composant qui affiche tous les pokémons présents dans notre application, sous forme de liste. Ce composant aura donc une propriété *pokémons* contenant tous les pokémons à afficher :

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    template: `<h1>Hello, Angular 2 !</h1>`,
6  })
7  export class AppComponent implements OnInit {
8    // 1. La propriété pokémons est un tableau de Pokemons.
9    private pokemons: Pokemon[];
10
11    // 2. Initialisation de la propriété pokémons.
12    ngOnInit() {
13      this.pokemons = [
14        { name: 'Bulbizarre', hp: 6 },
15        { name: 'Salamèche', hp: 6 },
16        { name: 'Carapuce', hp: 6 }
17      ];
18    }
19  }
```

Il s'agit d'un composant un peu plus complexe que le composant minimum que l'on avait vu au chapitre précédent, pour deux raisons :

- **L'ajout d'une propriété *pokemons*** : à la ligne 9, j'ai déclaré une propriété *pokemons*, et j'utilise le typage TypeScript pour indiquer que cette propriété devra contenir un tableau d'objet de type *Pokemon*. Cette propriété est privée et son usage est donc interne au composant *AppComponent*.
- **J'initialise la propriété *pokemons*** : à la ligne 13, dans une méthode nommée *ngOnInit*, j'initialise la propriété avec trois Pokémons, qui ont pour propriétés *name* et *hp*. ('hp' désigne le nombre de point de vie du Pokémon : *Health Point*).

D'accord pour la propriété pokemons, mais c'est quoi cette méthode ngOnInit, et ce nouvel import OnInit, et pourquoi tu implémentes cette fonction ?

Alors je ne vous ai pas tout dit. En réalité, *ngOnInit* n'est pas une simple méthode mais une méthode de cycle de vie du composant. Nous allons voir tout de suite de quoi il s'agit.

2. Les cycles de vie d'un composant

Chaque composant a un cycle de vie géré par Angular lui-même. Angular crée le composant, s'occupe de l'afficher, crée et affiche ses composants fils, vérifie quand les données des propriétés changent, et détruit les composants, avant de les retirer du DOM quand cela est nécessaire. Pratique, non ?

Angular nous offre la possibilité d'agir sur ces moments clefs quand ils se produisent, en implémentant une ou plusieurs interfaces, pour chacun des événements disponibles. Ces interfaces sont disponibles dans la librairie *core* d'Angular.

Chaque interface permettant d'interagir sur le cycle de vie d'un composant fournit une et une seule méthode, dont le nom est le nom de l'interface préfixé par *ng*. Par exemple, l'interface *OnInit* fournit la méthode *ngOnInit*, et permet de définir un comportement lorsque le composant est initialisé.

Voici ci-dessous la liste des méthodes disponibles pour interagir avec le cycle de vie d'un composant, dans l'ordre chronologique du moment où elles sont appelées par Angular.

Méthode	Comportement
ngOnChanges	C'est la méthode appelée en premier lors de la création d'un composant, avant même <i>ngOnInit</i> , et à chaque fois que Angular détecte que les valeurs d'une propriété du composant sont modifiées. La méthode reçoit en paramètre un objet représentant les valeurs actuelles et les valeurs

	précédentes disponibles pour ce composant.
ngOnInit	Cette méthode est appelée juste après le premier appel à <i>ngOnChanges</i> , et elle initialise le composant après que Angular ait initialisé les propriétés du composant.
ngDoCheck	On peut implémenter cette interface pour étendre le comportement par défaut de la méthode <i>ngOnChanges</i> , afin de pouvoir détecter et agir sur des changements qu'Angular ne peut pas détecter par lui-même.
ngAfterViewInit	Cette méthode est appelée juste après la mise en place de la vue d'un composant (et des vues de ses composants fils s'il en a).
ngOnDestroy	Appelée en dernier, cette méthode est appelée avant qu'Angular ne détruise et ne retire du DOM le composant. Cela peut se produire lorsqu'un utilisateur navigue d'un composant à un autre par exemple. Afin d'éviter les fuites de mémoire, c'est dans cette méthode que nous effectuerons un certain nombre d'opérations afin de laisser l'application « propre » (nous détacherons les

gestionnaires d'événements par exemple).
--

Même si vous ne définissez pas explicitement des méthodes de cycle de vie dans votre composant, sachez que ces méthodes sont appelées en interne par Angular pour chaque composant. Lorsqu'on utilise ces méthodes, on vient donc juste surcharger tel ou tel événement du cycle de vie d'un composant.

Les méthodes que vous utiliserez le plus seront certainement *ngOnInit* et *ngOnDestroy*, qui permettent d'initialiser un composant, et de le nettoyer proprement par la suite lorsqu'il est détruit.

2.1. Interagir sur le cycle de vie d'un composant

Nous allons interagir sur le cycle de vie d'un composant, et en particulier sur l'étape d'initialisation du composant.

Sachez que la méthode que nous allons voir est valable quelle que soit la méthode de cycle de vie.

La première chose à faire est d'importer l'interface que nous allons utiliser. Dans notre cas, il s'agit d'*OnInit* :

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

Ensuite, il faut implémenter cette interface pour le composant :

```
export class AppComponent implements OnInit {}
```

Enfin nous devons ajouter la méthode *ngOnInit* fournie par cette interface, dans notre composant :

```
export class AppComponent implements OnInit {  
  
  ngOnInit() {  
    // ce message s'affichera dans la console de votre navigateur  
    // dès que le composant est initialisé !  
    console.log('ngOnInit !');  
  }  
}
```

Et voilà, vous êtes maintenant capable d'ajouter une étape d'initialisation à vos composants ! Nous utiliserons beaucoup cette méthode pour initialiser les propriétés de nos composants.

Pourquoi est-ce qu'on s'embête à charger les données dans ngOnInit, alors que l'on pourrait le faire dans le constructeur du composant ?

Surtout pas ! C'est fortement déconseillé !

Il est chaudement recommandé de garder toute la logique de votre application en dehors du constructeur, surtout quand vous commencerez à récupérer les données d'un serveur distant. Le corps du constructeur peut éventuellement être utilisé pour les initialisations simples, et encore, nous allons voir tout de suite une autre manière de le faire.

Heu, j'initialise quoi du coup ? Et où ?

Je vais vous donner un exemple de code avec lequel vous devriez comprendre : vous avez un composant qui possède une propriété *titre*, une propriété *sous-titre* et une liste de tâches à afficher.

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

@Component({
  selector: 'task-component',
  template: '<h1>{{ title }} <em>{{ subtitle }}</em><h1>'
})
export class TaskListComponent implements OnInit {
  // 1. Je choisis de définir mon titre immédiatement lors de
  // la déclaration de ma propriété. C'est plus concis.
  title: string = "Liste de tâches";
  // 2. Je choisis d'assigner une valeur à cette propriété dans
  // le constructeur de mon composant, plutôt que directement.
  // J'ai le choix dans ce cas.
  subtitle: string;
  // 3. Je récupère les tâches à réaliser depuis un serveur distant.
  tasks: Task[];

  constructor() {
    // J'assigne une valeur à ma propriété. Il ne s'agit pas de
    // code dit 'complexe', je peux le faire dans le constructeur.
  }
}
```

```
this.subtitle = "Tâches urgentes";  
}  
  
ngOnInit() {  
  // La récupération de données depuis un serveur distant  
  // est une opération complexe  
  // (appel asynchrone, gestion des erreurs...)  
  // Ce genre d'initialisation a sa place dans la méthode ngOnInit.  
  this.tasks = ...;  
}  
}
```

Comme vous pouvez le voir, on peut choisir d'assigner des valeurs simples soit :

- Au moment où vous déclarez une propriété à votre composant.
- Soit plus tard dans le constructeur.

Vous avez le choix et cela dépend de la manière dont vous préférez vous organiser. La première méthode est plus compacte et évite d'ajouter des lignes de code supplémentaires à votre composant. De plus, cela nous permet d'avoir les initialisations des propriétés au même endroit. Dans la suite du cours c'est l'approche que je retiendrai.

En revanche les opérations dites « *complexes* », qui correspondent à des opérations propres à votre application, ne doivent pas se trouver dans le constructeur. Si vous avez compris ça, vos composants seront « *propres* » et ce sera un plaisir pour d'autres développeurs de relire votre code.

3. Une méthode pour gérer les interactions utilisateurs

Vous vous demandez sûrement comment gérer les interactions de l'utilisateur. Par exemple, si l'utilisateur clique sur un Pokémon de notre liste, nous voudrions probablement effectuer une action comme une redirection ou un affichage.

La capture de l'interaction utilisateur se fait côté template, et nous verrons comment faire au prochain chapitre. En revanche, lorsque l'utilisateur déclenche l'événement attendu, vous pouvez définir une méthode qui sera appelée à chaque fois.

```
export class AppComponent {  
  // ...  
  selectPokemon(pokemon: Pokemon) {  
    console.log("Vous avez cliqué sur " + pokemon.name);  
  }  
  // ...  
}
```

Comme vous le voyez, nous n'avons rien fait de compliqué ici. La méthode *selectPokemon* prend en paramètre un objet de type *Pokemon* et affiche son nom dans la console du navigateur.

Il ne nous reste plus qu'à développer un template pour ce composant, qui appellera notre méthode lorsque l'événement *'clic'* sera déclenché par l'utilisateur. Mais nous verrons comment faire cela depuis notre template, au chapitre suivant. Patience !

4. Un peu de pratique !

Bon passons à la pratique, et essayons d'enrichir quelque peu le composant *app.component.ts* de notre application. Je propose d'ajouter deux fonctionnalités que nous venons de voir :

- Injecter une liste de Pokémons dans le composant.
- Préparer une méthode qui devra gérer l'action lorsque l'utilisateur cliquera sur un Pokémon de cette liste.

Mais avant de vous laisser chercher tout seul, je me dois de vous donner deux choses :

- Une classe TypeScript permettant de modéliser un Pokémon.
- Un fichier de données contenant quelques Pokémons à injecter dans notre composant, à titre d'exemple.

Le modèle

Créer un fichier *pokemon.ts* dans le dossier *app* de notre projet, avec le code suivant :

```
export class Pokemon {  
  id: number;  
  hp: number;  
  cp: number;  
  name: string;  
  picture: string;  
  types: Array<string>;  
  created: Date;  
}
```

Les données

Maintenant, créons un fichier *mock-pokemons.ts* qui contiendra les données de plusieurs Pokémons. Ce fichier doit être placé dans le dossier *src/app* (le fichier est un peu long, mais il s'agit simplement de données mises à disposition pour notre application). Etant donné que nous n'allons pas modifier ce fichier dans la suite du cours, je vous propose de le récupérer directement depuis la page des ressources du cours.

Comme vous pourrez le constater, ce fichier ne fait qu'exporter la constante *POKEMONS*, qui contient des données nécessaires pour notre application.

Bon, maintenant vous avez tous les éléments nécessaires pour améliorer le composant *app.component.ts* de notre application. Pour rappel, voici ce que vous devriez pouvoir faire :

- Injecter une liste de Pokémons dans le composant.
- Préparer une méthode qui devra gérer l'action lorsque l'utilisateur cliquera sur un Pokémon de cette liste.

A vous de jouer maintenant !

5. Correction

Alors vous avez réussi ? Vous n'avez plus d'erreurs dans la console ?

Je vous propose tout de même une correction, afin que nous ayons la même base pour la suite. Recopiez donc le code ci-dessous, puis analysons l'ensemble :

```
1  import { Component, OnInit } from '@angular/core';
2  import { Pokemon } from './pokemon';
3  import { POKEMONS } from './mock-pokemons';
4
5  @Component({
6    selector: 'app-root',
7    template: '<h1>Pokémons</h1>',
8  })
9  export class AppComponent implements OnInit {
10    pokemons: Pokemon[] = null;
11    ngOnInit() {
12      this.pokemons = POKEMONS;
13    }
14    selectPokemon(pokemon: Pokemon) {
15      console.log('Vous avez sélectionné ' + pokemon.name);
16    }
17  }
```

Voici ce qui a été modifié, par rapport au composant qui affichait simplement « *Hello, World !* » :

- **Trois importations supplémentaires**, dans les trois premières lignes : importation de l'interface *OnInit*, récupération du modèle représentant un Pokémon, afin de pouvoir typer nos variables, et ajout de la constante *POKEMONS*, qui contient les données de différents Pokémon.
- **Modification du template** à la ligne 7, en mettant Pokémons comme titre, à la place de « *Hello, World !* ».

- **Déclaration d'une propriété** à la ligne 10 : *pokemons* qui doit contenir la liste des Pokémons à afficher à l'utilisateur.
- **La méthode d'initialisation *ngOnInit*** à la ligne 11, qui a pour fonction d'initialiser la propriété *pokemons* du composant avec les valeurs importées depuis *mock-pokemons.ts*.
- **La méthode *selectedPokemon*** à la ligne 14, qui permet d'afficher dans la console le nom du Pokémon sélectionné par l'utilisateur.

Nous voici avec un composant presque prêt, mais il lui manque encore une chose pour pouvoir fonctionner. C'est pourquoi nous ajouterons un template à notre composant dans le prochain chapitre !

Pour l'instant, ce composant ne fait rien d'autre que d'afficher un titre, nous devons le coupler à un template pour avoir un ensemble fonctionnel !

6. Conclusion

Nous savons désormais comment mettre en place un composant de base, intervenir sur les différentes étapes de son cycle de vie et l'initialiser avec des données. Nous avons vu comment développer la classe d'un composant.

Il ne nous manque plus qu'à associer un template à notre composant, car pour l'instant notre application se contente d'afficher un simple titre. Je vous propose de voir tout cela au chapitre suivant, qui sera peut-être un peu plus long que celui-ci !

En résumé

- Un composant fonctionne en associant la logique d'une classe TypeScript avec un template HTML.
- On utilise l'annotation *@Component* pour indiquer à Angular qu'une classe est un composant.
- On peut initialiser une propriété avec une valeur simple directement lors de sa déclaration ou dans le constructeur d'un composant.
- Des méthodes nous permettent d'interagir avec le cycle de vie d'un composant. Ces méthodes sont toutes préfixées par *ng*.
- La méthode *ngOnInit* permet de définir un comportement spécifique lors de l'initialisation d'un composant.
- Les méthodes de cycle de vie d'un composant que nous utiliserons le plus sont *ngOnInit* et *ngOnDestroy*.

Chapitre 7 : Les Templates

Nous allons voir un chapitre très important, celui des *templates*. Pour rappel, les templates sont les vues de nos composants, et ils contiennent le code de notre interface utilisateur. Il est donc important de bien comprendre ce chapitre pour pouvoir développer une expérience utilisateur de qualité sur votre site !

Je préfère vous prévenir, ce chapitre peut sembler assez long. Mais rassurez-vous, nous allons surtout voir une nouvelle syntaxe : la **syntaxe des templates** avec Angular.

1. Développer des templates

Pour l'instant, nous écrivons nos templates sous forme de chaîne de caractères. Mais parfois cette chaîne de caractères peut devenir trop longue pour tenir sur une seule ligne.

Imaginons que vous ayez un template très long, voilà ce que vous seriez obligé de faire :

```
@Component({  
  selector: 'long-template',  
  template: 'Ceci est un template très,'  
    + 'très,'  
    + 'très,'  
    + 'long !'  
})
```

Ce n'est vraiment pas pratique ! Je vous rassure, Angular ne nous oblige pas à faire ça. En fait nous avons même deux possibilités pour gérer les templates trop long. La première possibilité consiste à utiliser directement ES6 et la deuxième est de profiter d'une autre option que propose Angular.

Utiliser ES6

En fait, un template peut s'écrire sur plusieurs lignes, à condition d'utiliser le *backtick* : ` .

Il ne faut pas confondre le backtick ` avec la quote ' !

Ce symbole permet de déclarer une chaîne de caractères sur plusieurs lignes sans avoir à concaténer des chaînes de caractères ! Nous pouvons donc réécrire notre composant précédent de la manière suivante :

```
@Component({  
  selector: 'long-template',  
  template: `  
    Ceci est un template de plusieurs lignes  
    écrit sans problème avec le backtick et ES6,  
    nous évitant ainsi des concaténations pénibles.  
  `,  
})
```

Cette fonctionnalité est disponible avec ES6, mais pas avec ES5, d'où l'intérêt de l'utiliser.

Utiliser Angular

Bien entendu, parfois nos templates sont trop longs pour être écrits dans le même fichier que notre composant, et nous préférons décrire notre template dans un fichier à part. Angular propose l'option *templateUrl* pour cela :

```
@Component({  
  selector: 'long-template',  
  templateUrl: './templates/long-template.component.html'  
})
```

Grâce à cette option, vous devez simplement renseigner le chemin relatif vers votre template, par rapport à la racine de votre projet. Dans l'exemple ci-dessus, on suppose que le template se trouve dans le dossier *app* de notre projet.

Maintenant imaginons que vous souhaitez regrouper tous vos templates dans un dossier *templates*. Dans ce cas, vous devrez modifier le chemin relatif :

```
@Component({  
  selector: 'long-template',  
  templateUrl: './templates/long-template.component.html'  
})
```

2. L'interpolation

La manière la plus simple d'afficher la propriété d'un composant dans son template est d'utiliser le mécanisme de l'interpolation. Avec l'interpolation, nous pouvons afficher la valeur d'une propriété dans la vue d'un template, entre deux accolades, comme ceci : `{{ maPropriete }}`.

Par exemple, si vous avez une propriété *title* dans votre composant, et que vous souhaitez afficher sa valeur dans le template, comment faites-vous ? Et bien avec l'interpolation. Examinez l'exemple ci-dessous :

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

@Component({
  selector: 'title',
  // La syntaxe pour interpoler une propriété
  template: '<h1>{{ title }}</h1>'
})
export class TitleComponent {
  // La propriété interpolé
  title: string = "L'interpolation, c'est simple !";
}
```

Angular récupère automatiquement la valeur de la propriété *title* du composant, et l'injecte dans le template, pour qu'elle s'affiche dans le navigateur de l'utilisateur. Et ce n'est pas tout, Angular met à jour l'affichage dans le template si la propriété du composant est modifiée, et le tout sans que nous n'ayons rien besoin de faire !

Le fait que la propriété interpolée soit mise à jour automatiquement dans le template en cas de modification s'appelle le binding unidirectionnel. L'interpolation s'occupe de cela, en plus du simple affichage de la valeur de la propriété.

3. Syntaxe de liaison des données

L'interpolation est très pratique pour lier notre template et notre composant. Cependant, il existe d'autres manières de créer des liaisons entre les deux. L'interpolation a une particularité : elle pousse les données depuis le composant vers le template, mais pas dans le sens inverse.

Nous allons voir comment gérer les autres types de liaison.

Du composant vers le template

Nous pouvons pousser plusieurs données depuis le composant vers le template. Voici les liaisons que nous pouvons mettre en place :

Propriétés	Code	Explications
Propriété d'élément	<code></code>	On utilise les crochets pour lier directement la source de l'image à la propriété du composant <i>someImageUrl</i> .
Propriété d'attribut	<code><label [attr.for]="someLabelId ">...</label></code>	On lie l'attribut <i>for</i> de l'élément <i>label</i> avec la propriété de notre composant <i>someLabelId</i> .
Propriété de la classe	<code><div [class.special]="isSpecial ">Special</div></code>	Fonctionnement similaire, pour attribuer ou non la classe <i>special</i> à l'élément <i>div</i> .
Propriété de style	<code><button [style.color]="isSpecial?'red':'green' ">Special</div></code>	On peut également définir un style pour nos éléments de manière

		<p>dynamique : ici on définit la couleur de notre bouton en fonction de la propriété <i>isSpecial</i>, soit rouge, soit vert. (C'est un opérateur ternaire que l'on utilise comme expression).</p>
--	--	--

Du template vers le composant

Maintenant, si on souhaite lier des données dans l'autre sens, comment fait-on ? Voyons tout de suite comment gérer les interactions de l'utilisateur.

Nous verrons les liaisons bidirectionnelles dans le chapitre dédié sur les Formulaires.

4. Gérer les interactions de l'utilisateur

Quand un utilisateur clique sur un lien, presse un bouton ou entre un texte, on veut en être informé. Toutes ces actions lèvent des événements dans le DOM, avec lequel nous voudrions interagir.

Le DOM est une représentation structurée de votre page HTML, où chaque balise HTML représente un nœud.

Nous allons apprendre à lier n'importe quel événement du DOM à une méthode de composants, en utilisant la syntaxe de liaison d'événements d'Angular.

Créer une liaison avec les événements

La syntaxe pour *écouter* un événement est simple. On entoure le nom de l'événement du DOM par des parenthèses et on renseigne ensuite une action à réaliser, qui correspond à une méthode de notre composant. Par exemple, pour écouter un *clic* sur un bouton, rien de plus simple :

```
<button (click)="onClick()">Cliquez ici !</button>
```

Ici (*click*) correspond à l'événement que l'on souhaite intercepter : ici, un événement de type *clic*. La méthode *onClick* du composant associé est alors appelée, à chaque clic de la part de l'utilisateur.

Récupérer les valeurs entrées par l'utilisateur

Imaginons que l'on veuille récupérer une valeur entrée par un utilisateur, par exemple un nom qu'il rentre dans un champ texte, puis afficher cette valeur à l'écran, avec un message du type « *Bonjour, Jean* » ou « *Bonjour, Juliette* ».

Pour le moment, nous ne savons pas comment récupérer cette valeur.

Eh bien, rassurez-vous, Angular met à disposition un objet représentant un événement du DOM à travers la variable `$event`. Regardons tout de suite comment ça marche :

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

@Component({
  selector: 'evenement',
  template: `
    <input (keyup)="onKey($event)">
    <p>{{ values }}</p>`
})
export class EventComponent {
  values: string = "";

  onKey(event: any) {
    // texte entré par l'utilisateur
    this.values = "Bonjour " + event.target.value;
  }
}
```

Regardons les lignes importantes de ce composant :

- **A la ligne 6** : dans le template, à chaque fois que l'utilisateur déclenche l'événement `keyup`, c'est-à-dire qu'il tape du texte dans le champ texte `<input>`, alors la méthode `onKey($event)` est appelée, avec en paramètre un objet représentant l'événement soulevé.
- **A la ligne 15** : Notre méthode traite l'événement soulevé par l'utilisateur, en mettant à jour la propriété `value` de notre

composant. Ces modifications sont ensuite répercutées directement sur le template.

Le mot-clé `any` est un type propre à TypeScript, qui signifie simplement « n'importe quel type ».

La structure de l'objet `$event` est déterminée à l'avance, quel que soit l'événement soulevé, car cet objet représente un événement standard du DOM. Il contient des propriétés et des méthodes communes à tout événement. Ainsi, `$event.target` renvoie un objet de type `HTMLInputElement` (un champ texte), qui possède une propriété `value`, qui contient les données entrées par l'utilisateur. On peut donc bien accéder aux données entrées par l'utilisateur depuis nos composants.

Cependant, pour l'instant notre code est minimaliste, nous n'avons typé aucune variable de notre composant, afin d'en simplifier la lecture. Dans la réalité nous préférons avoir un typage fort, c'est-à-dire un typage le plus précis possible :

```
export class EventComponent {  
  values: string = "";  
  // avec typage fort  
  onKey(event: KeyboardEvent) {  
    this.values = 'Bonjour' + (<HTMLInputElement>event.target).value;  
  }  
}
```

Le problème, c'est que maintenant on se rend compte que notre code prend en compte les détails de l'objet `$event`, alors que cela ne concerne pas directement notre composant. Le code n'est pas très lisible et nous devons gérer des objets qui ne concernent pas directement notre application !

Heureusement, nous allons voir une méthode pour résoudre ce problème.

Et mais on n'a vu tout ça pour rien ?

Non, maintenant vous aurez vu comment cela fonctionne *de l'intérieur* et vous êtes capable d'interagir avec n'importe quel événement du DOM depuis votre composant !

5. Les variables référencées dans le template

Il existe une autre manière de récupérer les données entrées par les utilisateurs, sans la variable `$event`. Angular propose une fonctionnalité permettant de déclarer des variables locales dans le template. Ces variables nous garantissent un accès sur l'élément du DOM depuis le template.

On déclare une variable locale avec l'opérateur `#` :

```
@Component({
  selector: 'loop-back',
  template: `
    <input #box (keyup)="0">
    <p>{{box.value}}</p>
  `
})
export class LoopbackComponent { }
```

A la ligne 4, nous avons déclaré une variable `box` référence dans le template. La variable `box` est une référence à l'élément `<input>` sur lequel elle a été déclarée. Nous pouvons ensuite récupérer la valeur entrée par l'utilisateur grâce à la propriété `value`, et l'afficher via à l'interpolation à la ligne 5. L'interpolation sert à afficher la variable référencée dans le template.

Les variables référencées dans le template sont accessibles pour tous les éléments fils et frères de l'élément sur lequel elles ont été déclarées.

Heu, à quoi sert le `(keyup)="0"` à la ligne 4 ?

Ce code est important car Angular prend en compte la liaison de données entre le composant et le template seulement si on effectue une action en réponse à des événements asynchrones, comme des frappes sur le clavier.

Et pourquoi zéro ? Et bien le zéro est la plus courte déclaration pour signifier que nous ne voulons rien exécuter, mais simplement activer la liaison de données entre la variable locale et la valeur interpolée !

Les variables référencées dans le template peuvent être intrigantes au départ, mais elles permettent d'écrire un code plus lisible. Reprenons notre exemple précédent avec des variables référencées dans les templates :

```
@Component({
  selector: 'loop-back',
  template: `
    <input #box (keyup)="onKey(box.value)">
    <p>{{ values }}</p>
  `
})
export class LoopbackComponent {
  values = "";
  onKey(value: string) {
    this.values = 'Bonjour ' + value;
  }
}
```

C'est quand même plus simple que de manipuler l'objet *\$event* non ?

Je vous propose de voir ensemble deux autres événements qui pourront nous être utiles :

- Détecter lorsque l'utilisateur appuie sur *Entrée*.
- Lorsqu'il perd le *focus* sur un champ texte.

Détecter l'appui sur la touche Entrée

Imaginez que vous souhaitez récupérer les données qu'un utilisateur a saisies, mais seulement lorsqu'il appuie sur *Entrée*, et non à chaque fois qu'il tape une lettre !

Angular nous permet de filtrer les événements du clavier à travers une syntaxe spécifique. Nous pouvons écouter uniquement la touche *Entrée* en utilisant le pseudo-événement *keyup.enter* :

```
@Component({
  selector: 'enter',
  template: `
    <input #box (keyup.enter)="values=box.value">
    <p>{{values}}</p>
  `
})
export class EnterComponent {
  values = "";
}
```

Dans cet exemple, c'est seulement au moment où l'utilisateur appuiera sur la touche *Entrée* que la valeur de la propriété *values* sera prise en compte.

Il aurait été préférable de réaliser l'affectation *values=box.value* dans le composant plutôt que directement dans le template, ce qui est recommandé par la documentation officielle, mais je souhaitais vous montrer que c'est possible.

Détecter la perte du focus sur un élément

Nous pourrions améliorer l'exemple précédent en détectant si l'utilisateur clique ailleurs sur la page après avoir entré une donnée dans un champ texte.

En effet, si l'utilisateur sélectionne un autre élément de la page, alors qu'il était en train de remplir un champ texte, cela va provoquer une perte du focus sur le champ texte courant. Cette perte du focus correspond à un événement nommé *blur*, et il permet d'écouter la perte du focus sur un élément. Nous aimerions bien récupérer les données que l'utilisateur a tapées, et dont il a interrompu la saisie en cliquant ailleurs sur la page :

```
@Component({
  selector: 'enter',
  template: `
    <input #box
      (keyup.enter)="values=box.value"
      (blur)="values=box.value">
    <p>{{values}}</p>
  `
})
export class EnterComponent {
  ...
}
```

Comme vous le voyez, rien de compliqué, on a combiné l'événement *blur* avec la détection de la touche *Entrée* précédente, et ces deux événements déclenchent la même action : *values=box.value*.

6. Les directives *NgIf* et *NgFor*

Nous allons voir deux directives dont nous allons avoir souvent besoin dans nos templates : *NgIf* et *NgFor*.

Le chapitre prochain est dédié aux directives, nous pourrons voir de quoi il s'agit plus exactement.

Ces directives sont disponibles automatiquement dans tous les templates de notre application, car elles sont ajoutées par le *BrowserModule* au démarrage de l'application. Nous n'avons donc pas besoin de les importer dans notre composant pour pouvoir les utiliser !

Conditionner un affichage

Parfois on a besoin d'afficher une portion de template seulement dans des circonstances spécifiques. On a besoin de tester une condition pour décider d'afficher un élément ou non.

La directive `ngIf` permet d'insérer ou de retirer un élément à partir d'une condition qui peut être vraie ou fausse.

Essayons d'afficher un message dans notre template, seulement si un utilisateur est majeur :

```
<p *ngIf="utilisateur.age > 18">  
Ce message est top secret et vous ne pouvez le voir que si vous avez plus de 18 ans.  
</p>
```

L'expression entre les doubles guillemets est explicite : l'utilisateur ne verra ce message que s'il a plus de 18 ans (ce qui implique qu'il y a une propriété *utilisateur* dans le composant associé à ce template). Selon le résultat de la condition, Angular ajoute le paragraphe au DOM et le message apparaît, ou le message n'est pas affiché.

Heu... ? Pourquoi nous n'avons pas utilisé l'interpolation ? Comment Angular sait que 'utilisateur' est lié à une propriété d'un composant ?

La réponse est simple : `ngIf`. Cette directive vous permet d'appeler les propriétés de vos composants directement, sans utiliser la syntaxe avec les doubles accolades, lors de la définition de votre condition.

Avez-vous remarqué `*` devant le `ngIf` ? Sachez qu'il s'agit d'une part essentielle de la syntaxe, et que vous ne devez pas l'oublier sous peine de lever une erreur !

Afficher un tableau

Comment faites-vous si une de vos propriétés est un tableau ? En effet, comment afficher cette propriété dans votre template ? Et bien en utilisant une directive dédiée : *ngFor*.

Imaginons que nous souhaitons afficher une propriété d'un composant qui contient une liste de Pokémons, et que cette propriété se nomme *pokemons*. Nous devons utiliser la directive *ngFor* dans notre template :

```
<ul>  
<li *ngFor="let pokemon of pokemons">{{ pokemon.name }}</li>  
</ul>
```

L'utilisation de cette directive est intuitive, vous l'appliquez sur l'élément HTML que vous souhaitez afficher pour chaque élément de votre liste. Ici nous souhaitons afficher une liste des Pokémons, nous devons donc appliquer la directive sur l'élément **.

Ensuite, nous passons une expression à notre directive : *let pokemon of pokemons*.

Cette directive permet de définir une variable locale *pokemon* dans le code HTML qui est répété en boucle. Nous pouvons ensuite accéder à cette variable grâce à l'interpolation : *{{ pokemon.name }}*.

Le terme *of* permet de faire comprendre à Angular que cette variable locale sera extraite du tableau *pokemons*.

7. Un peu de pratique : Modifions les templates de notre application

Je vous propose d'améliorer le template qui liste les Pokémons, grâce à la directive *NgFor*.

Cependant, avant de développer notre template, je vous propose d'ajouter la librairie CSS *Materialize* pour avoir une interface utilisateur digne d'un vrai site. Vous n'êtes pas obligé de l'utiliser, mais cela vous permettra d'avoir la même base de code que celle utilisée dans ce cours.

Pour cela, ajoutez le CSS de Materialize pour styliser notre application. Ajoutez cette ligne dans le fichier *index.html*, avant la balise `</head>` :

```
<!-- Chargement de Materialize pour l'interface utilisateur -->  
<link rel="stylesheet"  
href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">
```

Ensuite, modifiez la ligne concernant le template, dans *app.component.ts*, car nous allons charger notre template depuis un fichier externe désormais :

```
templateUrl: './app.component.html'
```

Vous devez ensuite créer le fichier du template *app.component.html*, dans le dossier *app*.

C'est tout bon ! Attaquons le vif du sujet, notre template !

Je vous encourage fortement à essayer de le faire par vous-même avant de regarder la solution.

8. Correction

Je vous fournis ci-dessous une correction avec le style fourni par Materialize. Le choix de l'interface utilisateur est très subjectif, et il est tout à fait probable que vous n'ayez pas le même code que celui de la correction. Je vous invite à vous en inspirer pour avoir la même base dans la suite du cours. Voici le template du composant à ajouter dans le fichier *app.component.html* :

```
<h1 class="center">Pokémons</h1>
<div class="container">
  <div class="row">
    <div *ngFor="let pokemon of pokemons" class="col s6 m4">
      <div class="card horizontal" (click)="selectPokemon(pokemon)">
        <div class="card-image">
          <img [src]="pokemon.picture">
        </div>
        <div class="card-stacked">
          <div class="card-content">
            <p>{{ pokemon.name }}</p>
            <p><small>{{ pokemon.created }}</small></p>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

TOUTES les classes utilisées ci-dessus sont des classes issues de la librairie Materialize, et ne sont pas liées à la syntaxe des templates d'Angular.

Nous allons détailler, comme ça personne n'est perdu.

- **Ligne 1** : Un titre avec la classe *center* : c'est une classe de Materialize qui permet de centrer du texte.
- **Ligne 2** : Une balise *div* avec la classe *container* : c'est une classe de Materialize qui permet de centrer le contenu de notre page web.

- **Ligne 3** : Une balise *div* avec la classe *row* : c'est encore une classe de Materialize (décidément !) qui permet de définir une ligne dans la page, ce qui va nous permettre d'afficher nos Pokémons les uns à la suite des autres.
- **Ligne 4** : On utilise la directive ngFor, afin de créer un nouveau bloc pour chacun de nos Pokémons. On précise que sur des petits écrans, chaque Pokémon prendra la moitié de l'écran : s6, et sur les autres écrans (tablettes, PC) chaque Pokémon prendra un tiers de l'écran : m4. Ces classes signifient respectivement : s pour *small* (petits écrans) et m pour *medium* (écrans moyens). Le container que nous avons défini précédemment possède une largeur arbitraire de 12, donc s6 signifie : « *Prendre la moitié de l'écran sur les petits écrans* », car 6 est bien la moitié de 12 !
- **Ligne 5** : On utilise la classe *card* et *horizontal* pour définir un bloc avec une image à gauche et un texte à droite au sein de chaque bloc. On détecte le clic sur ce bloc, en appelant la méthode *selectPokemon* en passant en paramètre le pokémon sur lequel l'utilisateur a cliqué.
- **Ligne 6** : On utilise simplement une classe de Materialize pour ajouter un peu de style... je ne présenterai plus les lignes de ce genre.
- **Ligne 7** : On utilise la liaison de propriété pour définir une source à notre image. N'oubliez pas d'ajouter les crochets pour qu'Angular puisse interpréter l'expression passée en paramètre !
- **Ligne 11 et 12** : On utilise l'interpolation pour afficher les propriétés de nos Pokémons !

Je vous invite à jeter un coup d'œil à la documentation de Materialize (materializecss.com/cards) pour comprendre ce que font toutes ces classes. La documentation est très bien faite, et il nous suffit la plupart du temps de récupérer les classes dont on a besoin pour ajouter du style à notre page web.

Au final, voici ce que vous devriez obtenir, après avoir lancé l'application avec `ng serve -o`, bien sûr !

Pokémons



Notre application de Pokémons s'affiche dans le navigateur.

9. Conclusion

Nous avons vu beaucoup de choses pendant ce chapitre sur les templates Angular. Sachez que le sujet est vaste, mais que nous avons vu ici les fonctionnalités principales dont nous aurons besoin pour développer notre application.

Dans le prochain chapitre, je vous propose d'éclaircir un élément que nous n'avons pas encore vraiment exploré : les *directives*.

En résumé

- L'interpolation permet d'afficher des propriétés de nos composants dans les templates, via la syntaxe `{{ }}`.
- On peut lier une propriété d'élément, d'attribut, de classe ou de style d'un composant vers le template.
- Si nos templates sont trop longs, on peut utiliser le backtick ` d'ES6, ou définir nos templates dans des fichiers séparés.
- La directive *NgIf* permet de conditionner l'affichage d'un template.
- La directive *NgFor* permet d'afficher une propriété de type tableau dans un template.
- On peut gérer les interactions d'un utilisateur avec un élément de la page grâce à la syntaxe : `(' + 'nom de l'événement' + ')`.
- On peut référencer des variables directement dans le template plutôt que de manipuler l'objet `$event`.
- Les variables référencées dans le template sont accessibles pour tous les éléments fils et frères de l'élément dans lequel elles ont été déclarées.
- Essayez d'éviter de mettre la logique de votre application dans vos templates. Gardez-les le plus simple possible !

Chapitre 8 : Les Directives

Nous avons déjà utilisé les *directives* à plusieurs reprises, sans le savoir. Et pour cause, les directives se retrouvent partout dans une application Angular. Nous allons rapidement voir à quelles occasions nous avons déjà eu affaire aux directives, et comment créer ses propres directives !

Allez hop, au boulot !

1. Qu'est-ce qu'une directive ?

Commençons par un peu de théorie. Qu'est-ce que peut bien être une *Directive* ? Et bien comme je viens de vous le dire, vous avez déjà utilisé des directives, à votre insu !

Une directive est une classe Angular qui ressemble beaucoup à un composant, sauf qu'elle n'a pas de template (D'ailleurs, au sein du Framework, la classe *Component* hérite de la classe *Directive*). Au lieu d'annoter cette classe avec *@Component*, vous utiliserez *@Directive*, logique !

Une directive permet d'interagir avec des éléments HTML d'une page, en leur attachant un comportement spécifique.

Nous avons déjà utilisé les directives *NgIf* et *NgFor*, qui sont des directives structurelles. On peut avoir plusieurs directives appliquées à un même élément !

Une directive possède un sélecteur CSS, qui indique au Framework où l'activer dans notre template.

Lorsque Angular trouve une directive dans un template HTML, il instancie la classe de la Directive correspondante et donne à cette instance le contrôle sur la portion du DOM qui lui revient.

Bon, tout ça peut paraître un peu abstrait pour le moment. Avant de nous lancer dans le concret pour créer notre propre directive, nous allons voir les trois types de directives existantes :

Les composants : oui, vous avez bien lu, tous les composants que nous avons développés jusqu'à maintenant étaient également des directives ! Ils sont les pierres angulaires d'une application Angular (sans mauvais jeu de mots) et les développeurs peuvent s'attendre à devoir en écrire beaucoup.

Les directives d'attributs : Elles peuvent modifier le comportement des éléments HTML, des attributs, des propriétés et des composants. Elles sont représentées habituellement par des attributs au sein de balises HTML, d'où leur nom.

Les directives structurelles : ces directives sont responsables de mettre en forme une certaine disposition d'éléments HTML, en ajoutant, retirant ou manipulant des éléments et leur fils. Les directives `ngIf` et `ngFor` sont des directives structurelles.

2. Créer une directive d'attribut

Nous allons voir comment créer une directive d'attribut pour notre application. Ce type de directive va nous permettre de changer l'apparence ou le comportement d'un élément.

Je vous propose de créer la directive *BorderCardDirective*, qui permettra d'ajouter une bordure de couleur sur les Pokémons de notre liste, lorsque l'utilisateur les survolera avec son curseur. Nous fixerons également une hauteur commune à tous les pokémons, afin que les éléments de notre liste soient toujours alignés. Voici ce que vous devriez obtenir :

Pokémons



On affiche une bordure lorsque le curseur de l'utilisateur survole un Pokémon.

Ici je survole Salamèche avec mon curseur, mais comme j'ai réalisé l'image avec une capture d'écran, le curseur n'apparaît pas, c'est normal !

Pour commencer, nous savons qu'une directive d'attribut requiert au minimum une classe annotée avec *@Directive*.

Dans cette annotation, nous allons devoir préciser le sélecteur css qui permettra d'appliquer la directive sur nos éléments HTML. Dans la classe de notre directive nous décrirons le comportement souhaité.

Créons donc une directive globale à notre application dans le dossier *app*, nommée *border-card.directive.ts* :

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

@Directive({ selector: '[pkmnBorderCard]' })

export class BorderCardDirective {

  constructor(private el: ElementRef) {

    this.setBorder('#f5f5f5');

    this.setHeight(180);

  }
```

```
private setBorder(color: string) {
  let border = 'solid 4px ' + color;
  this.el.nativeElement.style.border = border;
}

private setHeight(height: number) {
  this.el.nativeElement.style.height = height + 'px';
}
}
```

Tout d'abord, nous avons importé deux d'éléments depuis la librairie *core* d'Angular. Détaillons chacun de ces éléments :

- **Directives** : Nous en avons besoin pour pouvoir utiliser l'annotation *@Directive*.
- **ElementRef** : Cet objet représente l'élément du DOM sur lequel nous appliquons notre directive, nous y avons directement accès en paramètre du constructeur.

Ensuite nous passons un sélecteur css en paramètre de l'annotation `@Directive`, à la ligne 3. Notre directive s'appliquera donc à tous les éléments du DOM qui ont un attribut `pkmnBorderCard`, conformément au sélecteur que nous avons indiqué. Vous avez remarqué que nous avons bien mis le nom de l'attribut entre crochets dans le sélecteur. En effet, si nous avions simplement mis ceci ...

```
@Directive({ selector: 'pkmnBorderCard'}) // à ne pas faire !  
export class BorderCardDirective { }
```

... alors notre directive se serait appliquée sur toutes les balises `<pkmnBorderCard>`, ce qui n'a pas de sens, puisque dans ce cas ce ne serait pas une directive d'attribut mais un composant !

Lorsque nous avons choisi le nom de notre directive, nous avons préfixé *border-card* par *pkmn*. Il est recommandé de préfixer leur nom, afin d'éviter le risque de collisions avec des librairies tierces, ou avec des attributs HTML standard.

Par exemple, si votre application s'appelle *AwesomeApp*, utilisez le préfixe *aa*`<Nom-de-votre-directive>`.

N'utilisez pas *ng* comme préfixe pour vos directives ! Ce préfixe est réservé par Angular. De plus, utilisez la syntaxe *camelCase* pour nommer vos directives.

Enfin on exporte la classe *BorderCardDirective* pour pouvoir utiliser cette directive dans nos composants !

Bref, récapitulons : Angular crée une nouvelle instance de notre directive à chaque fois qu'il détecte un élément HTML avec l'attribut `pkmnBorderCard`, et injecte l'élément du DOM et de quoi le modifier dans le constructeur de la directive. A partir de là, notre directive impose une bordure grise et une hauteur de 180px aux éléments HTML sur lesquels elle est rattachée, grâce aux méthodes *setBorder* et *setHeight* que nous avons définies.

3. Prendre en compte les actions de l'utilisateur

Ce que nous aimerions maintenant, c'est que la bordure change de couleur lorsque l'utilisateur survole un élément. Pour cela nous avons besoin de :

- Détecter lorsque le curseur entre ou sort d'un élément de la liste.
- Définir une action pour chacun de ces événements.

On va utiliser une nouvelle annotation `@HostListener`. Cette annotation permet de lier une méthode de notre directive à un événement donné. Nous allons donc créer deux nouvelles méthodes dans notre directive, qui seront appelées respectivement quand le curseur de l'utilisateur entre sur un élément, et quand il en ressort : ce sont les événements *mouseenter* et *mouseleave*.

Voici le code final de notre directive, qui utilise l'annotation `@HostListener` :

```
1  import { Directive, ElementRef, HostListener } from '...';
2
3  @Directive({ selector: '[pkmnBorderCard]' })
4  export class BorderCardDirective {
5
6    constructor(private el: ElementRef) {
7      this.setBorder('#f5f5f5');
8      this.setHeight(180);
9    }
10
11    @HostListener('mouseenter') onMouseEnter() {
12      this.setBorder('#009688');
13    }
14
15    @HostListener('mouseleave') onMouseLeave() {
16      this.setBorder('#f5f5f5');
```

```

17 }
18
19 private setBorder(color: string) {
20   let border = 'solid 4px ' + color;
21   this.el.nativeElement.style.border = border;
22 }
23
24 private setHeight(height: number) {
25   this.el.nativeElement.style.height = height + 'px';
26 }
27 }

```

Voici ce que nous avons ajouté par rapport à la version précédente de notre directive :

- Nous avons importé *HostListener* depuis la librairie core d'Angular à la ligne 1.
- Nous avons défini la propriété *el* grâce à la syntaxe des constructeurs de TypeScript, à la ligne 6. C'est pourquoi nous pouvons utiliser *this.el* ailleurs dans notre directive.
- Nous utilisons l'annotation *@HostListener* à la ligne 11 et 15, pour détecter deux événements : *mouseenter* et *mouseleave*.

Il est fortement recommandé d'utiliser *@HostListener* plutôt que de définir nous-même des écouteurs d'événements. Vous y gagnerez à tous les niveaux : cela vous évite d'interagir directement avec l'API du DOM, vous n'avez pas à détacher vous-même les écouteurs d'événements lorsque la directive est détruite, etc.

Voilà, nous avons désormais une directive qui va rajouter une bordure de couleur lorsqu'un utilisateur passera son curseur au-dessus d'un Pokémon de la liste. Il ne reste plus qu'à l'intégrer à notre application.

4. Utiliser notre directive

Il faut maintenant déclarer notre directive pour pouvoir l'utiliser dans les templates de nos composants. Ouvrez le fichier *app.module.ts* et ajoutez-y les lignes suivantes :

```
// ...
// importez la directive
import { BorderCardDirective } from './border-card.directive';

@NgModule({
  imports: [ BrowserModule ],
  // déclarez BorderCardDirective dans le module racine declarations: [ AppComponent,
  // BorderCardDirective ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Maintenant, il ne nous reste plus qu'à appliquer cette directive dans le template de notre composant *app.component.html*, comme ceci :

```
<div *ngFor='let pokemon of pokemons' class="col s6 m4">
  <!-- on applique notre directive ici : -->
  <div class="card horizontal" (click)="selectPokemon(pokemon)" pkmnBorderCard>
    ...
  </div>
</div>
```

Maintenant, lancez l'application avec *npm start* si vous l'aviez fermée, chaque Pokémon devrait être entouré d'une bordure de couleur lorsque vous le survolez !

5. Ajouter un paramètre à notre directive

Pour le moment, notre directive *pkmnBorderCard* n'est pas personnalisable.

À chaque utilisation, cette directive impose une couleur unique aux bordures. Je vous propose donc de paramétrer la couleur des bordures. De cette manière il sera possible de préciser une couleur lors de l'appel de la directive dans un template.

Pour cela, nous devons préciser une propriété d'entrée pour notre directive, avec l'annotation *@Input*. Nous allons ajouter une propriété *borderColor* pour paramétrer la couleur des bordures :

```
1 // 1. On ajoute 'Input' à nos importations
2 import { ..., Input } from '@angular/core';
3
4 @Directive({ selector: '[pkmnBorderCard]' })
5 export class BorderCardDirective {
6
7   constructor(private el: ElementRef) {
8     this.setBorder('#f5f5f5');
9     this.setHeight(180);
10  }
11
12 // 2. On declare notre propriété borderColor
13 @Input('pkmnBorderCard') borderColor: string;
14
15 @HostListener('mouseenter') onMouseEnter() {
16   // 3. On attribue la couleur souhaitée,
17   // sinon on attribue une valeur par défaut
18   this.setBorder(this.borderColor || '#009688');
19 }
20
21 @HostListener('mouseleave') onMouseLeave() {
22   this.setBorder('#f5f5f5');
23 }
24
25 private setBorder(color: string) {
26   let style = 'solid 4px ' + color;
```

```

27     this.el.nativeElement.style.border = border;
28   }
29
30   private setHeight(height: number) {
31     this.el.nativeElement.style.height = height + 'px';
32   }
33 }

```

Nous avons fait trois choses dans le code ci-dessous :

1. Importer l'annotation `@Input` depuis le paquet `@angular/core`, à la ligne 2.
2. Déclarer la propriété `borderColor` avec un alias, à la ligne 13. Les explications sur ce qu'est un alias se trouvent un peu plus bas.
3. Utiliser cette propriété dans la méthode `setBorder`, à la ligne 18. On utilise l'opérateur logique OU (`||`) pour définir une valeur par défaut s'il n'y a aucune couleur qui a été définie dans le template.

Ensuite, depuis votre template `app.component.html`, vous pouvez paramétrer votre directive comme ceci :

```

<div
  class="card horizontal"
  (click)="selectPokemon(pokemon)"
  [pkmnBorderCard]="red">

```

Vous obtiendrez une bordure rouge au survol. C'est quand même plus sympathique de pouvoir choisir la couleur des bordures !

Bon, de mon côté, j'ai quand même laissé la couleur par défaut.

C'est quoi, un « alias » ?

En fait, il y a deux manières de déclarer une propriété d'entrée : avec ou sans alias.

```

@Input() pkmnBorderCard: string; // sans alias

```

```
@Input('pkmnBorderColor') borderColor: string; // avec alias
```

Sans alias, nous sommes obligés d'utiliser le nom de la directive pour nommer la propriété. Or ce nom de propriété, *pkmnBorderColor*, n'est pas un nom adapté pour la couleur de nos bordures.

Heureusement, grâce à l'alias, vous pouvez nommer la propriété de la directive comme vous voulez, et utilisez ce nom ailleurs dans votre directive. Spécifiez simplement le nom de la directive dans l'argument *@Input*.

6. Réorganiser notre code

Pour terminer le développement de notre directive, je vous propose de remplacer les valeurs codées « en dur », par des propriétés :

- *initialColor* : la couleur initiale affichée au chargement de la page.
- *defaultColor* : la couleur par défaut si aucune couleur de bordure n'est précisée.
- *defaultHeight* : la hauteur par défaut du cadre pour la bordure.

Le code de ce que j'obtiens de mon côté :

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

@Directive({
  selector: '[pkmnBorderCard]'
})
export class BorderCardDirective {

  private initialColor: string = '#f5f5f5';
  private defaultColor: string = '#009688';
  private defaultHeight: number = 180;

  constructor(private el: ElementRef) {
    this.setBorder(this.initialColor);
    this.setHeight(this.defaultHeight);
  }
}
```

```
@Input('pkmnBorderCard') borderColor: string;

@HostListener('mouseenter') onMouseEnter() {
  this.setBorder(this.borderColor || this.defaultColor);
}
```

```
@HostListener('mouseleave') onMouseLeave() {
  this.setBorder(this.initialColor);
}

private setBorder(color: string) {
  let border = 'solid 4px ' + color;
  this.el.nativeElement.style.border = border;
}

private setHeight(height: number) {
  this.el.nativeElement.style.height = height + 'px';
}
}
```

Vous pouvez bien sûr continuer à vous entraîner en améliorant cette directive. Par exemple, en ajoutant des propriétés d'entrées pour paramétrer la largeur ou le type de la bordure : pointillé, double trait... ou tout ce qui vous passe par la tête !

Faites preuve d'imagination.

Et bien justement, en parlant d'imagination... comment je peux créer une directive avec plusieurs propriétés d'entrées, si jamais un jour j'en ai besoin ?

Une fois que vous savez créer une directive avec un paramètre, vous n'êtes plus très loin de la solution. Il suffit de passer autant de paramètres que vous le souhaitez à votre directive entre crochets, depuis le composant concerné, comme ceci :

```
<div  
  [myDirective] = 'option'  
  [first]='FirstParameter'  
  [second]='SecondParameter'  
  ...  
</div>
```

Et dans le code de la directive, on déclare autant de propriétés d'entrée que nécessaire :

```
@Input('myDirective') option;  
@Input('first') first;  
@Input('second') second;
```

Voilà, maintenant vous n'avez plus d'excuses pour développer la directive de vos rêves.

7. Conclusion

Nous avons vu comment créer et utiliser des directives dans notre application. Même si la plupart du temps nous utiliserons les directives natives fournis par Angular, ou des directives issues de librairies, il est parfois intéressant de définir ses propres directives, pour définir un comportement spécifique aux besoins de notre application.

Dans tous les cas, vous connaissez maintenant le fonctionnement !

En résumé

- On utilise l'annotation *@Directive* pour déclarer une directive dans notre application.
- Il existe trois types de directives différentes : les composants, les directives d'attributs et les directives structurelles (*ngFor* et *ngIf* par exemple).
- Une directive d'attribut permet d'agir avec les éléments HTML d'une page, en leur attachant un comportement spécifique.
- Une directive utilise un sélecteur CSS pour cibler les éléments HTML sur lesquels elle s'applique.
- Il est recommandé de préfixer le nom de ses directives pour éviter les problèmes de collisions.
- Angular crée une nouvelle instance de notre directive à chaque fois qu'il détecte un élément HTML avec l'attribut correspondant. Il injecte alors dans le constructeur de la directive l'élément du DOM *ElementRef*.
- Il faut déclarer notre directive pour pouvoir l'utiliser.
- On utilise l'annotation *@HostListener* pour gérer les interactions de l'utilisateur au sein d'une directive.

Chapitre 9 : Les Pipes

Grâce à *l'interpolation*, nous avons vu que nous pouvons afficher des données dans les templates. Mais parfois les données que l'on récupère ne peuvent pas être affichées directement à l'utilisateur. L'exemple le plus courant est le cas des dates.

Imaginez que vous récupérez une date depuis un serveur distant, qui représente la date de la dernière connexion de l'utilisateur : *Mon Apr 18 2017 12:45:26*. Vous ne souhaitez pas afficher ce texte directement à l'utilisateur, n'est-ce pas ? Vous avez donc besoin de le *formater* un peu, afin d'afficher quelque chose de plus lisible à l'utilisateur.

De plus, au fur et à mesure que vous développerez, vous vous rendrez compte que vous avez souvent besoin d'effectuer les mêmes *transformations*, dans plusieurs templates différents, à travers vos projets.

C'est pourquoi Angular dispose de *pipes*, pour vous permettre d'effectuer plus simplement ces transformations dans vos templates.

1. Utiliser un Pipe

Rien ne vaut un exemple pour comprendre. Je vous propose d'afficher des dates formatées à nos utilisateurs. Voici à quoi ressemble les dates affichées dans nos templates pour le moment : *Mon Aug 29 2016 17:28:45 GMT+0200 (CEST)*. On ne peut pas dire que ça soit très sexy !

Je vous propose d'appliquer un *pipe* présent nativement dans toutes les applications Angular : le pipe Date.

Prenons le composant suivant, qui affiche simplement une date d'anniversaire :

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

@Component({
  selector: 'birthday',
  template: `<p>Date d'anniversaire : {{ birthday | date }}</p>`
})
export class BirthdayComponent {
  birthday = new Date(1992, 12, 19);
}
```

Ce composant affichera comme date d'anniversaire *Dec 19, 1992*. C'est déjà plus sympa que ce que nous avions.

A la ligne 8, nous définissons une propriété *birthday* dans le composant, et nous lui attribuons comme valeur la date du 19 Décembre 1992 (Pourquoi pas ?). Ensuite regardez la ligne 5, nous utilisons l'interpolation pour afficher cette propriété, jusque-là rien d'anormal, mais ensuite, vous apercevez ce code : `| date`.

Qu'est-ce que cela, me direz-vous ? Et bien il s'agit ni plus ni moins de la syntaxe des *pipes* ! D'abord on place l'opérateur de pipe `|` à droite de la propriété sur laquelle on veut appliquer la transformation, et ensuite on applique le nom du pipe que l'on veut utiliser, à la suite de cet opérateur. Tous les pipes fonctionnent de cette manière.

1.1. Les pipes disponibles de base

Angular est fourni avec un certain nombre de pipes qui sont immédiatement disponibles dans tous nos templates. Pourquoi s'en priver ?

Voici une liste de pipes directement disponibles : *DatePipe* pour afficher les dates au bon format, *UpperCasePipe* et *LowerCasePipe* pour mettre un texte en minuscule ou en majuscule, *CurrencyPipe* pour l'affichage des devises (\$, €), ...

1.2. Combiner les pipes

Il est possible d'utiliser plusieurs pipes différents pour une même propriété, et de les combiner pour obtenir l'affichage souhaité.

Par exemple, imaginons que l'on souhaite afficher une date en majuscule, et bien il suffit d'utiliser le pipe *date* et le pipe *uppercase*, successivement, en utilisant l'opérateur de pipes :

```
<p>{{ birthday | date | uppercase }}</p>
```

Les transformations s'appliquent de la gauche vers la droite : d'abord *date* puis *uppercase*.

1.3. Paramétrer nos pipes

Les pipes peuvent faire l'objet d'une utilisation plus poussée. Par exemple pour *date*, on ne choisit pas vraiment le format que l'on souhaite afficher pour nos dates, on se contente du résultat que nous fournit le pipe. Heureusement, il est possible de passer des paramètres aux pipes, afin de mieux définir l'affichage souhaité.

Un pipe peut accepter n'importe quel nombre de paramètres facultatifs. On ajoute des paramètres à un pipe avec des « : » et les valeurs des paramètres. Par exemple, pour afficher un prix en euros :

```
<p>{{ 15 | currency:'EUR' }}</p>
```

Et on peut faire la même chose pour nos dates, en passant en paramètre un format pour l'affichage :

```
<p>{{ birthday | date:'dd/MM/yyyy' }}</p>
```

1.4. Améliorons l’affichage des dates de notre application

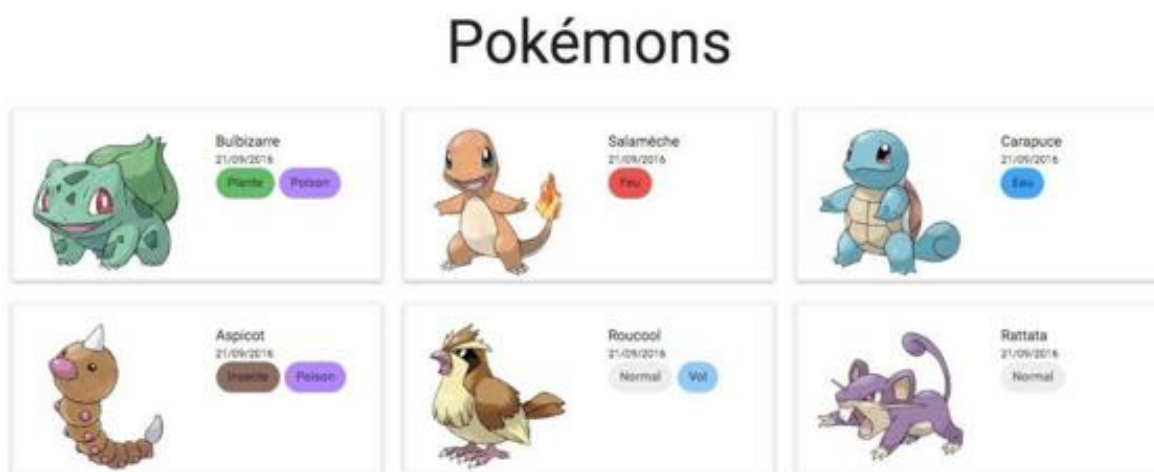
Essayons tout de suite d’ajouter un pipe dans notre application, afin de proposer le format de date suivant à nos utilisateurs : 19/12/1992 pour le 19 décembre 1992. Je vous laisse ajouter le pipe vous-même, et je vous donne la solution à la fin du chapitre.

Oui, il s’agit bien d’un exercice, mais qui vous prendra moins de deux minutes !

2. Créer un pipe personnalisé

Angular nous permet bien sûr de créer nos propres pipes, pour les besoins de notre application.

Je vous propose d'ajouter le pipe *PokemonTypeColorPipe* dans notre application. Il s'agit d'un simple pipe qui est censé renvoyer une couleur correspondant au type d'un Pokémon. Par exemple, si notre Pokémon est de type *eau*, alors le pipe renverra la couleur bleue, et si le Pokémon est de type *feu*, alors le pipe renverra la couleur rouge. Voilà ce que nous devrions obtenir :



Nos pokémons avec les pipes !

Dans notre dossier */app*, ajoutez un fichier nommé *pokemon-type-color.pipe.ts* :

```
import { Pipe, PipeTransform } from '@angular/core';

/*
 * Affiche la couleur correspondant au type du pokémon.
 * Prend en argument le type du pokémon.
```


** Exemple d'utilisation:*

** {{ pokemon.type | pokemonTypeColor }}*

**/*

@Pipe({name: 'pokemonTypeColor'})

export class PokemonTypeColorPipe implements PipeTransform {

transform(type: string): string {

let color: string;

switch (type) {

case 'Feu':

color = 'red lighten-1';

break;

case 'Eau':

color = 'blue lighten-1';

break;

case 'Plante':

color = 'green lighten-1';

break;

case 'Insecte':

color = 'brown lighten-1';

break;

case 'Normal':

color = 'grey lighten-3';

break;

case 'Vol':

```
color = 'blue lighten-3';  
  
break;
```

```
case 'Poison':  
  color = 'deep-purple accent-1';  
  break;  
case 'Fée':  
  color = 'pink lighten-4';  
  break;  
case 'Psy':  
  color = 'deep-purple darken-2';  
  break;  
case 'Electrik':  
  color = 'lime accent-1';  
  break;  
case 'Combat':  
  color = 'deep-orange';  
  break;  
default:  
  color = 'grey';  
  break;  
}  
return 'chip ' + color;  
}  
}
```

Examinons un peu ce code :

D'abord, on déclare un pipe en appliquant sur une classe l'annotation `@Pipe`, que l'on importe depuis la librairie `core` d'Angular.

Ensuite, la classe d'un pipe implémente l'interface *PipeTransform*. Cette interface possède une méthode *transform* qui accepte une valeur (qui correspond à la valeur de la propriété sur laquelle s'applique notre pipe, dans notre cas ce sera le type d'un Pokémon), suivie par plusieurs paramètres facultatifs (notre pipe n'aura pas besoin de paramètres, nous n'avons donc qu'un paramètre dans notre méthode *transform*).

Ensuite dans la fonction *transform* de notre pipe, on déduit la couleur à partir du type du Pokémon. Par exemple, si le Pokémon est de type *Feu*, alors le pipe renverra la couleur rouge.

Les valeurs renvoyées par le pipe sont des classes de la librairie *Materialize*, qui permettent d'attribuer une couleur aux éléments.

Pour pouvoir utiliser notre pipe, il ne faut pas oublier de le déclarer dans le fichier *app.module.ts* :

```
// ...
import { PokemonTypeColorPipe } from './pokemon-type-color.pipe';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [
    AppComponent,
    BorderCardDirective,
    // On déclare notre pipe PokemonTypeColorPipe dans le module :
    PokemonTypeColorPipe
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

C'est bon, il ne nous reste plus qu'à appliquer notre pipe dans le template *app.component.html* :

```
<h1 class='center'>Pokémons</h1>
...
<div *ngFor='let pokemon of pokemons' class="col s6 m4">
...
  <div class="card-content">
    <p>{{ pokemon.name }}</p>
    <!-- On ajoute le pipe sur les dates, avec le bon format: -->
    <p><small>
      {{ pokemon.created | date:"dd/MM/yyyy" }}
    </small></p>
    <!-- On utilise la directive ngFor,
           pour afficher tous les types d'un pokémon donnée : -->
    <span
      *ngFor='let type of pokemon.types'
```

```
class="{{ type | pokemonTypeColor }}">
  {{ type }}
</span>
</div>
...
</div>
```

A la ligne 9, nous utilisons le pipe natif *date*, afin de formater les dates de notre template. Il s'agit de la correction que je vous avais promise à la section précédente.

Enfin à la ligne 15, nous utilisons notre propre pipe *pokemonTypeColor*, afin d'afficher le type du Pokémon avec la couleur correspondante à son type !

3. Conclusion

Les *pipes* sont pratiques pour formater des données jugées trop « brutes », au sein de vos templates. Plutôt que de définir les mêmes transformations à travers plusieurs composants, on peut développer des *pipes* personnalisées afin de centraliser la définition de ces transformations.

C'est très pratique et il ne faut pas hésiter à utiliser les *pipes* dès que vous jugez que cela est nécessaire !

En résumé

- Les *pipes* permettent de formater les données affichées dans nos templates.
- L'opérateur des *pipes* est « | ».
- Angular fournit des *pipes* prêts à l'emploi, disponibles dans tous les templates de notre application : *DatePipe*, *UpperCasePipe*, *LowerCasePipe*, etc.
- Les *pipes* peuvent avoir des paramètres, mais tous les paramètres sont facultatifs.
- On peut créer des *pipes* personnalisés pour les besoins de notre application avec l'annotation *@Pipe*.
- Les pipes personnalisés doivent être déclarés avant de pouvoir être utilisés dans les templates de composants.

Chapitre 10 : Les Routes

Pour l'instant notre application est assez limitée, elle est composée d'un unique composant, accessible par défaut au démarrage de l'application. Nous ne pouvons donc pas développer une application plus complexe, avec plusieurs composants, et une navigation entre des *urls* différentes.

Je vous propose de remédier à ce problème, en dotant notre site web d'un système de navigation digne de ce nom. La question « *Comment mettre en place plusieurs pages dans mon application ?* » n'aura plus de secret pour vous.

1. Le fonctionnement de la navigation

Tout d'abord, vous devez savoir que le système de navigation fourni par Angular simule parfaitement la navigation auprès de votre navigateur. Lorsque vous naviguez dans votre application, vous pouvez revenir en arrière, rentrer directement une url dans la barre du navigateur pour rejoindre une autre page de votre application, et l'historique du navigateur sera automatiquement mis à jour ! Angular s'occupe pour nous de simuler la navigation auprès du navigateur.

... et comment on le met en place ce merveilleux système de navigation ?

Avant de voir l'aspect technique de la chose, nous allons voir comment sont organisées les routes dans une application.

Les routes doivent être regroupées par grande fonctionnalité au sein de votre application : on parle de *modules*.

Par exemple, toutes les routes concernant les Pokémons devront être centralisées au même endroit, c'est-à-dire dans le même module, et toutes les routes concernant les autres aspects de notre application (Connexion, Inscription, Profil) dans un autre module. Je vous propose de voir le système de routes d'Angular avec un système de routes basique entre deux pages, au sein de notre *module racine* existant.

2. Deux nouveaux composants

Pour pouvoir mettre en place la navigation dans notre application, nous allons avoir besoin d'au moins deux composants. Créez donc deux nouveaux composants dans le dossier *src/app* de notre projet :

Url	Composant	Rôle
/pokemons	<i>list-pokemon.component.ts</i>	Lister tous les Pokémons de l'application.
/pokemon/:id	<i>detail-pokemon.component.ts</i>	Afficher une page d'information sur un pokémon.

Le composant qui affichera des détails devra être accessible à l'url */pokemon/:id*, où *:id* désigne un nombre entier représentant l'identifiant du Pokémon. Par exemple, la route */pokemon/3* affichera le Pokémon avec l'identifiant n° 3 (qui est Carapuce dans notre cas).

Prenez les 5 prochaines minutes pour essayer de la faire par vous-même, il n'y a pas de pièges, promis !

Le code complet des deux composants est disponible plus loin dans le chapitre si vous êtes bloqués. De plus, vous pouvez tout de suite créer les fichiers de templates pour ces deux nouveaux composants.

2.1. Le composant DetailPokemonComponent

Avant de construire notre système de navigation, nous avons besoin de créer deux nouveaux composants : commençons par le composant *detail-pokemon.component.ts*. Créez ce fichier dans le dossier *app* de notre projet. Nous allons voir étape par étape comment construire ce fichier.

Les importations

Tout d'abord, nous devons importer un certain nombre d'éléments dont nous allons avoir besoin :

```
import { Component, OnInit } from '@angular/core';  
import { ActivatedRoute, Router, Params } from '@angular/router';  
import { Pokemon } from './pokemon';  
import { POKEMONS } from './mock-pokemons';
```

Il n'y a rien de nouveau ici, excepté à la ligne 2. Nous importons les éléments *ActivatedRoute*, *Router* et *Params*. Ces éléments vont nous être utiles pour extraire l'identifiant du pokémon à afficher, qui est contenu dans l'url du composant. Et l'élément *Router* va nous permettre de rediriger l'utilisateur.

L'annotation du composant

L'annotation du composant est standard. (On utilisera un autre fichier pour le template.)

```
@Component({  
  selector : 'detail-pokemon',  
  templateUrl : './detail-pokemon.component.html'  
})
```

La classe du composant

Voici le code de la classe du composant *detail-pokemon.component.ts* :

```
export class DetailPokemonComponent implements OnInit {
  pokemons: Pokemon[] = null; // La liste des pokémons de notre application.
  pokemon: Pokemon = null; // Le pokémon à afficher dans le template.

  // On injecte 'route' pour récupérer les paramètres de l'url,
  // et 'router' pour rediriger l'utilisateur.
  constructor(private route: ActivatedRoute, private router: Router) {}

  ngOnInit(): void {
    // On initialise la liste de nos pokémons :
    this.pokemons = POKEMONS;
    // On récupère le paramètre 'id' contenu dans l'url :
    let id = +this.route.snapshot.paramMap.get("id");
    // On itère sur le tableau de pokémons pour trouver le pokémon avec le bon identifiant :
    for (let i = 0; i < this.pokemons.length; i++) {
      // Si on trouve un pokémon avec le bon identifiant,
      // on affecte ce pokémon à la propriété du composant :
      if (this.pokemons[i].id == id) {
        this.pokemon = this.pokemons[i];
      }
    }
  }

  // Redirige l'utilisateur vers la page principale
  goBack(): void {
    this.router.navigate(['/pokemons']);
  }
}
```

Le type void permet d'indiquer à TypeScript qu'une fonction n'a pas de valeur de retour.

Dans ce composant, nous avons besoin de faire quelque chose de particulier lors de son initialisation : nous devons récupérer l'identifiant qui se trouve dans l'url (exemple : */pokemon/3*) et ensuite récupérer le Pokémon qui correspond à cet identifiant. Regardons ce code, morceaux par morceaux :

De la ligne 2 à 3, nous déclarons deux propriétés pour notre composant :

- **pokemons** : Cette propriété contiendra la liste de tous les pokémons de notre application, soit celle que nous afficherons également dans le composant *list-pokemon.component.ts*.
- **pokemon** : Cette propriété contiendra le Pokémon à afficher à l'utilisateur.

À la ligne 5, nous déclarons un constructeur pour notre composant. Ce composant contient un paramètre *route* et *router*. Je vous présenterai dans le chapitre dédié à l'injection de dépendance, ce que signifie réellement cette syntaxe. Pour l'instant, retenez que nous venons simplement de déclarer à Angular : « Dans ce composant, je vais avoir besoin de récupérer des informations depuis l'url du composant, et aussi de rediriger l'utilisateur grâce à ces deux paramètres ».

À partir de la ligne 8, nous effectuons un certain nombre d'opérations pour initialiser le composant.

D'abord, nous récupérons tous les Pokémon de notre application et nous les affectons à la propriété *pokemons* :

```
this.pokemons = POKEMONS;
```

Ensuite nous récupérons l'identifiant du Pokémon contenu dans l'url

```
1 let id = +this.route.snapshot.paramMap.get('id');
2
3 for (let i = 0; i < this.pokemons.length; i++) {
4   if (this.pokemons[i].id == id) {
5     this.pokemon = this.pokemons[i];
6   }
7 }
```

Ligne 1 : La première ligne permet de récupérer le paramètre *id* de la route. La propriété *snapshot* indique que nous récupérons ce paramètre de manière synchrone. C'est-à-dire que nous bloquons l'exécution du programme tant que nous n'avons pas récupéré l'identifiant du Pokémon à afficher. Ensuite nous accédons à l'objet *paramMap* contenant tous les paramètres de la route sous forme d'un dictionnaire de valeur, et on extrait le paramètre *id*. (On utilise l'opérateur JavaScript « + » pour convertir une chaîne de caractère en nombre, car *paramMap* ne contient que des chaînes de caractère par défaut).

Ligne 3 : On itère sur les Pokémon de l'application pour trouver celui à afficher.

De la ligne 4 à la ligne 6 : On cherche le Pokémon qui a un identifiant qui correspond à l'identifiant de la route. Une fois ce Pokémon trouvé, nous l'affectons à la propriété *pokemon* de notre composant (ligne 5 ci-dessus).

Si aucun Pokémon n'est trouvé, alors la valeur de la propriété *pokemon* restera null. Nous pourrions traiter ce cas dans le template par la suite.

Enfin, nous avons ajouté une méthode permettant à l'utilisateur de revenir à la liste des Pokémon, après avoir consulté un Pokémon particulier :

```
// Méthode recommandée
goBack(): void {
  this.router.navigate(['/pokemons']);
}
```

Nous utilisons ici le routeur d'Angular (que nous avons injecté depuis le constructeur) pour rediriger l'utilisateur vers la liste des Pokémon. La méthode *navigate* du router permet de faire cela, et elle prend l'url de redirection en paramètre, dans un tableau (car on peut vouloir passer des paramètres à la route).

Pourquoi il y a écrit « Méthode recommandée » en commentaire (ligne 1) ?

Et bien, nous pouvons utiliser une autre façon de faire pour rediriger l'utilisateur. On peut utiliser *Location*, qui est un service mis à disposition par Angular, et que nous pouvons utiliser pour interagir avec l'URL du navigateur.

```
// Autre méthode
import { Location } from '@angular/common';
// ...
constructor(private location:Location) {}
// ...

goBack(): void {
  this.location.back();
}
```

Le code parle de lui-même, on accède au service Location, et on demande un retour en arrière avec la méthode back, et hop, le tour est joué : notre utilisateur peut revenir à liste des Pokémons !

Si l'utilisateur est arrivé depuis une autre page que la liste des pokémons, il sera renvoyé vers cette autre page ! C'est pour cela que la première méthode est plus fiable, car on sait exactement où sera redirigé l'utilisateur.

2.2. Le template du composant

Le code du template *detail-pokemon.component.html* est disponible avec les ressources du cours, car il est très long.

Je ne vais pas vous détailler ligne par ligne ce composant, puisqu'il s'agit essentiellement de code HTML et de classes nécessaires à la librairie Materialize. En revanche, il y a certains éléments à remarquer :

- **Ligne 1** : La directive `ngIf` nous permet de vérifier qu'un Pokémon est disponible. Si aucun Pokémon n'est disponible, on affiche un message d'erreur, à la ligne 45.
- **Ligne 6** : On utilise la liaison de donnée pour déterminer la valeur de l'attribut `src` de l'image à afficher.
- **Ligne 27** : La directive `ngFor` nous permet d'afficher tous les types de notre Pokémon avec une couleur déterminée, mais nous avons déjà effectué cette opération précédemment dans ce cours.
- **Ligne 39** : Nous écoutons l'événement `click` sur le lien, qui déclenche la méthode `goBack`.

Et que ce passe-t-il si aucun Pokémon n'est trouvé à partir de l'identifiant trouvé dans l'url ?

A la ligne 45, nous détectons ce cas grâce à la directive `*ngIf='!pokemon'`. Le template du composant affichera alors simplement le message : « *Aucun Pokémon à afficher !* ».

2.3. Le composant de la liste des Pokémons

Nous devons ensuite créer notre *list-pokemon.component.ts*, qui contient pour une grande part le code qu'il y avait dans *app.component.ts*. Créer donc deux nouveaux fichiers *list-pokemon.component.ts* et *list-pokemon.component.html* et copiez respectivement le code de *app.component.ts* et *app.component.html* à l'intérieur.

Effectuez ensuite les quatres modifications ci-dessous dans le composant *list-pokemon.component.ts* :

- Renommer la classe du composant en *ListPokemonComponent*.
- Modifier l'option *selector* et *templateUrl* de l'annotation *@Component* :
 - *selector* vaut : *'list-pokemon'*.
 - *templateUrl* vaut : *'./list-pokemon.component.html'*.
- Importez la classe Router : *import { Router } from '@angular/router'*;
- Injecter un router dans le composant : *constructor(private router: Router) {}*.
- Utiliser le router dans la méthode *selectPokemon* pour rediriger l'utilisateur vers la page qui détaille un Pokémon :

```
selectPokemon(pokemon: Pokemon): void {  
  let link = ['/pokemon', pokemon.id];  
  this.router.navigate(link);  
}
```

On utilise la méthode *navigate* du router pour rediriger l'utilisateur, en passant en paramètre de cette méthode un tableau contenant

deux éléments : l'url de redirection et les paramètres éventuels pour la route.

On peut également effectuer une redirection directement depuis le template grâce à la directive `routerLink`. Par exemple : `Tous les Pokémons`.

3. Mise en place du système de navigation

Pour l'instant, le fonctionnement de notre application n'a pas été modifié du point de vue de l'utilisateur. Nous allons donc mettre en place un système de route avec les deux composants précédents :

- Lorsque l'application démarrera, c'est la liste des Pokémons qui sera affichée par défaut.
- Quand l'utilisateur sélectionnera un Pokémon, il sera redirigé vers la fiche d'information de ce Pokémon.

Pour commencer, nous devons reprendre le template de notre composant racine *app.component.html* et le modifier. Etant donné que ce composant n'a plus comme rôle d'afficher nos Pokémons, voilà à quoi il ressemble désormais :

```
<!-- Notre template a rétréci ! -->  
<router-outlet></router-outlet>
```

Il n'y a qu'une chose à remarquer ici, la balise *<router-outlet>*. Lorsque nous naviguerons dans notre application, le template des composants parcourus sera injecté immédiatement au sein de cette balise. Par exemple, si je me rends sur l'url */pokemons*, alors le template du composant *list-pokemon.component.ts* sera injecté dans la balise *<router-outlet>* du template de *AppComponent*.

Si actuellement vous avez des erreurs dans votre projet, c'est parfaitement normal ! Nous sommes entrain de mettre en place la navigation dans notre application, et tout n'est pas encore correctement branché. Il nous manque encore quelques étapes.

AppComponent est le composant père des deux autres composants de notre application.

Mais alors, on peut mettre le code commun à nos deux composants dans ce composant père ?

Et oui ! Et pour illustrer ce propos, je vous propose d'ajouter une barre de navigation à notre projet. Modifier le template `app.component.html` :

```
<!-- Barre de navigation -->
<nav>
  <div class="nav-wrapper teal">
    <a href="#" class="brand-logo center">
      Application de Pokémons
    </a>
  </div>
</nav>

<!-- Contenu des composants -->
<router-outlet></router-outlet>
```

Nous avons juste ajouté une barre de navigation avec des balises et des classes propres à Materialize. La partie que vous devez retenir est la balise `<router-outlet>` à la ligne 9.

Aussi, nous pouvons mettre à jour la classe `app.component.ts`, car ce composant n'a plus de logique particulière :

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {}
```

Nous avons maintenant trois composants prêts à l'emploi.

Cependant, il nous reste encore trois choses à faire pour mettre en place notre système de navigation :

- Ajouter une balise HTML `<base>` dans le fichier `index.html`.
- Déclarer les routes de notre application dans le fichier `app.routing.ts`.
- Importer ces routes dans le module de notre application `app.module.ts`.

Allez, vous allez bientôt voir les résultats de votre travail !

3.1. La balise

L'élément HTML `<base>` définit l'URL de base à utiliser pour composer toutes les urls relatives contenues dans un document. Vérifier que cette balise est bien présente dans votre balise d'en-tête `<head>` :

```
<head>  
<base href="/">  
<!-- ... -->  
</head>
```

Il ne peut y avoir qu'une seule balise `<base>`.

3.2. Créer les routes

Afin de déclarer les routes de notre application, nous allons créer un module dédié `app-routing.module.ts` dans le dossier `app` :

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ListPokemonComponent } from './list-pokemon.component';
import { DetailPokemonComponent } from './detail-pokemon.component';

// routes
const appRoutes: Routes = [
  { path: 'pokemons', component: ListPokemonComponent },
  { path: 'pokemon/:id', component: DetailPokemonComponent },
  { path: '', redirectTo: 'pokemons', pathMatch: 'full' }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule { }
```

D'abord, on importe deux nouvelles classes : `Routes` et `RouterModule`. Ces deux classes nous aident à déclarer les routes de notre application. On doit donc lancer l'application avec un tableau de routes `appRoutes` que l'on fournit en paramètre de la fonction `RouterModule.forRoot`, à la ligne 15 ci-dessus.

Par défaut, le Router d'une application n'a pas de routes jusqu'à ce qu'on les configure !

On importe également les composants :

- `ListPokemonComponent`,
- et `DetailPokemonComponent` ...

... qui sont utilisés pour construire la navigation.

Ensuite, on déclare une constante `appRoutes` à la ligne 7, qui contient la liste de nos routes sous forme de tableau.

Enfin nous exportons nos routes sous la forme d'un module, que nous pouvons maintenant déclarer dans le module de l'application !

*Heu, ... je n'ai pas tout compris.
Pourquoi nous déclarons nos routes
dans un nouveau module ?*

Excellente question. En fait, il y a deux écoles.

- Vous pouvez définir vos routes sous forme d'une constante de type tableau, et les déclarer directement dans le fichier `app.module.ts` ;
- Appliquer un découpage plus systématique et déclarer les routes dans un module dédié, comme nous l'avons fait.

La seule règle est de ne jamais mélanger les deux manières de faire dans le même projet. Choisissez une seule façon de faire.

Certains développeurs ignorent le module de lorsque la configuration est simple et fusionnent la déclaration des routes directement dans le module principal (par exemple, `AppModule`). C'est vrai que cela a pu vous sembler exagéré de déclarer deux modules pour seulement quelques routes.

Cependant, la deuxième méthode est tout de même la plus recommandée (c'est celle que nous appliquons dans ce cours), car elle permet de mieux découper la structure de votre application, et vous pouvez configurer plus facilement des modules de routes pour chaque fonctionnalité dans votre application.

Et pourquoi nous avons déclaré trois routes, alors que nous n'avons prévu que deux ?

La raison est simple, au démarrage de notre application, l'url appelée est « l'url vide », c'est-à-dire celle qui correspond au simple nom de domaine de votre application : dans mon cas il s'agit de `http://localhost:3000/`, mais le numéro du port peut être différent selon votre machine : 3002, 8080, ...

Comme cette url ne correspond à aucune url de mes composants, nous faisons une redirection vers la route `/pokemons`. Ainsi au démarrage de notre application, nous affichons bien notre liste de Pokémons.

3.3. Déclarer nos routes auprès du module racine

Nous devons déclarer les routes de notre application dans le module racine `app.module.ts` :

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

import { AppComponent } from './app.component';
import { BorderCardDirective } from './border-card.directive';
import { PokemonTypeColorPipe } from './pokemon-type-color.pipe';
import { ListPokemonComponent } from './list-pokemon.component';
import { DetailPokemonComponent } from './detail-pokemon.component';
import { AppRoutingModuleModule } from './app-routing.module';

@NgModule({
  declarations: [
    AppComponent,
    BorderCardDirective,
    PokemonTypeColorPipe,
    ListPokemonComponent,
    DetailPokemonComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModuleModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Nous avons rajouté deux choses dans ce module :

- D'abord nous avons intégré nos routes. À la ligne 9, on importe le module contenant nos routes, et nous les intégrons dans le module à la ligne 21, avec les éléments importés.
- Ensuite nous déclarons nos deux nouveaux composants auprès de notre module, à la ligne 16 et 17. Il faut bien sûr ne pas oublier de les importer avant (ligne 7 et 8).

C'est bon, vous pouvez lancer l'application avec la commande `ng serve`, et admirer le résultat !

4. Gérer les erreurs 404

Notre application doit être en mesure de gérer le cas où la page demandée par l'utilisateur n'existe pas.

Par exemple, si votre internaute rentre une url différente de celles traitées par votre application, comme `http://localhost:3000/poke`, vous aurez l'erreur suivante dans la console du navigateur : `Error: Cannot match any routes: 'poke'`.

En fait, vous aurez cette erreur chaque fois que l'utilisateur se rend sur une url différente de `/pokemons` ou `/pokemon/:id`. Et c'est normal, puisque notre système de navigation n'est pas fait pour intercepter les autres routes. Ce que nous voudrions, c'est intercepter toutes les routes qui ne sont pas gérées par notre application et afficher une page d'erreur à l'utilisateur.

Pour cela, nous allons créer un composant `PageNotFoundComponent` qui aura pour rôle d'afficher un message d'erreur à l'utilisateur. Créez donc ce composant `page-not-found.component.ts` dans le dossier `app` :

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

@Component({
  selector: 'page-404',
  template: `
    <div class="center">
      
      <h1>Hey, cette page n'existe pas !</h1>
      <a [routerLink]="['/pokemons']" class="waves-effect waves-teal btn-flat">
        Retourner à l' accueil
      </a>
    </div>
  `
})
export class PageNotFoundComponent { }
```

Ce composant n'a pas de logique propre, il s'agit juste d'un template qui sera affiché à l'utilisateur.

On utilise la directive `routerLink` à la ligne 10 pour rediriger l'utilisateur vers la page d'accueil.

Il ne nous reste plus qu'à intercepter les routes qui ne sont pas prises en charge par notre application et de les rediriger vers ce composant. Rien n'est plus simple grâce à l'opérateur `***`, qui permet d'intercepter **toutes les routes** au sein de votre application, dans le fichier `app-module.routing.ts` :

```
// ...
import { PageNotFoundComponent } from './page-not-found.component';

const appRoutes: Routes = [
// ...
{ path: '***', component: PageNotFoundComponent }
];
```

Ainsi, toutes les routes qui ne sont pas interceptées par notre système de navigation seront redirigées vers le composant *PageNotFoundComponent*. Pensez donc à déclarer cette route en dernier dans votre tableau de routes, car si vous la mettez en premier, alors toutes les routes de votre application pointeront vers votre page d'erreur ! Ce n'est pas vraiment ce que souhaitent vos utilisateurs !

L'ordre dans lequel vous déclarez vos routes a une importance, les routes déclarées en premier doivent être les plus « précises », et les routes les plus générales à la fin !

Allez, il ne nous reste plus qu'à déclarer ce composant dans notre module racine *app.module.ts* :

```
// ...
import { PageNotFoundComponent } from './page-not-found.component';

@NgModule({
// ...
declarations: [
// ...
PageNotFoundComponent,
```

```
// ...  
],  
// ...  
})  
export class AppModule { }
```

Essayer maintenant de lancer l'application, et entrez une url pouvant provoquer une erreur, comme /404 par exemple. Vous devriez obtenir ceci :



La page d'erreur 404 de notre application.

Aussi, je tiens à m'excuser auprès de tous ceux qui déteste les Pokémons, et qui sont obligés de les supporter partout dans ce cours !

5. Débogguer vos routes

Lorsque vous mettez en place vos routes, il arrive souvent d'avoir quelques erreurs dans votre application. Heureusement, Angular vous permet de débogguer simplement le système de route de votre application, dans le fichier *app-routing.module.ts* :

```
@NgModule({  
  imports: [  
    RouterModule.forRoot(appRoutes, {enableTracing : true})  
  ],  
})
```

Cela vous évitera d'ajouter des *console.log* dans toute votre application, pour voir d'où peut venir l'erreur !

6. Conclusion

Nous avons vu dans ce chapitre comment créer un système de routing dans notre application, en liant deux composants simples. Nous avons également vu comment gérer les erreurs 404, le cas où l'utilisateur souhaite afficher un Pokémon qui n'existe pas, et comment centraliser du code commun entre deux composants fils dans un composant parent.

Mais pour aller vers des applications plus poussées, je vous propose de voir dans le prochain chapitre comment développer une application à plusieurs modules, ce qui nous permettra de mieux organiser le futur code de notre application au fur et à mesure qu'elle se développera. Allez, courage !

En résumé

- Angular simule la navigation de l'utilisateur auprès du navigateur, sans que nous n'ayons rien à faire.
- On construit un système de navigation en associant une url et un composant dans un fichier à part.
- Le système de routes d'Angular interprète les routes qui sont déclarées du haut vers le bas.
- La balise <router-outlet> permet de définir où le template des composants fils sera injecté. Cette balise est disponible dans tous les templates des composants du module racine.
- L'opérateur permettant d'intercepter toutes les routes est **.
- Les routes doivent être regroupées par fonctionnalité au sein de modules.

Chapitre 11 : Les Modules

Nous allons voir dans ce chapitre comment créer un nouveau module pour notre application, afin de mieux organiser le code de notre application. En effet, au fur et à mesure que notre application grandit, si nous rajoutons tous nos fichiers dans le dossier app, on obtiendra rapidement un sacré bazar !

Nous allons donc créer un module consacré uniquement à la gestion des Pokémons dans notre application. En sachant comment ajouter un nouveau module dans votre application, vous serez capable de créer des applications de n'importe quelle taille, vos ambitions n'auront plus de limites.

Ce chapitre prolonge donc le chapitre précédent, en créant un système de navigation plus complexe, avec plusieurs modules.

Du point de vue utilisateur, notre application restera la même que celle du chapitre précédent. Nous allons modifier l'architecture interne de notre application, ce qui fera une grande différence pour vous en tant que développeur.

1. Le module racine et les autres

Au risque de me répéter, nous allons revoir rapidement les fondamentaux sur les modules !

Les applications Angular sont modulaires, et elles possèdent leur propre système de modules. Chaque application possède au minimum un module : le *module racine*, qui est nommé AppModule par convention. Alors que ce module peut suffire pour les petites applications, la plupart des applications ont besoin des *modules de fonctionnalités*.

Les modules de fonctionnalité sont un ensemble de classes et d'éléments, dédiés à un domaine spécifique de votre application.

Quel que soit la nature du module, un module est toujours une classe avec le décorateur `@NgModule`. Ce décorateur prend en paramètre un objet avec des propriétés qui décrivent le module. Les propriétés les plus importantes sont :

- **declarations** : Les classes de vues qui appartiennent à ce module. Angular a trois types de classes de vues : les *composants*, les *directives*, et les *pipes*. Il faut toutes les renseigner dans ce tableau.
- **exports** : Il s'agit d'un sous-ensemble des déclarations, qui doivent être visibles et utilisables dans les templates de composants d'autres modules.
- **imports** : Les classes exportées depuis d'autres modules, dont on a besoin dans le module.
- **providers** : Cette propriété concerne les services et *l'injection de dépendances* que nous traiterons au prochain chapitre.

- **bootstrap** : Cette propriété ne concerne que le module racine. Il faut y renseigner le composant racine, c'est-à-dire le composant qui sera affiché au lancement de l'application.

JavaScript a son propre système de module, qui est complètement différent et sans rapport avec le système de module d'Angular.

En JavaScript, chaque fichier est un module, et tous les objets définis dans ce fichier appartiennent au module. Le module déclare certains objets comme publics en les déclarant avec le mot-clé *export*. Ensuite d'autres modules JavaScript utilisent le mot-clé *import* pour accéder à ces objets. C'est ce mécanisme que l'on utilise lorsque l'on fait :

```
export class AppComponent {}
```

Et par la suite :

```
import { AppComponent } from './app.component';
```

Les systèmes de modules de JavaScript et d'Angular sont différents mais complémentaires. Nous utilisons les deux pour écrire nos applications. Cependant, dans ce chapitre nous étudions bien les modules d'Angular.

2. Créer un module

Nous allons passer tout de suite à la pratique, en créant un nouveau module permettant de centraliser les éléments concernant la gestion des Pokémons de notre application.

La première chose à faire est de créer un dossier *pokemons* pour notre module, dans le dossier *app*.

C'est dans ce dossier que nous placerons tous les éléments relatifs au module : templates, composants, routes, directives, pipes, etc ...

Nous devons maintenant déplacer les éléments que nous avons déjà développés dans le dossier du module *pokemons*, à savoir :

- Le composant qui affiche la liste des Pokémons *list-pokemon.component.ts* et *list-pokemon.component.html*.
- Le composant qui affiche un Pokémon spécifique *detail-pokemon.component.ts* et *detail-pokemon.component.html*.
- Le pipe *pokemon-type-color-type.pipe.ts* que nous avons développé précédemment.
- La directive *border-card.directive.ts*.
- Notre modèle *pokemon.ts* qui représente nos Pokémons dans l'application.
- Les données de l'application issue de *mock-pokemon.ts*.

Mettez donc tous ces fichiers dans le dossier *pokemons*.

Ensuite, nous devons créer le module proprement dit dans un fichier dédié *pokemons.module.ts*, dans le dossier *src/app/pokemons*. C'est ce fichier qui servira à articuler les autres éléments du module :

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

```

import { CommonModule } from '@angular/common';

import { ListPokemonComponent } from './list-pokemon.component';
import { DetailPokemonComponent } from './detail-pokemon.component';
import { BorderCardDirective } from './border-card.directive';
import { PokemonTypeColorPipe } from './pokemon-type-color.pipe';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    ListPokemonComponent,
    DetailPokemonComponent,
    BorderCardDirective,
    PokemonTypeColorPipe
  ],
  providers: []
})
export class PokemonsModule {}

```

Nous devons remarquer plusieurs choses dans ce module :

- Nous n'importons pas le *BrowserModule* mais le *CommonModule* dans le tableau *imports*. En fait, le *BrowserModule* incluait déjà le *CommonModule*, ce qui nous fournissait un certain nombre d'éléments dans les templates de nos composants, comme les directives **ngIf* et **ngFor*, que n'avons jamais eu besoin d'importer au cas par cas. Le *BrowserModule* permet simplement de démarrer l'application dans le navigateur, ce que nous n'avons plus besoin de faire dans nos sous-modules : le *CommonModule* est suffisant.
- Nous déclarons quatre éléments dans le tableau *declarations*. Ce sont des éléments qui n'appartiennent plus au module racine, mais au module spécifique *pokemons*. Nous déclarons les composants, les directives et les pipes propre à ce module.
- Le tableau *providers* est vide. C'est dans ce tableau que l'on peut déclarer des Services propres à ce module. Nous

verrons dans le chapitre suivant ce que sont les Services.

Les routes du module

Il faut maintenant que notre module *pokemon* dispose de ses propres routes, indépendamment des routes définies au niveau global de l'application dans *app-routing.module.ts*. En effet, les modules sont destinés à regrouper des fonctionnalités indépendantes les unes des autres le plus possible, et donc chaque module doit posséder les routes qui lui sont propres, en interne. Créons donc un fichier *pokemons-routing.module.ts* dans le dossier *pokemons* :

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

import { ListPokemonComponent } from './list-pokemon.component';
import { DetailPokemonComponent } from './detail-pokemon.component';

// routes definition
const pokemonsRoutes: Routes = [
  { path: 'pokemons', component: ListPokemonComponent },
  { path: 'pokemon/:id', component: DetailPokemonComponent }
];

@NgModule({
  imports: [
    RouterModule.forChild(pokemonsRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class PokemonRoutingModule { }
```

Il y a une différence par rapport au fichier des routes globales de notre application. On utilisait jusqu'à maintenant la méthode statique *forRoot* pour enregistrer nos routes auprès du Router. Mais là nous définissons les routes du sous-module *pokemons* : on doit donc utiliser la méthode *forChild* pour enregistrer les routes additionnelles.

N'utilisez pas la méthode *forRoot* dans les fichiers de définitions des routes de vos sous-modules, sous peine de lever une erreur !

Le router se chargera de combiner et d'assembler TOUTES les routes de notre application !

Cela nous permet de définir les routes de nos sous-modules sans avoir à modifier les routes principales de notre configuration. Il ne nous reste donc plus qu'à importer ces routes dans notre module *pokemons.module.ts* :

```
// ...
import { PokemonRoutingModule } from './pokemons-routing.module';

@NgModule({
  imports: [
    CommonModule,
    PokemonRoutingModule
  ],
  // ...
})
```

Voilà, notre module *pokemons* est théoriquement prêt !

Comment ça 'théoriquement' ? Notre application ne fonctionne pas ?

Non, pas vraiment ! Rappelez-vous que notre application est composée maintenant de deux modules : *app.module.ts* et *pokemons.module.ts*. Nous venons de développer un module dédié à la gestion de nos Pokémons, mais il nous reste maintenant à mettre à jour le module racine de notre application.

3. Mettre à jour le reste de l'application

Nous devons à présent mettre à jour le module racine de notre application. Voici à quoi ressemble actuellement notre module racine *app.module.ts* :

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

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { AppRoutingModule } from './app-routing.module';

import { PageNotFoundComponent } from './pagenotfound.component';


// Les quatres importations suivantes sont inutiles maintenant !

import { BorderCardDirective } from './shadow-card.directive';

import { PokemonTypeColorPipe } from './pokemon-type-color.pipe';

import { ListPokemonComponent } from './list-pokemon.component';

import { DetailPokemonComponent } from './detail-pokemon.component';


@NgModule({
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    BorderCardDirective, // on a plus besoin de ça ici...
```

```

    PokemonTypeColorPipe, // ... de ça non plus ...

    ListPokemonComponent, // ... ni de ça ...

    DetailPokemonComponent, // ... et pas de ça non plus !

    PageNotFoundComponent
],
bootstrap: [ AppComponent ]
}))

export class AppModule { }

```

Il y a plusieurs éléments dont nous n'avons plus besoin au niveau de ce module, car nous les avons transférés au niveau du module *pokemon*.

En revanche, nous avons besoin de laisser certains éléments au niveau global de notre application. Par exemple le composant *PageNotFoundComponent*, qui est utile dans toute l'application.

Je vous invite donc à modifier notre module racine de la manière suivante :

- Supprimer les lignes commentées ci-dessus qui ne sont plus utiles.
- Importer le module *PokemonsModule* dans le module racine.

Vous devriez obtenir le code suivant pour *app.module.ts* :

```

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

import { PokemonsModule } from './pokemons/pokemons.module';

```

```
import { AppComponent } from './app.component';
import { PageNotFoundComponent } from './page-not-found.component';

@NgModule({
  // L'ordre de chargement des modules est très important
  // par rapport à l'ordre de déclaration des routes !
  imports: [
    BrowserModule,
    PokemonsModule,
    AppRoutingModule // pour l'ordre de déclaration des routes !
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Remarquez l'ordre de déclaration des modules à la ligne 14 et 15. On charge d'abord le sous-module avant le module racine. En effet, l'ordre d'importation des modules ci-dessus détermine directement l'ordre de déclaration des routes. Rappelez-vous que notre module racine contient une route qui intercepte toutes les urls qui ne font pas partie de votre application, afin de rediriger l'utilisateur vers une page 404 personnalisée. Si vous chargez d'abord le module racine, votre application se contentera d'afficher cette page d'erreur à l'utilisateur, quelle que soit la page que ce dernier demande !

Maintenant nous devons mettre à jour le fichier des routes globales *app-routing.module.ts*, qui ne contient plus que deux routes : la route de redirection au démarrage de l'application et la route de gestion des erreurs :

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { PageNotFoundComponent } from './page-not-found.component';

const appRoutes: Routes = [
  { path: '', redirectTo: 'pokemons', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
  ],
  exports: [
    RouterModule
  ]
})
```

```
]
})
export class AppRoutingModule { }
```

Voilà, votre application fonctionne à nouveau, faites le test avec *ng serve -o* !

L'application fait exactement la même chose qu'auparavant, mais notre code est mieux organisé. La réorganisation de notre code a nécessité un certain nombre d'opérations : nous avons dû changer le contenu de certains fichiers, et en créer d'autres. Il aurait été plus simple de définir à l'avance l'architecture de notre application, non ?

C'est ce que nous allons faire maintenant : voir tout de suite comment sera structuré notre application finale, avant de nous lancer dans la suite des développements !

4. Structurer l'architecture de notre application

Pour l'instant notre application comprend deux modules : *AppModule* (le module racine) et *PokemonModule* (le sous-module qui gère les Pokémons).

Je vous propose de réfléchir à l'architecture finale de notre application. Notre application sera un simple gestionnaire de Pokémons. L'utilisateur se connectera à son espace privé, et pourra ensuite modifier les informations des Pokémons, ou les supprimer.

L'espace privée de l'utilisateur sera le module *PokemonModule*, dans lequel nous pourrons ajouter les éléments nécessaires à nos développements au fur et à mesure. Nous ajouterons un composant *LoginComponent* dans le module racine pour permettre à l'utilisateur de se connecter.

Hé, mais l'architecture de notre application est déjà en place du coup, non ? On a juste besoin de rajouter les composants nécessaires dans le bon module, au fur et à mesure ?

Bien vu !

Je souhaitais juste que vous en preniez bien conscience ! Notre application est prête à encaisser les développements futurs sans problème !

Des modules pour factoriser votre code

Angular propose deux modules pour factoriser votre code sur sa documentation officielle, le *SharedModule* et le *CoreModule*. Voyons ensemble chacun de ces deux modules :

Le Shared Module

L'équipe Angular recommande de créer un module nommé *SharedModule* pour centraliser les composants, les directives et les pipes qui seront utilisés dans des composants appartenants à des modules de fonctionnalités. Vous pouvez aussi partager des services statiques, dans le sens où ils ne dépendent pas de l'état de votre application. Par exemple, un service nommé *TextFilterService* à sa place dans le *SharedModule*, mais pas un service nommé *SharedUserData*.

En pratique, à chaque fois que vous vous servez d'un élément mentionné ci-dessus dans au moins deux composants différents, appartenant à des modules de fonctionnalités, factoriser ces éléments dans le *SharedModule*. Par convention, ce module se trouve dans le dossier *app/shared/shared.module.ts*.

Il peut être importé dans tous les modules qui nécessitent des éléments partagés du *SharedModule*.

Le core Module

Le rôle du *CoreModule* est différent du *SharedModule*. Il doit seulement importer les services et les modules de l'application qui peuvent être factorisés. Son rôle est d'alléger le module racine.

Ce module ne doit être importé qu'une seule fois dans le *AppModule*.

Pendant ce temps, dans la vraie vie...

Heu, ce n'est pas un peu lourd d'avoir deux modules de factorisation différents ?

Je suis tout à fait d'accord avec vous, pour notre application, cela ajoutera beaucoup de lourdeur, pour pas grand-chose ! De plus, la plupart des applications professionnelles n'utilisent pas ces deux modules différents, souvent un seul module nommé *CommonModule* suffit.

Rappelez-vous, il s'agit de recommandations de la part de l'équipe d'Angular, pas de règles. De plus, Angular ne verra aucune différence si vous appelez ce module *TotoModule* ou *MachinModule*. Par contre, les autres développeurs qui travaillent avec vous risquent de ne pas être contents !

En tous cas dans la suite de ce cours, je n'utiliserai pas ces modules, car notre application de démonstration est trop petite pour justifier leur utilisation. La documentation officielle est très claire à ce sujet, si votre application est de petite taille, le module racine suffit pour factoriser les éléments globaux de votre application.

Cependant, si vous voulez vous entraîner à la fin de ce cours, en rajoutant de nouvelles fonctionnalités, vous pourrez alors factoriser certains éléments, pensez-y !

5. Conclusion

Nous sommes maintenant capables de créer des applications bien organisées en regroupant nos fonctionnalités en module. Cela nous permet de créer des applications plus importantes qu'auparavant, sans nous emmêler les pinceaux dans notre code.

Il y a cependant une chose que nous n'avons pas encore vue : avec tous ces composants différents que nous allons devoir créer, comment allons-nous factoriser le code présent dans plusieurs composants différents ? C'est ce que nous allons voir dans le chapitre suivant sur les Services !

En résumé

- Il existe deux types de modules : le module racine et les modules de fonctionnalité, appelés également sous-modules.
- On déclare un module avec l'annotation `@NgModule`, quel que soit le type du module.
- On peut créer des applications complexes en ajoutant des modules de fonctionnalité au module racine.
- Chaque module regroupe tous les composants, directives, pipes, services, ... liés au développement d'une fonctionnalité donnée, dans un dossier à part.
- Chaque module peut disposer de ses propres routes également.
- On définit les routes de nos sous-modules avec *forChild*, et *forRoot* pour le module racine.
- Modifier l'architecture d'une application existante peut être pénible, il est préférable de prévoir l'architecture des modules de votre application à l'avance.

Chapitre 12 : Les services et l'injection de dépendances

Maintenant que nous avons une architecture solide pour notre application, je vous propose de commencer à l'enrichir avec des *Services*. Plusieurs de nos composants vont avoir besoin d'accéder et d'effectuer des opérations sur les Pokémons de l'application : nous allons centraliser ces opérations, ainsi que les données concernant les Pokémons.

On va donc créer un service, qui sera utilisable pour tous les composants du module *pokemons*, afin de leur fournir un accès et des méthodes prêtes à l'emploi pour gérer les Pokémons !

1. Créer un service simple

Nous allons créer un service qui s'occupe de fournir des données et des méthodes pour gérer nos Pokémons. L'objectif est de masquer au reste de l'application la façon dont nous récupérons ces données, et le fonctionnement interne des méthodes.

Créons donc un fichier *pokemons.service.ts* dans le dossier de notre module */app/pokemons*.

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

@Injectable({
  providedIn: 'root',
})
export class PokemonsService { }
```

Rassurez-vous, ce service ne fait rien pour l'instant ! Cependant, remarquez le décorateur *@Injectable* appliqué à notre service. Il permet d'indiquer à Angular que ce service peut lui-même avoir d'autres dépendances. Pour l'instant, notre service n'a pas de dépendances, mais il est fortement recommandé d'ajouter le décorateur *@Injectable* systématiquement (en fait, ne vous posez même pas la question).

Il y a aussi une propriété *providedIn*, à laquelle nous avons passé la valeur « root ». Cette propriété permet d'indiquer à Angular que ce sera la même instance du service qui sera fournie à travers toute l'application. Ce mécanisme s'appelle *l'injection de dépendances*, mais nous y reviendrons prochainement.

Il existe une méthode un peu plus ancienne pour fournir un service à travers toute l'application. Cela consistait à passer le service en question à la propriété *Providers* du module racine de votre application. Mais ce n'est plus vraiment d'actualité, car la syntaxe est moins lisible pour le même résultat.

Heu, pourquoi nous n'avons pas eu besoin d'ajouter @Injectable sur nos composants ? Ils ont bien des dépendances eux aussi, non?

C'est une excellente question. En fait la réponse se trouve dans l'annotation `@Component` elle-même, qui s'occupe déjà de ça pour nous ! (de même pour `@Pipe` et `@Directive`).

Je vous propose de créer plusieurs méthodes au sein de ce service :

- Une méthode *getPokemons*, qui renvoie la liste de tous les Pokémons de notre application.
- Une autre méthode *getPokemon*, qui renvoie un Pokémon en fonction de son identifiant.
- Enfin, une dernière méthode *getPokemonTypes*, qui renvoie la liste de tous les types possibles pour un pokémon. (Nous nous servirons de cette méthode dans le chapitre sur les Formulaires, pour permettre à l'utilisateur de sélectionner les types qu'il souhaite attribuer à un pokémon.)

```
import { Injectable } from '@angular/core';
import { Pokemon } from './pokemon';
import { POKEMONS } from './mock-pokemons';

@Injectable()
export class PokemonsService {

  // Retourne tous les pokémons
  getPokemons(): Pokemon[] {
    return POKEMONS;
  }

  // Retourne le pokémon avec l'identifiant passé en paramètre
  getPokemon(id: number) {
    let pokemons = this.getPokemons();
```

```

for(let index = 0; index < pokemons.length; index++) {
  if(id === pokemons[index].id) {
    return pokemons[index];
  }
}

getPokemonTypes(): Array<string> {
  return [
    'Plante', 'Feu', 'Eau', 'Insecte', 'Normal', 'Electrik',
    'Poison', 'Fée', 'Vol', 'Combat', 'Psy'
  ];
}
}

```

Le code des trois nouvelles méthodes est assez explicite, je souligne juste le fait que j'ai dû importer le modèle d'un Pokémon et les données de base, à la ligne 2 et 3. Pour l'instant, nos composants n'utilisent pas encore ce nouveau service. Regardons tout de suite comment injecter ce service dans les composants qui en ont besoin !

2. Utiliser un service

Je vous propose de voir comment utiliser un service en essayant d'injecter notre *PokemonsService* dans un composant quelconque, de façon théorique dans un premier temps.

Pour utiliser un service dans un composant, la première chose à faire est de l'importer :

```
import { PokemonsService } from './pokemons.service';
```

Jusque-là, rien d'anormal. Maintenant, comment récupérer une instance de notre service dans nos composants ? En utilisant le constructeur ?

```
let pokemonsService = new PokemonsService();
```

Ne faite J-A-M-A-I-S ça !

Mais pourquoi ? Qu'est-ce qu'il y a de si terrible, c'est bien comme ça qu'on instancie une classe, non ?

Alors, il y a trois problèmes majeurs :

- D'abord, notre composant devra savoir comment créer le *pokemonsService*. Si nous modifions le constructeur de notre service par la suite, nous devons retrouver chaque composant où nous avons utilisé le service : ce n'est vraiment pas terrible.
- Ensuite, on crée une nouvelle instance du service à chaque fois que l'on utilise le mot-clé *new*. Que se passe-t-il si le service doit mettre en cache des pokémons, et les partager avec tous les composants ? On ne pourrait pas le faire avec

ce système, car nous aurions besoin de la même instance de notre service à travers toute l'application.

- Enfin, lorsque vous développez un service, le consommateur de votre service ne doit pas se demander comment fonctionne le service de l'intérieur. En quelque sorte, il doit s'agir d'une boîte noire, prête à l'emploi.

Mais comment on fait alors, pour utiliser un service dans un composant ?

Rassurez-vous, Angular propose *d'injecter* nos services, plutôt que de nous laisser les instancier nous-mêmes !

Injecter un service

L'injection d'un service dans un composant est très simple car Angular nous propose de le faire en deux lignes seulement :

- Importer la classe du service en question.
- Ajouter un constructeur qui définit une propriété privée.

Reprenons le fichier *list-pokemon.component.ts* et modifiez le constructeur pour injecter notre service :

```
// Pensez à importer le service PokemonsService d'abord :  
import { PokemonsService } from './pokemons.service';  
  
constructor(  
  private router: Router,  
  private pokemonsService: PokemonsService) {}
```

Avec cette simple ligne, nous avons fait beaucoup de choses. Nous avons injecté une instance de *PokemonsService* dans notre composant. Cette instance est désormais disponible sous forme de propriété privée dans notre composant :

```
let pokemonsService: PokemonsService = this.pokemonsService;
```

De plus, étant donné que nous avons utilisé l'injection de dépendance, Angular nous garantit que cette instance est unique à travers toute notre application. Ainsi, si j'injecte ce service dans un autre composant, il s'agira bien de la même instance de ce service. Le fait que l'instance du service soit unique nous permettra d'utiliser les Services comme stockage provisoire des données.

Angular sait désormais fournir l'instance de *pokemonsService* lorsque l'on crée un nouveau *ListPokemonComponent*.

Le constructeur lui-même ne fait rien comme vous pouvez le constatez. Il s'occupe simplement de définir les bons paramètres afin d'injecter le service *PokemonsService* dans notre composant. Souvent, vous aurez besoin d'injecter un service dans un service. Le fonctionnement est le même que pour les composants, il faut utiliser également le constructeur pour injecter le service : on parle du *constructor injection pattern*.

Consommer le service

Nous n'avons plus qu'à utiliser ce service pour injecter des Pokémons dans notre composant ! Modifions la méthode d'initialisation de *list-pokemons.component.ts* :

```
ngOnInit(): void {  
  //this.pokemons = POKEMONS;  
  this.pokemons = this.pokemonsService.getPokemons();  
}
```

On pourrait même découper notre code d'une meilleure façon, de manière à ne pas avoir de 'logique métier' au sein de *ngOnInit*, mais dans une méthode dédiée :

```
ngOnInit(): void {  
  this.getPokemons();  
}  
  
getPokemons(): void {  
  this.pokemons = this.pokemonsService.getPokemons();  
}
```

De cette manière, nous avons réduit le contenu de *ngOnInit* à l'essentiel. Le code parle de lui-même : lorsque le composant est initialisé, on récupère tous les Pokémons.

Et notre composant *detail-pokemon.component.ts* ? Nous allons utiliser le même service, bien sûr. Voici ce que nous devons faire :

- Importer la classe du service *PokemonsService* dans *DetailPokemonComponent*.
- Ajouter le service dans le composant grâce au constructeur.
- Ensuite, nous n'avons plus qu'à adapter le code de notre composant avec la méthode du service.

Voici le code que j'obtiens de mon côté pour le composant *detail-pokemon.component.ts*, j'ai supprimé pas mal de choses qui n'était plus utiles :

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { Pokemon } from './pokemon';
import { PokemonsService } from './pokemons.service';

@Component({
  selector : 'detail-pokemon',
  templateUrl : './detail-pokemon.component.html'
})
export class DetailPokemonComponent implements OnInit {
  pokemon: Pokemon = null;

  constructor(
    private route: ActivatedRoute,
    private router: Router,
    private pokemonsService: PokemonsService) {}

```

```

ngOnInit(): void {
  let id = +this.route.snapshot.paramMap.get('id');
  this.pokemon = this.pokemonsService.getPokemon(id);
}

goBack(): void {
  this.router.navigate(['/pokemons']);
}
}

```

Vous remarquerez que la propriété *pokemons* a été retirée du composant, nous n'en avons plus besoin car nous utilisons directement la méthode *getPokemon* du service !

Nous avons deux composants différents, qui récupèrent les données depuis le même service, ainsi notre accès aux données est bien centralisé. Qu'est-ce qu'il nous faudrait de plus ?

Alors, je suis parfaitement d'accord avec vous ! Mais je vous propose maintenant que nous fassions un peu de théorie concernant *l'injection des dépendances* avec Angular, c'est indispensable pour mieux comprendre la suite. Ce ne sera pas trop long, promis !

3. L'injection de dépendances

Nous allons voir comment fonctionne l'injection de dépendance, c'est-à-dire comment injecter nos services dans nos composants, et comment délimiter un espace dans notre application depuis lequel le service est disponible.

Angular dispose de son propre mécanisme d'injection, et on ne peut pas vraiment développer une application sans cet outil.

On parle souvent de DI (Dependency Injection) pour désigner l'injection de dépendances, et je vais souvent utiliser cet acronyme, ne soyez pas surpris.

Comme nous l'avons vu, il est quasiment obligatoire d'utiliser la DI plutôt que des constructeurs pour gérer nos dépendances.

Qu'est-ce que la DI exactement ?

L'injection de dépendances est un modèle de développement (ou *design pattern* pour nos amis anglo-saxons), dans lequel chaque classe reçoit ses dépendances d'une source externe plutôt qu'en les créant elle-même.

Imaginez que le mécanisme d'injection possède quelque chose appelé un *injecteur*. Dans ce cas, on utilise *l'injecteur* pour gérer les dépendances de nos classes, sans s'occuper nous-même de les gérer.

Angular crée un injecteur à l'échelle de l'application durant le processus de démarrage de l'application :

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

Nous n'avons pas à nous en occuper. En revanche, nous devons enregistrer des *fournisseurs* pour rendre disponible le service là où

nous en avons besoin : au niveau d'un module, de toute l'application ou d'un composant. C'est ce que nous allons voir maintenant !

3.1. Fournir un service à toute l'application

Parfois, nous avons besoin de rendre un service disponible au niveau de l'ensemble de l'application. Pour cela, vous devez fournir votre service au niveau du module racine de votre application, grâce à la propriété *providedIn*, et à l'injecteur racine *root* :

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

@Injectable({
  providedIn: 'root',
})
export class AwesomeService {
  // ...
}
```

Immédiatement, le service *AwesomeService* est disponible dans toute l'application !

3.2. Fournir un service au niveau d'un module

Dans le cas où votre service ne doit être disponible que dans un module, il est inutile de le fournir au niveau de toute l'application. Par exemple, notre service *PokemonService* pourrait être disponible uniquement au niveau du *PokemonsModule*, et non ailleurs dans l'application.

Pour cela, il faut simplement modifier la propriété *providedIn* du service en question, et lui passer le module où on souhaite injecter le service :

```
import { PokemonsModule } from './pokemons.module'

@Injectable({
  providedIn: PokemonsModule,
})
export class AwesomeService {...}
```

C'est tout !

Maintenant, le service fictif *AwesomeService* est disponible uniquement dans le module *PokemonsModule*.

Essayer toujours d'être le plus précis possible dans votre stratégie d'injection de dépendances. Ne fournissez jamais tous vos services au niveau de l'application si cela n'est pas nécessaire.

3.3. Fournir un service au niveau d'un composant

Si vous avez besoin de fournir un service à un seul endroit dans un composant, vous pouvez simplement utiliser la propriété *providers* de l'annotation `@Component` :

```
@Component({  
  selector: 'some-component',  
  template: '...',  
  providers: [AwesomeService]  
})  
export class SomeComponent { }
```

Généralement, on développe un service pour factoriser le comportement entre plusieurs composants. Donc vous ne devriez pas trop avoir à utiliser ce type d'injection.

Bien, en tous cas vous savez maintenant comment créer un service et l'injecter dans vos composants si besoin !

4. Conclusion

Nous avons pu voir comment centraliser le code commun à plusieurs composants dans des services, et les rendre disponibles dans telle ou telle partie de notre application, grâce aux fournisseurs.

Sachez cependant que j'ai été volontairement simpliste et que le système d'injection de dépendance dans Angular est plus complexe que ce que l'on a vu dans ce chapitre (système hiérarchique des injections, fournir d'autres éléments que des classes : strings, objets, fonctions ...), mais nous avons vu l'essentiel pour construire notre application.

En résumé

- Il faut ajouter l'annotation *@Injectable* sur tous nos services.
- Un service permet de factoriser et de centraliser du code qui peut être utile ailleurs dans l'application.
- On utilise *l'injection de dépendances* pour rendre un service disponible dans un composant.
- On ne gère jamais nous-mêmes les dépendances sur un composant ou un service, on passe toujours par l'injection de dépendances.
- L'injection de dépendance permet de garantir que l'instance de notre service est unique à travers toute l'application.
- On définit un fournisseur de service pour déterminer dans quelles zones de notre application notre service sera disponible.

Partie 3 : Aller plus loin avec Angular

Chapitre 13 : Formulaires pilotés par le template

Les formulaires sont omniprésents dans les applications : il y a un formulaire pour la connexion, un formulaire de contact, un formulaire pour modifier telle ou telle donnée, pour mettre à jour son profil, etc.

Cependant, la gestion des formulaires a toujours été complexe. Entre le traitement des données utilisateurs, la validation, l'affichage de messages d'erreurs ou de succès, etc. il faut souvent beaucoup de travail aux développeurs pour offrir une expérience complète et agréable aux utilisateurs. Heureusement, Angular peut nous aider à créer des formulaires, et il va nous faciliter le travail !

1. Introduction

Lorsque vous souhaitez créer un formulaire avec Angular, vous avez la possibilité d'utiliser deux modules différents : **FormsModules** et **ReactiveFormsModule**.

Ces deux modules répondent au même besoin, mais avec une approche différente. Le premier, *FormsModules*, développe une partie importante du formulaire dans le template (on parle de *Template-driven form*), alors que *ReactiveFormsModule* est plus centré sur le développement du formulaire côté composant. Il n'y a pas une méthode mieux qu'une autre dans l'absolu. Cependant, le module *FormsModule* est plus adapté :

- Pour les petits formulaires
- Pour se faire la main en tant que débutant.

Je crois que vous avez compris la suite, je vous présenterai dans ce chapitre la première façon de faire, celle avec *FormsModule* !

Ces deux modules proviennent de la même librairie [@angular/forms](https://angular.io/forms).

Avant d'aborder le vif du sujet, il y a plusieurs éléments que je dois vous présenter :

- Le module *FormsModule*.
- La directive *NgForm*.
- La directive *NgModel*.

1.1. Le module FormsModule

Ce module va nous aider à développer des formulaires : il met notamment à notre disposition les directives *NgForm* et *NgModel* que nous allons utiliser. Nous devons le déclarer dans le module *pokemons* de notre application.

1.2. La directive **NgForm**

À partir du moment où *FormsModule* a été importé dans un module, la directive **NgForm** devient active sur toutes les balises `<form>` de ce module. Nous n'avons pas besoin d'ajouter des sélecteurs supplémentaires dans les templates !

Pour chaque formulaire où elle est appliquée, la directive *NgForm* crée une instance de *FormGroup* au niveau global du formulaire, nous reviendrons plus tard sur le rôle de cet objet. En tout cas, sachez qu'une référence à cette directive nous permet de savoir si le formulaire que remplit l'utilisateur est valide ou non, au fur et à mesure que l'utilisateur complète ce formulaire.

De plus, on peut être notifié lorsque l'utilisateur déclenchera la soumission de formulaire !

1.3. La directive NgModel

Cette directive doit s'appliquer sur chacun des champs du formulaire, et ce pour plusieurs raisons :

- Cette directive créera une instance de *FormControl* pour chaque champ de votre formulaire, comme un *input* ou un *select* par exemple.
- Chaque *FormControl* est une brique élémentaire du formulaire, qui encapsule l'état donné d'un champ. Il a pour rôle de traquer la valeur du champ, les interactions avec l'utilisateur, la validité des données saisies et de garder la vue synchronisée avec les données.
- Chaque *FormControl* doit être défini avec un nom. Pour cela, il suffit d'ajouter l'attribut *name* à la balise HTML associée.
- Lorsque cette directive est utilisée dans au sein d'une balise *<form>*, la directive s'occupe pour nous de l'enregistrer auprès du formulaire comme un élément fils de ce formulaire. En combinant cette directive avec *NgForm*, on peut donc savoir en temps réel si le formulaire est valide ou non !
- On peut aussi utiliser la directive *NgModelGroup* pour créer des sous-groupes de champs, à l'intérieur d'un formulaire.

En plus, la directive *NgModel* s'occupe de mettre en place une **liaison de données bidirectionnelle** pour chacun des champs du formulaire ! Nous aurons souvent besoin de cette liaison bidirectionnelle pour les formulaires : gérer les interactions de l'utilisateur côté template, et traiter les données saisies côté composant.

Et ce n'est pas tout !

La directive s'occupe également d'ajouter et de retirer des classes spécifiques sur chaque champ. Nous pouvons ainsi savoir si

l'utilisateur a cliqué ou non sur un champ, si la valeur du champ a changée, ou s'il est devenu invalide. En fonction de ces informations, nous pourrions changer l'apparence d'un champ, et faire apparaître un message d'erreur, ou de confirmation.

Et quelles sont les classes ajoutées par la directive exactement ?

Bonne question.

Le tableau ci-dessous montre quelles classes sont ajoutées, et à quelles conditions :

État du champ	1	0
La valeur du champ a été visitée au moins une fois, c'est-à-dire que l'utilisateur a cliqué ou effectué une tabulation sur ce champ.	ng-touched	ng-untouched
La valeur du champ a changé par rapport à sa valeur initiale.	ng-dirty	ng-pristine
La valeur est considéré comme correcte par rapport aux règles de validation de ce champ.	ng-valid	ng-invalid

2. Créer un formulaire

Avant de nous lancer « *tête baissée* » dans la création d'un formulaire, essayons de spécifier un peu nos besoins. De quoi avons-nous besoin exactement ?

Notre futur formulaire doit permettre **d'éditer certaines propriétés d'un Pokémon**. En effet, nous n'allons pas modifier l'identifiant ou la date de création d'un Pokémon, puisque par définition ces données ne sont pas appelées à être modifiées. Il nous reste donc les propriétés suivantes :

- Les points de vie du Pokémon.
- Les dégâts du Pokémon.
- Le nom du Pokémon.
- Les types du Pokémon.

Ainsi, notre formulaire comptera quatre champs.

Il sera un composant à part entière, chargé de gérer les données saisies par l'utilisateur, et qui permettra d'éditer un Pokémon. Cependant comme le code du template sera assez complexe, nous allons découper le template et le composant du formulaire dans deux fichiers séparés :

- La classe du composant : *pokemon-form.component.ts*.
- Le template du composant : *pokemon-form.component.html*.

Placez ces deux fichiers dans le dossier du module dédié à la gestion des pokémons : `/app/pokemons/`.

Par convention, il est recommandé d'avoir un fichier dédié pour chaque tâche dans votre application : un fichier par module, un fichier par template, un fichier par formulaire, etc...

2.1. Le composant du formulaire

Voici donc le code de la classe de notre formulaire *pokemon-form.component.ts*, que je vous donne ci-dessous et que je détaillerai ensuite :

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

import { Router } from '@angular/router';

import { PokemonsService } from './pokemons.service';

import { Pokemon } from './pokemon';

@Component({
  selector: 'pokemon-form',
  templateUrl: './pokemon-form.component.html'
})
```

```
export class PokemonFormComponent implements OnInit {
  @Input() pokemon: Pokemon; // Propriété d'entrée du composant
  types: Array<string>; // Types du pokémon : 'Eau', 'Feu', etc.

  constructor(
    private pokemonsService: PokemonsService,
    private router: Router) {}

  ngOnInit() {
    // Initialisation de la propriété types
    this.types = this.pokemonsService.getPokemonTypes();
  }

  // Détermine si le type passé en paramètres appartient ou non
  // au pokémon en cours d'édition.
  hasType(type: string): boolean {
    let index = this.pokemon.types.indexOf(type);
    if (index > -1) return true;
    return false;
  }
}
```



```

}

// Méthode appelée lorsque l'utilisateur ajoute ou retire un type
selectType($event: any, type: string): void {
  let checked = $event.target.checked;
  if (checked) {
    // Si l'utilisateur coche un type, on l'ajoute à la liste
    this.pokemon.types.push(type);
  } else {
    // Si l'utilisateur décoche un type, on le retire de la liste
    let index = this.pokemon.types.indexOf(type);
    if (index > -1) {
      this.pokemon.types.splice(index, 1);
    }
  }
}

```

```

// La méthode appelée lorsque le formulaire est soumis.

onSubmit(): void {

  console.log("Submit form !");

  let link = ['/pokemon', this.pokemon.id];

  this.router.navigate(link);

}

}

```

Alors, je vous dois quelques explications. Prenons les choses les unes après les autres. D'abord, les importations :

```

import { Component, Input, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { PokemonsService } from './pokemons.service';
import { Pokemon } from './pokemon';

```

À la ligne 1, on importe les éléments *Component*, *Input* et *OnInit* depuis la librairie *@angular/core*. Les éléments *Component* et *OnInit*, nous les connaissons déjà.

En revanche, l'élément *Input* est nouveau. Il permet d'écrire une propriété d'entrée pour un composant. C'est-à-dire que si nous définissons ce composant comme fils d'un autre composant, il faudra nécessairement renseigner les valeurs des propriétés d'entrée de ce composant.

Regardez, à la ligne 12, nous utilisons l'annotation *@Input()* pour indiquer à Angular que notre composant a une propriété d'entrée nommée *pokemon*.

Cela signifie que notre composant ne peut pas fonctionner si nous ne lui avons pas passé un Pokémon comme valeur d'entrée. En effet, c'est indispensable pour un formulaire chargé de créer et éditer des Pokémon !

Les composants avec une propriété d'entrée sont souvent les fils d'autres composants, puisqu'ils dépendent d'eux pour récupérer cette valeur d'entrée. Par exemple, notre formulaire sera le fils du composant *EditPokemonComponent* que nous créerons plus tard, qui se chargera de lui transmettre le Pokémon à modifier.

À la ligne 2, on importe *Router*, qui nous permettra de rediriger l'utilisateur après avoir soumis le formulaire. Ensuite on retrouve notre service *PokemonsService* et notre modèle *Pokemon*, que nous connaissons déjà.

Nous définissons ensuite deux propriétés pour ce composant :

```
@Input() pokemon: Pokemon;  
types: Array<string>;
```

La première propriété est la propriété d'entrée du composant, et la deuxième est le tableau *types*, qui doit contenir tous les types de Pokémon disponibles, afin de les afficher dans le formulaire.

Ensuite, nous implémentons quatre méthodes dans ce composant :

- **La méthode `ngOnInit`** : On utilise cette méthode pour initialiser la propriété *types* avec les types de Pokémons récupérées depuis le service *pokemons.service.ts*. Cela nous permettra de construire un champ spécifique où l'utilisateur pourra sélectionner les types du Pokémon qu'il souhaite ajouter ou retirer.
- **La méthode `hasType`** : Cette méthode permet de savoir si le pokémon en cours d'édition possède ou non le type passé en paramètre. Au chargement du formulaire, on peut ainsi pré-cocher les *checkbox* correspondant aux types que possède déjà le Pokémon courant.
- **La méthode `selectType`** : A chaque fois que l'utilisateur sélectionne ou désélectionne un type, le modèle du Pokémon est mis à jour en ajoutant ou en supprimant le type correspondant.
- **La méthode `onSubmit`** : C'est la méthode qui est appelée lorsque le formulaire est soumis, une fois que tous les champs sont considérés comme valides. Concrètement, cette méthode effectue simplement une redirection vers la page de détail du Pokémon. Les changements ont déjà été pris en compte par la directive *NgModel* !

2.2. Le template du formulaire

Maintenant, passons au code du template, que vous trouverez sur la page des ressources du cours.

Ne vous laissez pas impressionner par la quantité importante de code !

Il s'agit majoritairement de code HTML, agrémenté de certaines classes de la librairie *Materialize*, afin de donner un aspect plus sympathique au formulaire. Prenons les éléments les uns après les autres. D'abord à la première ligne :

```
<form
  *ngIf="pokemon"
  (ngSubmit)="onSubmit()"
  #pokemonForm="ngForm">
```

Vous vous demandez peut-être ce que *ngSubmit* apporte ? Eh bien, cela permet de lier l'événement de validation du formulaire à la méthode *onSubmit* du composant, chargé de gérer la soumission du formulaire.

Ensuite, nous utilisons la directive *NgForm*, en l'utilisant pour déclarer une variable de template nommée *pokemonForm*. Cela nous permettra d'avoir accès à l'état de validité du formulaire, ailleurs dans le template. Nous verrons comment faire cela.

Enfin, la directive *NgIf* nous assure qu'un Pokémon a bien été transmis au composant du formulaire. Dans le cas contraire, nous affichons un message d'erreur plutôt que le formulaire !

2.3. Les champs du formulaire

Chaque champ du formulaire est représenté par une ou plusieurs lignes de code. Les quatre premiers champs sont assez similaires, je vais donc vous présenter seulement un champ sur les quatre : le champ *name*.

En revanche, la gestion des types est plus complexe et fera l'objet d'une explication à part.

Voici donc le champ du formulaire permettant d'éditer le nom d'un Pokémon :

```
<!-- Le champ "name" d'un pokémon : -->
<div class="form-group">
  <label for="name">Nom</label>
  <input
    id="name"
    type="text"
    class="form-control"
    name="name"
    [(ngModel)]="pokemon.name"
    #name="ngModel">
</div>
```

Hormis les balises et les attributs que nous connaissons, il y a deux éléments nouveaux.

L'utilisation de la directive *NgModel*. Cette directive permet la **liaison bidirectionnelle** entre le composant et le template. La syntaxe `[()]` est donc la combinaison du *property-binding* (liaison de propriétés pour pousser une valeur du composant vers le template, avec les crochets) et du *event-binding* (liaison d'événement du template vers le composant, avec les parenthèses).

La déclaration de la variable de template *name* à la ligne 6. En lui associant la directive *NgModel*, cette variable nous permet de déclarer le champ sur lequel elle est rattachée comme étant un *FormControl*. Le champ est ainsi rattaché au formulaire global, et

nous pouvons obtenir des informations dessus : Est-il valide ? A-t-il déjà été modifié ?

Pour se souvenir dans quel ordre mettre les crochets et les parenthèses pour NgModel, imaginer simplement que c'est une boîte qui contient une banane : [()].

Le champ qui gère les types d'un Pokémon est un peu plus complexe :

```
1  <!-- Le champ "types" d'un pokémon : -->
2  <form class="form-group">
3  <label for="types">Types</label>
4  <p *ngFor="let type of types">
5    <label>
6      <input type="checkbox"
7        class="filled-in"
8        id="{{ type }}"
9        [value]="type"
10       [checked]="hasType(type)"
11       (change)="selectType($event, type)"/>
12
13   <span [attr.for]="type">
14     <div class="{{ type | pokemonTypeColor }}">
15       {{ type }}
16     </div>
17   </span>
18 </label>
19 </p>
20 </form>
```

Il y a plusieurs choses à relever dans cet extrait de code :

- **Ligne 4** : On utilise la directive *NgFor* pour créer une liste de cases à cocher, avec un type de Pokémon associé à chacune.
- **Ligne 8 et 13** : On utilise le nom du type (du Pokémon), pour lier la balise *<input>* et *<label>*, grâce à l'interpolation et à la liaison de l'attribut *for*.

- **Ligne 9** : On définit le nom du type comme valeur du champ.
- **Ligne 10** : On coche automatiquement la case si le Pokémon possède le type associé à cette case.
- **Ligne 11** : A chaque fois que l'utilisateur coche ou décoche une case, on déclenche un appel à la méthode *selectType*, avec le type qui a été ajouté ou retiré, afin de réajuster le modèle côté composant.
- **Ligne 14** : On applique le pipe *pokemonTypeColor*, pour afficher un petit label de couleur avec le nom du type, à côté de chaque case à cocher.

2.4. Le bouton submit

La dernière chose qui manque à notre formulaire, c'est un bouton pour la validation :

```
<button type="submit"  
class="waves-effect waves-light btn"  
[disabled]="!pokemonForm.form.valid">  
Valider  
</button>
```

Hormis les attributs *type* et *class* que vous connaissez déjà, nous utilisons la liaison de propriété *disabled* pour désactiver le bouton de validation si le formulaire est invalide, ce qui oblige l'utilisateur à avoir correctement rempli tous les champs auparavant !

Mais comment on détermine si le formulaire est valide ou non ?

Très bonne question, c'est ce que nous allons voir tout suite, en ajoutant des règles de validation sur nos champs !

3. Ajouter des règles de validation

Avant de voir comment ajouter des règles de validation, je vous propose de définir quelles restrictions nous souhaitons implémenter, champ par champ :

Champ	Condition(s) de validité	Implémentation proposée
Name	Champ requis, chaîne de caractères de 1 à 25 lettres.	attribut required, pattern= »[AZa-z]{1,25} »
HP	Champ requis, nombre entre 0 et 999.	attribut required, pattern= »[0-9]{1,3} »
CP	Champ requis, nombre entre 0 et 99.	attribut required, pattern= »[0-9]{1,2} »
Types	Champ requis, l'utilisateur doit sélectionner entre 1 et 3 types différents dans une liste donnée.	Validation personnalisé avec la méthode <code>isTypesValid</code> du composant.

Les quatre premières règles de validation peuvent être implémentées facilement grâce aux nouveaux attributs apportés par HTML 5 : *required* pour rendre un champ obligatoire, et *pattern* qui permet de définir une expression régulière pour valider un champ.

Voici comment implémenter ces règles de validation pour le champ *name*, à la ligne 5 :

```
<!-- Pokemon name -->
```

```

<div class="form-group">
<label for="name">Nom</label>
<input type="text" class="form-control" id="name"
  pattern="^[a-zA-Z0-9àâãäåçéêëëïîĩñóòôöùûüýÿæœ]{1,25}$"
  required
  [(ngModel)]="pokemon.name" name="name"
  #name="ngModel">
</div>

```

Nous avons simplement ajouté l'attribut *required* et *pattern*, avec la bonne expression régulière.

Vous devez faire la même chose pour les trois autres champs : *pictures*, *hp* et *cp*. Il suffit d'ajouter la bonne expression régulière dans le champ correspondant. Reprenez le tableau ci-dessus et ajoutez les règles de validation, je vous laisse le faire tout(e) seul(e), ce n'est pas bien méchant !

Maintenant, regardons comment implémenter un validateur personnalisé pour les types des *pokémons*. Nous devons limiter leur nombre entre un et trois d'après les règles de validation que nous avons établi. Commençons par développer cette méthode côté composant :

```

// Valide le nombre de types pour chaque pokémon (entre 1 et 3)
isTypesValid(type: string): boolean {

  // Le pokémon a un seul type, qui correspond au type passé en paramètre.
  // Dans ce cas on renvoie false, car l'utilisateur ne doit pas pouvoir décocher ce type (sinon
  // le pokémon aurait 0 type, ce qui est interdit)
  if (this.pokemon.types.length === 1 && this.hasType(type)) {
    return false;
  }

  // Le pokémon a au moins 3 types.
  // Dans ce cas il faut empêcher à l'utilisateur de cocher un nouveau type, mais pas de
  // décocher les types existants.
  if (this.pokemon.types.length >= 3 && !this.hasType(type)) {
    return false;
  }

  // Après avoir passé les deux tests ci-dessus, on renvoie true, c'est à dire que l'on
  // autorise l'utilisateur à cocher ou décocher un nouveau type.

```

```
return true;
}
```

Cette méthode s'occupe de renvoyer un booléen, pour savoir si une case à cocher doit être verrouillée ou non :

- Si l'utilisateur a sélectionné une seule case, il faut l'empêcher de pouvoir désélectionner cette case. Sinon il pourrait choisir de ne renseigner aucun type pour un pokémon, et ce n'est pas ce que nous voulons.
- Si l'utilisateur a déjà sélectionné trois cases, alors il faut l'empêcher de pouvoir sélectionner d'autres cases, mais il doit pouvoir désélectionner les types déjà présents s'il souhaite modifier son pokémon !

On fait aussi appel à la méthode `hasType` pour vérifier que nous ne verrouillons pas des cases que l'utilisateur a déjà cochées, pour lui permettre de les désélectionner.

Ensuite pour verrouiller les cases à cocher de la liste, il faut lier le résultat de la méthode `isTypesValid` à la propriété `disabled`, grâce à la liaison de propriété (ligne 10 ci-dessous) :

```
<!-- Pokemon types -->
<div class="form-group">
  <label for="types">Types</label>
  <div *ngFor="let type of types" class="row">
    <input type="checkbox"
      class="filled-in"
      id="{{ type }}"
      [value]="type"
      [checked]="hasType(type)"
      [disabled]="!isTypesValid(type)"
      (change)="selectType($event, type)"
    />
    <label [attr.for]="type">
      <span class="{{ type | pokemonTypeColor }}">{{ type }}</span>
    </label>
  </div>
</div>
```

Lorsque la méthode *isTypesValid* renvoie *false*, alors la case à cocher est verrouillée et ne peut plus être sélectionnée ! C'est exactement le comportement souhaité !

Bien sûr, la validation côté client n'est jamais suffisante si vous sauvegardez vos données sur un serveur distant. Pensez à valider les données aussi côté serveur, sous peine d'introduire d'importantes failles de sécurité dans votre application !

4. Prévenir l'utilisateur en cas d'erreurs

Pour l'instant, si le formulaire contient des erreurs, le bouton *submit* restera verrouillé et inutilisable : l'utilisateur verra bien que quelque chose ne va pas dans le formulaire, mais il ne saura pas pourquoi !

Nous devons prévenir l'utilisateur sur ce qui ne fonctionne pas. Pour cela, nous allons **utiliser les classes ajoutées automatiquement par la directive NgModel**, pour faire deux choses :

- Utiliser ces classes avec le CSS, en ajoutant une bordure verte ou rouge en fonction de la validité des champs du formulaire : cela permettra à l'utilisateur d'avoir un retour visuel sur l'état de validité d'un champ.
- Utiliser ces classes avec les variables de templates et la directive *NgIf*, pour afficher un message en cas d'erreur sur un champ.

4.1. Ajouter des indicateurs visuels

Nous devons simplement ajouter une feuille style qui s'appliquera uniquement sur le composant du formulaire, ce qui est le comportement par défaut d'un composant Angular grâce au Shadow DOM.

Afin de respecter le principe de *1 tâche = 1 fichier*, nous allons créer une feuille de style à part. Créez un fichier nommé *pokemon-form.component.css* (à placer dans le dossier du module des Pokémons) :

```
.ng-valid[required], .ng-valid.required {  
  border-left: 5px solid #42A948; /* bordure verte */  
}  
  
.ng-invalid:not(form) {  
  border-left: 5px solid #a94442; /* bordure rouge */  
}
```

Que fait ce code ? Il s'occupe de deux choses :

- Il affiche une bordure verte à gauche, si un champ requis est valide.
- Sinon, il affiche une bordure rouge pour les éléments invalides qui ne sont pas des éléments de type *form* : c'est-à-dire tous les champs inputs, select, etc.

Ensuite, nous devons lier le composant de notre formulaire à cette nouvelle feuille de style :

```
@Component({  
  selector: 'pokemon-form',  
  templateUrl: './pokemon-form.component.html',  
  styleUrls: ['./pokemon-form.component.css']  
})  
export class PokemonFormComponent implements OnInit { ... }
```

Comme vous le constatez, nous utilisons l'attribut *styleUrls*, afin de renseigner un tableau de chemins relatifs vers des feuilles de styles, qui seront appliquées sur le template du composant.

Il y a bien un 's' à l'option *styleUrls*, contrairement à *templateUrl*. En effet, il est possible de passer plusieurs feuilles de styles à un composant, mais un seul template. Logique !

On aurait pu également utiliser la propriété *styles* qui permet d'écrire le CSS directement dans le fichier qui contient le composant :

```
@Component({
  selector: 'pokemon-form',
  templateUrl: 'app/pokemons/pokemon-form.component.html',
  styles: [
    .ng-valid[required], .ng-valid.required {
      border-left: 5px solid #42A948;
    }
    .ng-invalid:not(form) {
      border-left: 5px solid #a94442;
    }
  ]
})
export class PokemonFormComponent implements OnInit { ... }
```

Cependant, comme pour les templates, si votre code est trop long, cela n'est pas très pratique. De plus, vos feuilles de styles sont souvent communes à plusieurs composants, il est donc préférable de les centraliser dans des fichiers à part.

4.2. Afficher des messages d'erreurs

L'affichage des messages d'erreurs pour les champs `<input>` peut se faire de la manière suivante : nous combinons la liaison de la propriété `hidden` avec la variable de template `name` :

```
<!-- Pokemon name -->

<div class="form-group">

<label for="name">Nom</label>

<input type="text" class="form-control" id="name"

    pattern="[a-zA-Z0-9áâãäåçèéêëìíîïñóôõöúûüýÿæœ]{1,25}"

    required

    [(ngModel)]="pokemon.name" name="name"

    #name="ngModel">
```

```
<!-- error -->
<div [hidden]="name.valid || name.pristine"
    class="card-panel red accent-1">
    Le nom du pokémon est requis (1-25).
</div>
</div>
```

Regardez à la ligne 11, le message d'erreur est masqué si la propriété `name` est valide, ou qu'elle n'a jamais été modifiée. En effet, on ne va pas afficher un message d'erreur à l'utilisateur lorsqu'on affiche le formulaire pour la première fois, sous prétexte qu'il y a encore des champs vides !

Le principe est le même pour les champs *Picture*, *Cp* et *Hp*. Vous êtes tout à fait capable de les ajouter vous-même à ce niveau. Il vous suffit d'ajouter les attributs avec les valeurs énumérées dans le tableau un peu plus haut.

L'interface que nous allons développer pour les types de Pokémon (avec les cases à cocher qui se verrouillent et déverrouillent automatiquement en fonction d'un certain état de validité) est assez ergonomique pour que l'utilisateur comprenne tout seul les règles de validation, nous n'avons pas besoin d'afficher un message spécifique en plus, cela surchargerait inutilement l'interface utilisateur.

5. Intégration du formulaire

Avant de pouvoir utiliser notre formulaire, il faut l'intégrer dans l'application existante. La première chose à faire est de développer un composant parent pour le formulaire, que nous appellerons *edit-pokemon.component.ts* :

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Pokemon } from './pokemon';
import { PokemonsService } from './pokemons.service';

@Component({
  selector: 'edit-pokemon',
  template: `
    <h2 class="header center">Editer {{ pokemon?.name }}</h2>
    <p class="center">
      <img *ngIf="pokemon" [src]="pokemon.picture">
    <p>
    <pokemon-form [pokemon]="pokemon"></pokemon-form>
  `,
})
export class EditPokemonComponent implements OnInit {
  pokemon: Pokemon = null;

  constructor(
    private route: ActivatedRoute,
    private pokemonsService: PokemonsService) {}

  ngOnInit(): void {
    let id = +this.route.snapshot.params['id'];
    this.pokemon = this.pokemonsService.getPokemon(id);
  }
}
```

Ce composant ne possède rien de spécial, à part à la ligne 13, dans le template.

Il s'agit du composant de notre formulaire, que nous injectons dans ce template ! La syntaxe entre crochets indique une **liaison de propriété sur la propriété d'entrée du composant**. A droite, on passe le Pokémon de la propriété *d'EditPokemonComponent*. Il

s'agit du Pokémon qui doit être édité, et que l'on récupère depuis la méthode *ngOnInit*.

Nous pouvons maintenant utiliser ce composant *EditPokemonComponent* pour afficher le formulaire, avec comme url : */pokemon/edit/:id*.

Le point d'interrogation dans `{{ pokemon?.name }}` signifie « Affiche le nom du pokémon, seulement si le pokémon existe ». Cela évite de lever une erreur inutilement si l'identifiant passé en paramètre de l'url ne correspond à aucun pokémon. De plus, on a appliqué la directive *NgIf* sur l'image d'un pokémon, afin d'éviter de faire une requête « dans le vide », si le pokémon n'a pas encore été chargé.

Pourquoi ne pas avoir développé le formulaire directement dans le composant EditPokemonComponent ?

Il aurait été possible de le faire, mais laissez-moi vous rappeler deux choses :

- Nous pourrions par la suite réutiliser ce formulaire pour gérer le cas d'ajout de Pokémon dans l'application, par exemple. Il est donc logique d'avoir développé le formulaire à part.
- Essayez toujours de respecter le principe de *1 tâche = 1 fichier*. Cela s'applique aussi pour les formulaires !

Nous voilà avec un composant prêt à l'emploi que nous pouvons ajouter n'importe où dans l'application, pour permettre à l'utilisateur d'éditer ses Pokémon !

Nous devons désormais déclarer plusieurs éléments dans le module Pokémon *pokemons.module.ts* :

- *FormsModule*, le module permettant de créer des formulaires ;
- Le composant de notre formulaire *PokemonFormComponent* ;
- Le composant *EditPokemonComponent*.

```
// ...
import { FormsModule } from '@angular/forms';
import { EditPokemonComponent } from './edit-pokemon.component';
import { PokemonFormComponent } from './pokemon-form.component';
// ...

@NgModule({
  imports: [
    CommonModule,
    FormsModule, // nouvelle déclaration
    pokemonsRouting
  ],
  declarations: [
    ListPokemonComponent,
    DetailPokemonComponent,
    EditPokemonComponent, // nouvelle déclaration
    PokemonFormComponent, // nouvelle déclaration
    BorderCardDirective,
    PokemonTypeColorPipe
  ],
  providers: [PokemonsService]
})
export class PokemonsModule {}
```

Nous devons ensuite déclarer une nouvelle route pour l'édition d'un pokémon. Modifier donc le fichier *pokemons-routing.module.ts* en ajoutant la route suivante :

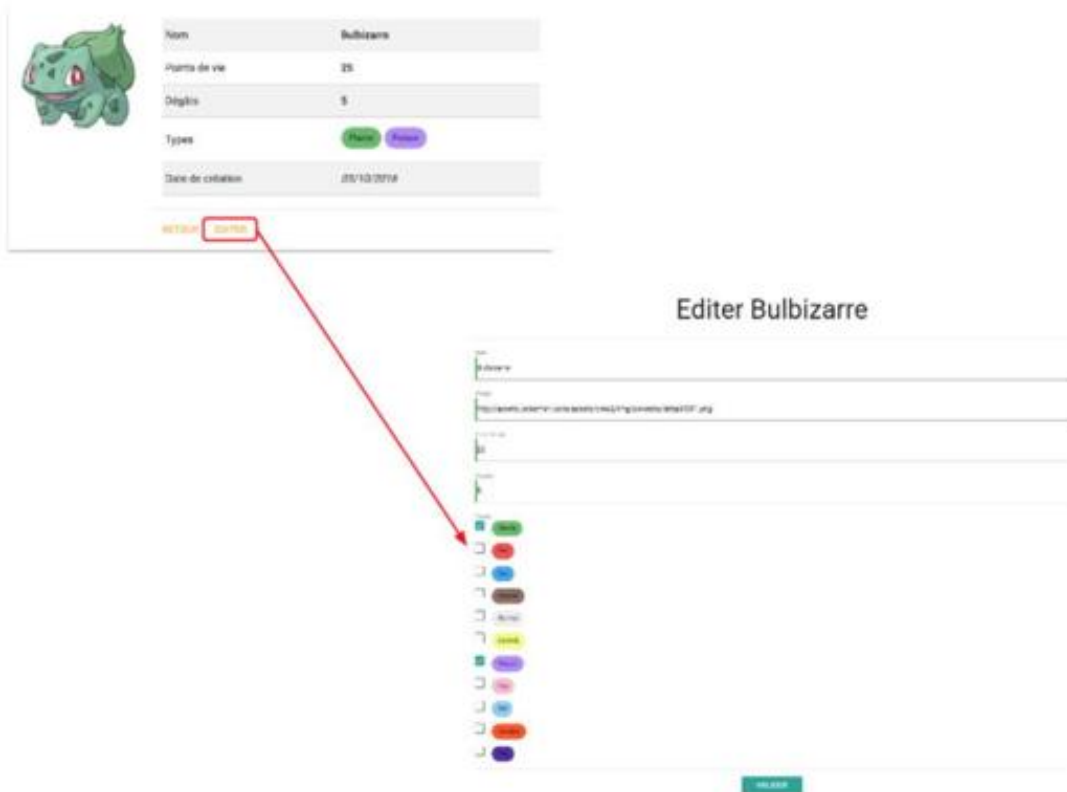
```
import { EditPokemonComponent } from './edit-pokemon.component';
// ...

const pokemonsRoutes: Routes = [
  { path: 'pokemons', component: ListPokemonComponent },
  // ajouter la route d'édition
  { path: 'pokemon/edit/:id', component: EditPokemonComponent },
  { path: 'pokemon/:id', component: DetailPokemonComponent }
];
// ...
```

Rappelez-vous que l'ordre de déclaration des routes à une importance !

Il ne nous reste plus qu'une seule chose à faire : ajouter un lien vers la page d'édition d'un Pokémon. L'endroit le plus approprié pour ce lien semble être la page de détail d'un Pokémon. Ouvrez donc le template *detail-pokemon.component.html* et ajoutez la ligne suivante :

```
<!-- ... -->
<div class="card-action">
  <a (click)="goBack()">Retour</a>
  <!-- On rajoute un lien vers la page d'édition de ce pokémon -->
  <a (click)="goEdit(pokemon)">Editer</a>
</div>
<!-- ... -->
```



Le formulaire d'édition des pokémons est bien intégré à l'application.

On ajoute ensuite la méthode *goEdit* dans la classe du composant *detail-pokemon.component.ts* :

```
// On crée une méthode qui s'occupe de la redirection  
goEdit(pokemon: Pokemon): void {  
  let link = ['/pokemon/edit', pokemon.id];  
  this.router.navigate(link);  
}
```

Et voilà ! Maintenant, lorsque l'utilisateur cliquera sur le lien *Editer* , il sera redirigé vers le formulaire d'édition, déjà pré-rempli avec les valeurs du Pokémon sur lequel il a cliqué !

6. Conclusion

Les formulaires ne sont jamais simples à développer, il faut mettre en place des processus de traitements et de validation des données. Même si Angular nous simplifie la tâche, il nous a fallu quand même un peu de travail !

Avez-vous remarqué qu'à chaque fois que l'on recharge notre application dans le navigateur, les données concernant le pokémons sont réinitialisées, et que nos changements ne sont pas pris en compte ? Nous verrons d'ici la fin de la partie 3 comment persister nos modifications sur un serveur distant, afin de donner une « mémoire » à notre application.

En résumé

- Il y a deux modules différents pour développer des formulaires avec Angular: *FormsModule* et *ReactiveFormsModule*.
- Le *FormsModule* est pratique pour développer des formulaires de petites tailles, et met à disposition les directives *NgForm* et *NgModel*.
- La directive *NgModel* ajoute et retire certaines classes au champ sur lequel elle s'applique. Ces classes peuvent être utilisées pour afficher des messages d'erreurs ou de succès, et des indicateurs visuels.
- La syntaxe à retenir pour utiliser *NgModel* est `[()]`.
- On peut utiliser les attributs HTML5 pour gérer la validation côté client, comme *required* ou *pattern*.
- On peut utiliser des validateurs personnalisés en développant ses propres méthodes de validation.
- Il faut toujours effectuer une validation côté serveur en complément de la validation côté client, si vous avez prévu de stocker des données depuis votre application.

Chapitre 14 : La programmation réactive

Il y a un élément indispensable à la plupart des applications, que nous n'avons pas encore abordé : comment communiquer avec un serveur distant ?

Comme pour les formulaires, il y a plusieurs manières de faire des appels sur le réseau : avec les *Promesses* ou avec les *Observables* et la *programmation réactive*.

Nous verrons quels sont les outils existants pour interagir avec un serveur distant, et comment ils fonctionnent, en attendant de le faire pour de « vrai » dans le chapitre prochain. Allez, au boulot !

Ce chapitre est théorique et les extraits de code sont donnés à titre d'exemple. Nous ne modifierons pas notre application de Pokémon dans ce chapitre !

1. Tenir ses promesses

Les promesses sont natives en JavaScript depuis l'arrivée d'ES6. Ce n'est plus un objet interne à Angular comme c'était le cas dans la version 1.x, et donc nous pouvons les utiliser sans avoir besoin de nouvelles importations.

Nous avons déjà traité des promesses dans le chapitre sur ES6, mais nous allons quand même faire un petit rappel.

Les promesses sont là pour essayer de simplifier la programmation asynchrone. La **programmation asynchrone** désigne un mode de fonctionnement dans lequel les opérations sont non-bloquantes. Concrètement, cela signifie que votre utilisateur peut continuer à utiliser votre application web (navigation, formulaires, ...) sans que le site soit bloqué dès que vous faites un appel au serveur.

On peut utiliser des fonctions de callbacks pour gérer les appels asynchrones, mais les promesses sont plus pratiques.

En fait, lorsque vous créez une promesse (avec la classe *Promise*), vous lui associez implicitement une méthode *then* qui prend deux arguments : une *callback de succès* et une *callback d'erreur*. Ainsi, lorsque la promesse a réussi, c'est la callback de succès qui est appelée, et en cas d'erreur, c'est la callback d'erreur qui est invoquée.

Voici un exemple d'une promesse qui récupère un utilisateur depuis un serveur distant, à partir de son identifiant :

```
let recupererUtilisateur = function(idUtilisateur) {  
  return new Promise(function(resolve, reject) {  
    // appel asynchrone au serveur pour récupérer  
    // les informations d'un utilisateur...  
    // à partir de la réponse du serveur,
```

```

// on extrait les données de l'utilisateur :

let utilisateur = response.data.utilisateur;

if(response.status === 200) {
  resolve(utilisateur);
} else {
  reject('Cet utilisateur n'existe pas !');
}
})
}

```

Dans cet extrait, la fonction *recupererUtilisateur* prend en paramètre l'identifiant d'un utilisateur, et renvoie une promesse qui contient l'objet utilisateur correspondant. En revanche, en cas d'erreur lors de l'appel, la promesse renvoie un message d'erreur.

Et voilà, vous venez de créer une fonction qui renvoie une promesse, avec les informations du serveur ! Vous pouvez ensuite utiliser la méthode *then* dans votre projet :

```

recupererUtilisateur(idUtilisateur)

.then(function (utilisateur) {

  console.log(utilisateur); // en cas de succès

  this.utilisateur = utilisateur; // en cas de succès

}, function (error) {

  console.log(error); // en cas d'erreur

});

```

Ce code est fonctionnel, mais il n'est pas très lisible, vous ne trouvez pas ?

On peut améliorer cette lisibilité avec les « *fonctions fléchées* », que nous avons également vues dans le chapitre sur ES6. Nous les utilisons beaucoup avec la programmation asynchrone, car elles permettent de remplacer les fonctions anonymes, qui sont omniprésentes dans les appels asynchrones en JavaScript, avec une syntaxe plus élégante.

Voici le même code que précédemment, mais avec l'utilisation des fonctions fléchées :

```
recupererUtilisateur(idUtilisateur)
  .then(utilisateur => {
    console.log(utilisateur); // en cas de succès
    this.utilisateur = utilisateur; // en cas de succès
  }, error => console.log(error); // en cas d'erreur
);
```

Bon d'accord, j'ai bien compris les promesses. Mais quel est le rapport avec le titre du chapitre : la programmation réactive ?

Les promesses peuvent couvrir beaucoup de cas d'appels asynchrones, la gestion des événements, etc. Mais les promesses ont aussi leurs limites, surtout quand il faut gérer un grand nombre de requêtes dans un délai très court. Comment gérer proprement plusieurs événements en même temps, sans saturer notre code de promesses dans tous les sens ? La réponse est la programmation réactive.

2. La programmation réactive

Un véritable cours sur Angular ne peut pas passer à côté de la **programmation réactive**. Plus qu'une librairie ou un effet de mode, la programmation réactive est une nouvelle manière d'appréhender la programmation asynchrone... et c'est peut-être l'avenir des applications modernes, vous n'auriez pas tort de parier dessus !

La programmation réactive n'est pas quelque chose de nouveau, c'est simplement une façon différente de concevoir une application. L'idée est de considérer les interactions qui se déroulent dans l'application comme des événements, sur lesquels on peut effectuer des opérations : regroupements, filtrages, combinaisons, etc.

Ainsi, les événements, comme les clics de souris, deviennent des **flux d'événements asynchrones** auxquels on peut **s'abonner**, pour ensuite pouvoir y réagir. Et il est possible de créer des flux à partir de tout et n'importe quoi, pas seulement des clics ! Prenons des exemples concrets. Il est très fréquent de devoir réagir à des événements :

- côté navigateur, en ajoutant des déclencheurs selon les interactions de l'utilisateur.
- côté serveur, en traitant des requêtes à la base de données ou à des services tiers.

Bref, dans la programmation réactive, toutes ces séquences d'événements sont appelées des flux. Retenez ceci :

Programmation réactive = Programmation avec des flux de données asynchrones.

De manière générale, tous ces événements sont poussés par un *producteur de données*, vers un *consommateur*. Notre rôle en tant que développeur sera de définir des **écouteurs d'événements**

(consommateur), sous forme de fonctions, pour réagir aux **différents flux** (producteurs de données). Les écouteurs d'événements sont nommées des *Observer* et le flux lui-même est le sujet observé, on parle d'*Observable*.

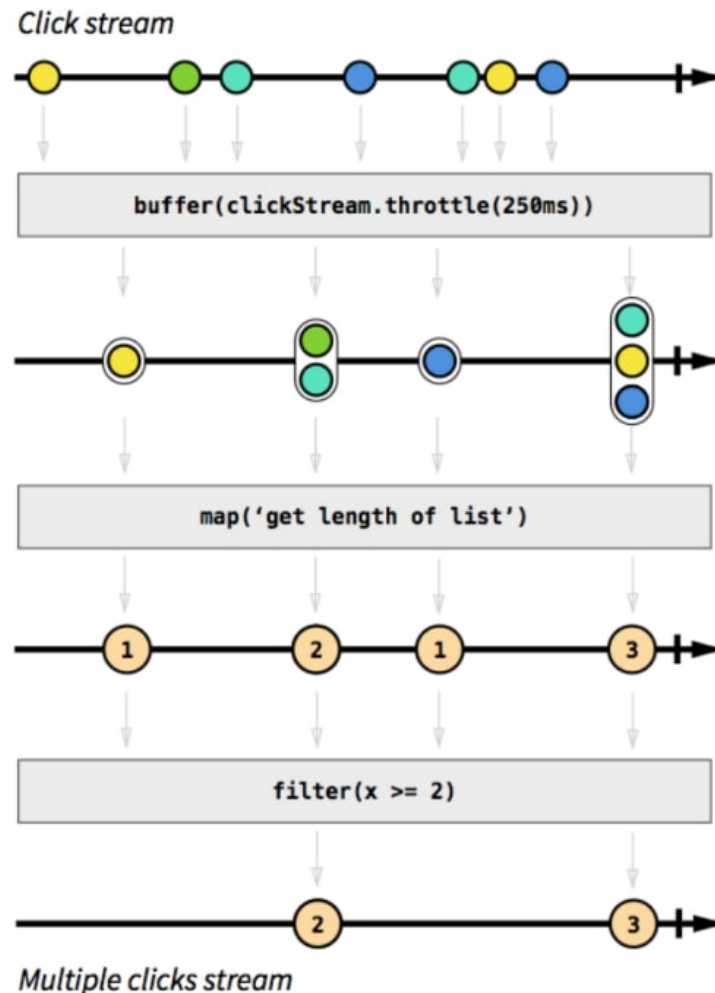
Lorsqu'on s'abonne à un flux pour capter ses événements, on dit que l'on s'inscrit au flux.

2.1. Qu'est-ce qu'un « flux » ?

Comme nous venons de le voir, un flux est une séquence d'événements en cours, qui sont ordonnés dans le temps. Si on observe un utilisateur qui clique plusieurs fois sur un bouton (pour une raison quelconque), la succession des clics peut être modélisée comme un flux d'événements.

Ce qui est intéressant, c'est que l'on peut appliquer des opérations sur ce flux d'événements.

Prenons un exemple : on souhaite détecter les doubles clics de l'utilisateur, et ignorer les clics simples. Pour cela, nous allons considérer qu'il y a un double clic s'il y a moins de 250 ms d'écarts entre deux clics. Nous pouvons modéliser notre problème grâce au schéma ci-dessous :



Les boîtes grises sur le schéma représentent des fonctions qui permettent de transformer un flux en un autre flux modifié. En tout, il y a trois opérations qui sont appliquées à partir du flux initial :

- D'abord, on regroupe les clics qui sont séparés par un délai inférieur à 250 millisecondes. C'est ce que fait la fonction *throttle*. Même si nous n'avons pas vu cette fonction, pas de panique, il s'agit juste de comprendre qu'elle permet de transformer un flux initial en un nouveau flux, selon des critères donnés.
- On dispose maintenant d'un flux de clics regroupés. On applique dessus la fonction *map* pour transformer chaque

groupe en un entier, qui correspond à la longueur de la liste. Par exemple, un groupe de deux clics vaut maintenant la valeur « 2 », tout simplement.

- Pour finir, il ne nous reste plus qu'à filtrer les groupes dont la valeur est supérieure à deux. En effet, on considère comme un double clic tous les groupes de plus de deux clics. On applique donc un filtre sur le flux précédent avec la fonction *filter(x >= 2)*.

Ça y est ! En seulement 3 opérations, on a obtenu le flux souhaité ! On peut maintenant s'abonner à ce flux et réagir aux événements comme on le souhaite ! Et cet exemple est juste la partie visible de l'iceberg :

- Le flux sur lequel nous venons de travailler était simple, mais les opérations utilisées peuvent aussi bien s'appliquer sur d'autres flux, comme par exemple un flux de réponses d'un serveur distant.
- Et il existe bien sûr plus d'opérations que juste celles que nous venons de voir !

Si vous voulez voir quelles autres opérations sont disponibles pour les flux, je vous recommande le site rxmarbles.com, qui vous permet en plus de comprendre en un coup d'œil le fonctionnement de telle ou telle opération.

2.2. Traitement des flux

On peut faire plus que simplement *s'abonner* à un flux. En fait, les flux peuvent émettre trois types de réponses différentes. Pour chaque type de réponses, on peut définir une fonction à exécuter :

- Une fonction pour traiter les différentes valeurs de la réponse : un nombre, des tableaux, des objets, etc.
- Une fonction pour traiter le cas d'erreur.
- Une fonction pour traiter le signal de « fin ». Cela signifie simplement que le flux est « *terminé* », et qu'il n'émettra plus d'événements.

Ce qu'il faut retenir, c'est que les événements du flux représentent soit les valeurs de la réponse (en cas de succès), soit des erreurs, ou des terminaisons.

3. La librairie RxJS

Pour le moment, ce que nous avons vu est très théorique. Vous vous demandez sûrement à quoi cela ressemble, concrètement. En fait, pour faciliter l'implémentation de la programmation réactive, on utilise souvent des librairies spécifiques.

La bibliothèque la plus populaire pour la programmation réactive, dans l'écosystème JavaScript, est RxJS. Et c'est également celle qui a été choisie par l'équipe de développement d'Angular !

3.1. Les Observables

Dans RxJS, un *flux d'événements* est représenté par un objet appelé un **Observable**.

Les Observables sont très similaires à des tableaux. Comme eux, ils contiennent une collection de valeurs. Un Observable ajoute juste la notion de valeur reportée dans le temps : dans un tableau, toutes les valeurs sont disponibles immédiatement. Dans un observable en revanche, les valeurs viendront au fur et à mesure, plus tard dans le temps.

On peut traiter un Observable avec des opérateurs similaires à ceux des tableaux. C'est exactement ce que nous avons fait lorsque nous avons appliqué des opérations sur les flux. Par exemple :

- *take(n)* : cette fonction va récupérer les n premiers éléments d'un flux et se débarrasser des autres.
- *map(fn)* : applique la fonction passée en paramètre sur chaque événement et retourne le résultat. Précédemment, on a appliqué cette méthode aux flux pour récupérer le nombre d'éléments dans un groupe de clics.
- *filter(predicat)* : permet de filtrer seulement les événements qui répondent positivement au prédicat passé en paramètre.
- *merge(s1, s2)* : permet de fusionner deux flux ! Et oui, c'est possible.
- *subscribe(fn)* : applique la fonction passée en paramètre à chaque événement reçu du flux. C'est cette fonction que nous utiliserons le plus souvent. D'ailleurs, cette méthode accepte aussi une deuxième fonction en paramètre, consacrée à la gestion des erreurs. Et lorsque le flux est « *terminé* », il enverra un événement de terminaison que l'on peut détecter avec une troisième fonction !
- et bien d'autres...

Pour illustrer le fonctionnement, imaginons un Observable, dans lequel nous remplaçons les événements par des nombres afin de simplifier la compréhension :

```
Observable.fromArray([1, 2, 3, 4, 5])  
  
.filter(x => x > 2) // 3, 4, 5  
  
.map(x => x * 2) // 6, 8, 10  
  
.subscribe(x => console.log(x)); // Affiche le résultat !
```

C'est exactement le même fonctionnement que pour les flux ! Un observable est une simple collection asynchrone, dont les événements arrivent au cours du temps.

Et rappelez-vous, on peut construire des observables depuis une requête AJAX, un événement du navigateur, une réponse de Web-Socket, une promesse, etc. Bref, tout ce qui est asynchrone !

3.2. Observable ou Promesse ?

Les observables sont différents des promesses, même s'ils y ressemblent par certains aspects, car ils gèrent tous deux des valeurs asynchrones. Mais un observable n'est pas quelque chose à usage unique : il continuera d'émettre des événements jusqu'à ce qu'il émette un événement de terminaison ou que l'on se désabonne de lui. Alors qu'une promesse ne peut gérer qu'un seul événement à la fois, et qu'une fois émis, il n'est plus annulable via une promesse.

Globalement, l'utilisation des promesses est plus simple, et dans de nombreux cas elles sont suffisantes pour répondre aux besoins de votre application. Par exemple, pour récupérer des données depuis un serveur distant et les afficher à l'utilisateur, l'utilisation d'un observable n'est pas forcément nécessaire.

Dans le chapitre suivant, nous verrons deux exemples concrets, qui illustrent quand est-ce qu'il est préférable de choisir un observable plutôt qu'une promesse.

Enfin, sachez qu'il est possible de transformer un Observable en une Promesse très simplement, avec la méthode *toPromise* de RxJS :

```
import 'rxjs/add/operator/toPromise';

function giveMePromiseFromObservable() {
  return Observable.fromArray([1, 2, 3, 4, 5])
    .filter(x => x > 2) // 3, 4, 5
    .map(x => x * 2) // 6, 8, 10
    .toPromise();
}
```

Ensuite, plutôt que d'appliquer la méthode *subscribe*, propre aux observables, vous pourrez utiliser la méthode *then* !

Pour le moment, les Observables ne font pas partie de la spécification ECMAScript officielle, mais peut-être dans une version future. Des efforts sont faits dans ce sens.

4. Conclusion

La programmation réactive peut sembler un peu compliquée à appréhender au début, mais elle permet d'élever le niveau d'abstraction de votre code, et au final vous devrez moins vous soucier de certains détails d'implémentation.

De plus, les applications web évoluent sans cesse et sont maintenant de plus en plus dynamiques : l'édition d'un formulaire peut déclencher automatiquement une sauvegarde sur un serveur distant, un simple commentaire peut se répercuter directement sur l'écran de tous les utilisateurs connectés, etc. En tant que développeurs, il nous faut des outils capables de gérer tout ça, et la programmation réactive en fait partie.

Dans le prochain chapitre, nous verrons comment utiliser le module HTTP fourni par Angular, pour récupérer des données d'un serveur distant. Nous verrons également un cas concret où les observables nous seront utiles : un champ de recherche avec auto-complétion !

En résumé

- Les promesses sont natives en JavaScript depuis l'arrivée de la norme ES6.
- La programmation réactive implique de gérer des flux de données asynchrones.
- Un flux est une séquence d'événements ordonnés dans le temps.
- On peut appliquer différentes opérations sur les flux : regroupements, filtrages, troncatures, etc.
- Un flux peut émettre trois types de réponses : la valeur associée à un événement, une erreur, ou un point de terminaison pour mettre fin au flux.
- La librairie RxJS est la librairie la plus populaire pour implémenter la programmation réactive en JavaScript.
- Dans RxJS, les flux d'événements sont représentés par un objet appelé Observable.

Chapitre 15 : Effectuer des requêtes HTTP standards

Après un chapitre plutôt théorique sur la programmation réactive, nous allons maintenant mettre en pratique nos connaissances pour améliorer notre application de Pokémons ! En effet, pour le moment, cette application est assez simple et permet seulement d'éditer des Pokémons.

Ce que nous aimerions faire, c'est communiquer avec un serveur distant pour récupérer les Pokémons, les éditer et sauvegarder les changements sur le serveur, en ajouter ou en supprimer. Bref, que vous sachiez effectuer toutes les opérations http standards depuis votre application Angular. C'est parti ?

Vous apercevrez le terme API plusieurs fois à partir de maintenant. Une API est une interface de programmation. C'est ce qui vous permet de communiquer avec un service distant depuis votre application. Par exemple, pour stocker les données sur un serveur distant de manière durable.

1. Mettre en place le module `HttpClientModule`

Le module *HttpClientModule* permet de faire communiquer votre application Angular avec un serveur distant, via le protocole HTTP.

Il ne s'agit pas d'un module de base d'Angular, et il faut l'importer depuis la librairie `@angular/common/http`. Le fichier *systemjs.config.js* est déjà configuré pour charger cette librairie, on peut donc l'importer depuis n'importe quel module de l'application. Nous allons le déclarer dans la liste *imports* du module racine *app.module.ts* :

```
// ... on importe le module HttpClientModule :
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    // ...
    HttpClientModule, // Ajoutez le module dans la liste 'imports'
    // ...
  ],
  // ...
})
export class AppModule { }
```

Voilà, le module est désormais disponible partout dans l'application !

2. Simuler une API Web

Jusqu'à maintenant, nous stockions et récupérions nos données depuis le service *PokemonsService*, en utilisant le fait que l'instance de ce service soit unique à travers toute l'application. Seulement ce système ne nous convient plus, et nous voulons pouvoir communiquer avec un serveur distant désormais !

Malheureusement, nous ne disposons pas d'un serveur pour gérer les requêtes de notre application de Pokémons. D'ailleurs, il faudrait même suivre un autre cours, consacré exclusivement à cette tâche !

Mais alors, comment va-t-on faire ?

Mais non, mais non ! En fait, Angular nous permet de simuler une API qui fonctionnera comme si nous utilisions un serveur distant ! Cette simulation est possible grâce au module *InMemoryWebApiModule* de la librairie *angular-in-memory-web-api*. Nous allons voir comment cela fonctionne.

D'abord, installez cette librairie avec la commande suivante :

```
npm install angular-in-memory-web-api --save-dev
```

Je vous recommande fortement de couper la commande `ng serve` en appuyant sur CTRL+C, avant de lancer l'installation de cette librairie.

L'option `--save-dev` permet d'installer cette librairie en tant que dépendance de développement. Comme nous simulons une API Web, nous en aurons besoin que lors de nos développements normalement. (Cependant, dans le cadre de notre application de démonstration, nous utiliserons notre API simulée même lors de déploiement).

Ensuite, créez un fichier nommé *in-memory-data.service.ts*, dans le dossier *app*, et ajoutez-y le code suivant :

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
import { Injectable } from '@angular/core';
import { POKEMONS } from './pokemons/mock-pokemons';

@Injectable({
  providedIn: 'root',
})
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    let pokemons = POKEMONS;
    return { pokemons };
  }
}
```

Contrairement aux apparences, ce service fait plus que de simplement renvoyer une liste de Pokémons.

La classe *InMemoryDataService* implémente l'interface *InMemoryDbService*, qui nécessite d'implémenter la méthode *createDb*. Cette méthode permet de simuler une petite base de données et une API pour notre application.

Cette API met en place plusieurs points de terminaisons sur lesquels on peut effectuer des requêtes :

```
// Exemple avec une collection de 'pokémons'
GET api/pokemons // renvoie tous les pokémons
GET api/pokemons/1 // renvoie le pokémon avec l'identifiant 1
PUT api/pokemons/1 // modifie le pokémon avec l'identifiant 1

// Cette requête retourne les pokémons dont le nom
// commence par 'exp'.
// '^exp' est une expression régulière
GET api/pokemons?name=^exp
```

Bien pratique n'est-ce pas ? Il est quasiment possible de faire toutes les requêtes possibles pour une petite application de démonstration : ajout, suppression, modification, recherche, etc. (Nous verrons la recherche dans le chapitre suivant)

Cette méthode peut servir pour des applications de tests ou de démonstrations. Bien sûr, pour une vraie application vous utiliserez plutôt un serveur distant.

Nous allons voir dans la suite de ce chapitre comment utiliser cette API plutôt que les données du fichier *mock-pokemons.ts*.

Il ne nous reste plus qu'à déclarer notre API simulée auprès du reste de l'application. Pour cela, il est recommandé d'enregistrer les services globaux à toute l'application dans la liste *imports* du module racine *app.module.ts* :

```
// ...  
  
// Importations pour charger et configurer l'API simulée.  
  
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';  
import { InMemoryDataService } from './in-memory-data.service';  
  
// ...
```

```
@NgModule({  
  imports: [  
    // ...  
    HttpClientModule,  
    HttpClientInMemoryWebApiModule.forRoot(InMemoryDataService, {  
      dataEncapsulation: false }),  
  ],  
  // ...  
})
```

La méthode de configuration *forRoot* prend en paramètre la classe *InMemoryDataService*, qui apprête la base de données en mémoire avec les données du service, à la ligne 11.

Voilà, notre API est prête à être utilisée, au boulot !

Le module *HttpClientInMemoryWebApiModule* intercepte les requêtes HTTP et retourne les réponses simulées du serveur. Ainsi, même s'il s'agit d'une API simulée, nous allons interagir avec cette API comme avec n'importe quel service distant. C'est exactement le même fonctionnement pour nous, côté Angular. Retirez l'importation quand un « vrai » serveur est prêt à recevoir vos requêtes.

L'option *dataEncapsulation* permet de préciser le format des données renvoyées par l'API. Les données peuvent être encapsulées dans un objet avec une clef *data* :

```
{
  data : {
    // vos données seront ici si l'option est à true.
  }
}
```

Dans notre cas, laissez cette option à *false*.

3. Mettre à jour notre service

Ça y est, le grand jour est arrivé, nous allons modifier notre service *PokemonsService* pour effectuer des requêtes sur notre API !

Nous allons réécrire les deux méthodes du service *PokemonsService*, afin qu'elles appellent désormais notre API :

- La méthode *getPokemons* qui retourne tous les Pokémons.
- La méthode *getPokemon(id)* qui retourne le Pokémon avec l'identifiant *id*.

La méthode *getPokemonTypes* reste inchangée, puisqu'elle n'interagit pas avec notre API.

Commençons par préparer notre service aux appels vers notre API. Nous allons avoir besoin de plusieurs éléments nouveaux :

- La classe *HttpClient*, pour effectuer des requêtes ;
- La classe *HttpHeaders*, pour pouvoir modifier les en-têtes de nos requêtes.
- La classe *Observable*, nous utiliserons des Observables pour chacune de nos requêtes.
- Les opérateurs *tap*, *of* et *catchError* de la librairie RxJS. Nous allons en avoir besoin lors de nos appels à l'API.

Ajoutez donc ces nouvelles importations dans le fichier *pokemons.service.ts*, et injectez la classe *HttpClient* dans le constructeur de notre service :

```
// l'ancienne importation
```

```
import { POKEMONS } from './mock-pokemons';

// les nouvelles importations dont nous allons avoir besoin.

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable, of } from 'rxjs';

import { catchError, tap } from 'rxjs/operators';
```

```
// le point d'accès à notre API
private pokemonsUrl = 'api/pokemons';

// le constructeur
constructor(private http: HttpClient) { }
```

J'ai également ajouté une nouvelle propriété privée *pokemonsUrl*, afin de déclarer le point d'accès vers notre API à un seul endroit. Si jamais ce chemin devez changer, nous pourrions le modifier simplement sans devoir réécrire toute nos requêtes. (Pour rappel, ce point de terminaison est généré automatiquement pour notre API, grâce au service *InMemoryDbService*.)

Maintenant, nous pouvons développer des méthodes pour appeler notre API ! Modifions tout de suite la méthode *getPokemons* comme ceci :

```
// Avant (avec 'mock-pokemons.ts')
getPokemons(): Pokemon[] {
  return POKEMONS;
}

// Après (avec une requête Http)
getPokemons(): Observable<Pokemon[]> {
  return this.http.get<Pokemon[]>(this.pokemonsUrl).pipe(
    tap(_ => this.log(`fetched pokemons`)),
    catchError(this.handleError('getPokemons', []))
  );
}
```

En recopiant ce code vous aurez des erreurs dans un premier temps, car nous devons terminer nos développements lors des prochaines étapes. Pas d'inquiétudes.

On retourne toujours les mêmes Pokémons, mais on le fait différemment. Regardons ligne par ligne ce que fait cette nouvelle méthode. D'abord la signature a changé :

```
getPokemons() : Observable<Pokemon[]> {...}
```

La méthode *getPokemons* retourne désormais un Observable, qui contiendra un tableau de Pokémons. Ensuite :

```
return this.http.get<Pokemon[]>(this.pokemonsUrl).pipe(...);
```

La méthode *http.get* retourne un Observable, qui s'occupe d'envoyer une requête HTTP de type GET sur la route '*api/pokemons*'. Nous pouvons également typer directement les données de retours grâce à la syntaxe :

```
// L'observable retournera un tableau d'objet Pokemon  
get<Pokemon[]>
```

Puis nous effectuons immédiatement deux opérations sur cet Observable grâce aux opérateurs *tap* et *catchError* :

- L'opérateur *tap* : Il permet d'interagir sur le déroulement des événements générés par l'Observable, en exécutant une action quelconque. Vous pouvez utiliser cet opérateur pour le débogage ou l'archivage des logs par exemple. Dans notre cas, on affiche un petit message dans la console, pour rappeler la méthode qui a été appelée.
- L'opérateur *catchError* : sans surprise, cette méthode permet intercepter les erreurs éventuelles. On appelle alors la méthode *handleError*, que je vous présenterai tout de suite après.

De plus, le module *HttpClientModule* s'occupe pour nous de fournir toutes les réponses au format JSON.

Portez une attention particulière à la forme des données renvoyées par le serveur. Notre API renvoie directement les données dans un objet. Mais une autre API peut renvoyer ses données sous un autre format, à l'intérieur d'une propriété 'data', par exemple ! Soyez donc attentif à cela et ajustez votre code en fonction de l'API que vous utilisez.

En attendant, les composants qui vont utiliser ce service n'ont pas à savoir que nous récupérons les données d'une API ou d'un fichier interne à l'application. Et c'est bien le but : **déléguer l'accès aux données dans un service tiers** ! En revanche, nous devons ajuster les composants car désormais notre service renvoie des Observables.

Et quelle est cette méthode log appelée avec l'opérateur tap ?

C'est une petite méthode très simple, que voici :

```
private log(log : string) {  
  console.info(log);  
}
```

Ajoutez cette méthode dans le service *pokemons.service.ts*. Cela permet de centraliser la gestion des logs de notre service. Si plus tard vous souhaitez archiver vos logs, plutôt que de simplement les afficher dans la console, il vous suffira de modifier cette méthode.

3.1. Gestion des erreurs

Nous devons également gérer les cas d'erreurs éventuelles, dans le cas où la requête n'aboutirait pas. C'est le rôle de l'opérateur *catchError* :

```
catchError(this.handleError('getPokemons', []))
```

C'est une étape importante, nous devons anticiper les erreurs HTTP qui pourraient se produire (pour des raisons indépendantes de notre volonté, bien sûr).

C'est à ça que va servir la méthode `handleError` ?

Oui ! Vous devez ajouter cette méthode dédiée à la gestion des erreurs dans le service *pokemons.service.ts* :

```
private handleError<T>(operation='operation', result?: T){  
  return (error :any): Observable<T> => {  
    console.log(error);  
    console.log(` ${operation} failed: ${error.message}`);  
  
    // Je donne des explications sur "of" un peu plus tard.  
    return of(result as T);  
  }  
}
```

La syntaxe `<T>` en Typescript désigne un type : number, string...

Comme notre application est une simple démonstration, on se contente d'afficher une erreur dans la console. Dans le cas d'une application en production, vous pourrez bien sûr mettre en place un système plus élaboré !

Si vous ne comprenez pas exactement ce code, voici quelques éléments qui pourront vous aider :

- Le paramètre '*operation*' est le nom de la méthode qui a causé l'erreur. Par défaut, ce paramètre vaut '*operation*'.
- Le paramètre '*results*' est une donnée facultative, à renvoyer comme résultat de l'Observable.
- L'instruction *return*, à la ligne 6, permet de laisser notre application fonctionnée en renvoyant un résultat adapté à la méthode qui a levé l'erreur.

En effet, chaque méthode du service renvoie un Observable dont le résultat a un type différent : un tableau, ou un objet Pokémon dans notre cas. La méthode *handleError* prend un paramètre de *type* (le fameux *T* de TypeScript), afin que cette méthode puisse renvoyer une valeur sûre pour chaque méthode, c'est-à-dire le type attendu par cette méthode. Ainsi, l'application peut continuer à fonctionner même si une erreur survient.

L'opérateur *of* de RxJS

Nous utilisons l'opérateur *of* dans la méthode *handleError* :

```
return of(result as T);
```

Cet opérateur permet de convertir n'importe quel argument passé en paramètre en un Observable (un tableau, une chaîne de caractères, etc.). Par exemple :

```
return of([])
```

Cet extrait de code renvoie un simple tableau vide, mais sous forme d'Observable.

C'est bien pratique pour renvoyer un résultat vide, sans lever une erreur dans votre application.

3.2. Récupérer un Pokémon à partir d'un identifiant

Nous devons également mettre à jour la méthode permettant de récupérer un unique Pokémon à partir de son identifiant :

```
getPokemon(id: number): Observable<Pokemon> {  
  const url = `${this.pokemonsUrl}/${id}`; // syntaxe ES6  
  
  return this.http.get<Pokemon>(url).pipe(  
    tap(_ => this.log(`fetched pokemon id=${id}`)),  
    catchError(this.handleError<Pokemon>(`getPokemon id=${id}`))  
  );  
}
```

Comme vous pouvez le constater, cette méthode est très similaire à celle que nous venons de voir. La seule différence est l'url utilisée et le type de retour. Cette url retourne un seul Pokémon, et non un tableau de plusieurs Pokémon.

3.3. Utiliser le service mis à jour

Le service *PokemonsService* renvoie désormais des Observables, et à ce titre nous devons mettre à jour les composants qui consomment ce service, à savoir :

- *list-pokemon.component.ts* :

```
// AVANT
getPokemons(): void {
  this.pokemons = this.pokemonsService.getPokemons();
}

// APRES
getPokemons(): void {
  this.pokemonsService.getPokemons().subscribe(pokemons => this.pokemons =
pokemons);
}
```

detail-pokemon.component.ts & *edit-pokemon.component.ts* (cette portion de code est identique pour les deux fichiers) :

```
// AVANT
ngOnInit(): void {
  let id = +this.route.snapshot.params['id'];
  this.pokemon = this.pokemonsService.getPokemon(id);
}

// APRES
ngOnInit(): void {
  let id = +this.route.snapshot.params['id'];
  this.pokemonsService.getPokemon(id).subscribe(pokemon => this.pokemon =
pokemon);
}
```

Vous pouvez essayer de démarrer l'application avec *npm start* dès maintenant, et contempler les Pokémons issues de votre API !

Si vous ne l'avez pas encore fait, vous pouvez supprimer l'importation de la constante `POKEMONS` dans le fichier `pokemons.service.ts`. En effet, cette importation est devenue inutile, car notre service charge les Pokémons depuis l'API simulée désormais.

4. Modifier un pokémon

Essayez d'éditer un Pokémon depuis l'application. Modifiez quelques-unes de ces caractéristiques, et cliquez sur le bouton « valider ». Vous êtes alors redirigé vers la page avec les informations du Pokémon, et ... toutes les modifications ont disparu, elles n'ont pas été prise en compte !

Mais pourquoi cela ne fonctionne plus ?

Eh bien, quand notre application utilisait une liste de données issue d'un fichier statique, les modifications étaient répercutées directement aux Pokémon de cette liste unique, commune à toute l'application. Mais maintenant que nous extrayons les données à partir d'une API, si nous voulons persister nos changements, nous aurons besoin de les écrire sur le serveur, c'est-à-dire de faire une requête HTTP !

4.1. Une méthode de modification

Nous avons besoin d'une nouvelle méthode dans notre service *PokemonsService*, pour persister les modifications faites depuis le formulaire d'édition.

La structure de la requête pour modifier un Pokémon est similaire aux deux autres requêtes précédentes, bien que nous allons utiliser une requête de type PUT (c'est le type de requête utilisé pour modifier une ressource) afin de persister les changements côté serveur :

```
1  ...
2  // La méthode updatePokemon persiste les modifications
3  // du pokémon via l'API :
4  updatePokemon(pokemon: Pokemon): Observable<Pokemon> {
5    const httpOptions = {
6      headers: new HttpHeaders({
7        'Content-Type': 'application/json'
8      })
9    };
10
11    return this.http.put(
12      this.pokemonsUrl,
13      pokemon,
14      httpOptions
15    ).pipe(
16      tap(_ => this.log(`updated pokemon id=${pokemon.id}`)),
17      catchError(this.handleError<any>('updatePokemon'))
18    );
19  }
20  ...
```

Il y a quelques éléments à signaler dans cette méthode :

- **L'en-tête de la requête** (ligne 5) : On déclare un en-tête pour signaler que le format du corps de la requête sera du JSON.

On ajoute ensuite cet en-tête à la requête Http à la ligne 9.

- **Le point d'accès à l'API** (ligne 12) : Les modifications s'appliquent sur un Pokémon particulier, il faut donc renseigner l'url de l'API, le pokémon sur lequel on souhaite effectuer une modification, ainsi qu'une en-tête pour signaler que le format de la requête sera du format JSON.
- **Le traitement de la requête** (ligne 16 et 17) : On effectue les traitements « habituels » avec les opérateurs *tap* et *catchError*. Mais vous connaissez déjà !

Chaque requête HTTP contient un en-tête et un corps. Il est possible d'ajuster les deux en fonction des besoins de votre application.

4.2. Sauvegarder les données

Bien, nous avons ajouté une méthode permettant de persister les modifications effectuées sur un Pokémon, il faut maintenant nous en servir ! Pour cela, nous allons modifier la méthode qui gère la soumission du formulaire d'édition des Pokémon, dans *pokemon-form.component.ts* :

```
// La méthode appelée lorsque le formulaire est soumis pour la validation
onSubmit(): void {
  this.pokemonsService.updatePokemon(this.pokemon).subscribe(() => this.goBack());
}

// J'ai découpé la redirection dans une méthode dédiée goBack
goBack() : void {
  let link = ['/pokemon', this.pokemon.id];
  this.router.navigate(link);
}
```

La méthode *onSubmit* persiste les modifications effectuées sur le pokémon, en utilisant la méthode *updatePokemon* que nous venons de développer. Ensuite l'utilisateur est redirigé vers la page détaillant ce Pokémon, avec les dernières modifications prises en compte, grâce à la méthode *goBack*.

Rafraîchissez le navigateur et réessayez de modifier un Pokémon. Les changements réalisés via le formulaire devraient maintenant être persistés !

Hourra !

Bien sûr, si vous redémarrez ou rafraîchissez l'application, la base de données sera réinitialisée avec les données par défaut ! Pour persister les données entre deux sessions, il faudrait être en possession d'un serveur distant, dédié à la sauvegarde.

5. Supprimer un pokémon

Je vous propose d'ajouter une nouvelle fonctionnalité permettant de supprimer un Pokémon. Voici comment je vois les choses :



On ajoute la possibilité pour l'utilisateur de supprimer un pokémon.

C'est le composant `detail-pokemon.component.ts` qui va implémenter cette nouvelle fonctionnalité. Mais pour commencer, nous devons ajouter une méthode pour supprimer un Pokémon dans notre service de gestion des Pokémon `pokemons.service.ts` :

```
// La méthode de suppression d'un poékmon
deletePokemon(pokemon : Pokemon): Observable<Pokemon> {
  const url = `${this.pokemonsUrl}/${pokemon.id}`;
  const httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };

  return this.http.delete<Pokemon>(url, httpOptions).pipe(
    tap(_ => this.log(`deleted pokemon id=${pokemon.id}`)),
    catchError(this.handleError<Pokemon>('deletePokemon'))
  );
}
```

Ensuite nous devons ajouter une méthode de suppression dans le composant *detail-pokemon.component.ts* :

```
// Nouvelle méthode de suppression d'un Pokémon
delete(pokemon: Pokemon): void {
  this.pokemonsService.deletePokemon(pokemon).subscribe(
    _ => this.goBack()
  );
}
```

Enfin, il ne nous reste plus qu'à mettre à jour notre interface en conséquence. Repérer la balise avec la classe *card-action* dans le template *pokemon-detail.component.html*. C'est le code HTML qui correspond aux boutons permettant d'agir sur un Pokémon. Voici le nouveau code HTML :

```
<div class="card-action">
  <a class="waves-effect waves-light btn" (click)="goBack()">Retour</a>
  <a class="waves-effect waves-light btn" (click)="goEdit(pokemon)">Editer</a>
  <a (click)="delete(pokemon)">Supprimer</a>
</div>
```

On a ajouté un nouveau bouton de suppression à la ligne 4, et on a également ajouté des classes pour mettre en avant les boutons *Retour* et *Editer*, afin d'éviter que l'utilisateur ne supprime un Pokémon par erreur.

Bon, et si maintenant on ajoutait un bouton permettant d'ajouter un nouveau Pokémon dans notre application ?

Si vous êtes motivé, vous pouvez ajouter une page ou un message de confirmation lorsque l'utilisateur clique sur le bouton Supprimer. Cela permettra d'éviter les suppressions « par accident ».

6. Ajouter un Pokémon

Vous vous en doutiez sûrement, il reste une dernière opération à voir : celle d'ajouter un Pokémon. En effet, à force de supprimer des Pokémon, nous commençons à ne plus en avoir dans notre application.

Pour cette dernière fonctionnalité, nous allons avoir un peu plus de développement à faire, car nous devons adapter le formulaire d'édition d'un Pokémon, afin de le réutiliser pour ajouter un Pokémon.

Ajouter un Pokémon

Type:

Name:

Image:

Pour le jeu :

Niveau :

Typer :

☐ Plaine

☐ Feu

☐ Eau

☐ Sol

☒ Normal

☐ Électrik

☐ Poison

☐ Glace

☐ Vent

☐ Combat

☐ Dragon

VALLER

On va pouvoir ajouter de nouveaux pokémons dans notre application !

Voici donc le « plan de bataille » que je vous propose :

- Commencer par ajouter une méthode permettant d'ajouter un Pokémon sur notre *API simulée*, ce sera déjà ça de fait.
- Créer un composant *AddPokemon* pour gérer l'ajout d'un nouveau Pokémon.
- Adapter notre modèle *pokemon.ts* pour l'ajout d'un pokémon grâce à un constructeur TypeScript, permettant de déclarer des valeurs par défaut pour un nouveau Pokémon.
- Déclarer ce nouveau composant auprès du module des Pokémon, et du module de gestion des routes.
- Adapter notre formulaire d'édition d'un Pokémon pour l'ajout.
- Ajouter un lien sur la page qui liste les Pokémon, pour rediriger l'utilisateur vers notre le formulaire d'ajout.

6.1. Ajouter une méthode POST

Pour ajouter un nouveau Pokémon dans notre API, nous allons avoir besoin d'une requête http de type POST. C'est le type de requête dédiée à l'ajout de nouveaux éléments. Bien sûr, le module *HttpClient* prend en charge ce cas. Ajoutez donc la fonction *addPokemon* dans notre service *pokemons.service.ts* :

```
/** POST pokemon */
addPokemon(pokemon: Pokemon): Observable<Pokemon> {
  const httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };

  return this.http.post<Pokemon>(this.pokemonsUrl, pokemon, httpOptions).pipe(
    tap((pokemon: Pokemon) => this.log(`added pokemon with id=${pokemon.id}`)),
    catchError(this.handleError<Pokemon>('addPokemon'))
  );
}
```

Le code d'ajout est très similaire à ce qu'on a déjà vu, à la différence près qu'ici nous utilisons requête de type POST. Bon, ça c'est fait. Passons maintenant à la suite !

7. Conclusion

Nous avons bien avancé dans ce chapitre. Nous savons maintenant comment mettre en place toutes les opérations http standards dans nos applications Angular. On parle des opérations CRUD : Create, Read, Update, Delete. Et puis nous avons utilisés aussi quelques opérateurs de la librairie RxJS : *tap*, *catchError*, *of*, etc.

Nous avons également pu voir comment simuler une API web en mémoire. C'est très pratique pour les tests et les démonstrations. Vous pouvez commencer à développer vos requêtes http avant même d'avoir un serveur à votre disposition !

En résumé

- Le module Angular utilisé pour effectuer des requêtes http sur le réseau est *HttpClient*.
- Il est possible de mettre en place une API web de démonstration au sein de votre application. Cela vous permettra d'interagir avec un jeu de données configuré à l'avance.
- Le module *HttpClient* permet d'effectuer facilement les quatre opérations de base nécessaire pour interagir avec une API : ajout, récupération, modification et suppression.
- RxJS fournit plusieurs opérateurs basiques pour traiter nos requêtes.
- Un même formulaire peut servir à ajouter ou éditer un élément dans votre application.
- C'est le rôle de l'API de déterminer un identifiant unique pour nos nouveaux objets. Vous ne devez jamais le faire dans le code de l'interface, car vous ne pouvez pas garantir que cet identifiant soit unique.

Chapitre 16 : Effectuer des traitements asynchrones avec RxJS

Dans le chapitre précédent, nous avons pu voir comment effectuer toutes les opérations de base avec http, mais je vous propose d'aller plus loin afin de profiter au mieux de la puissance de la programmation réactive.

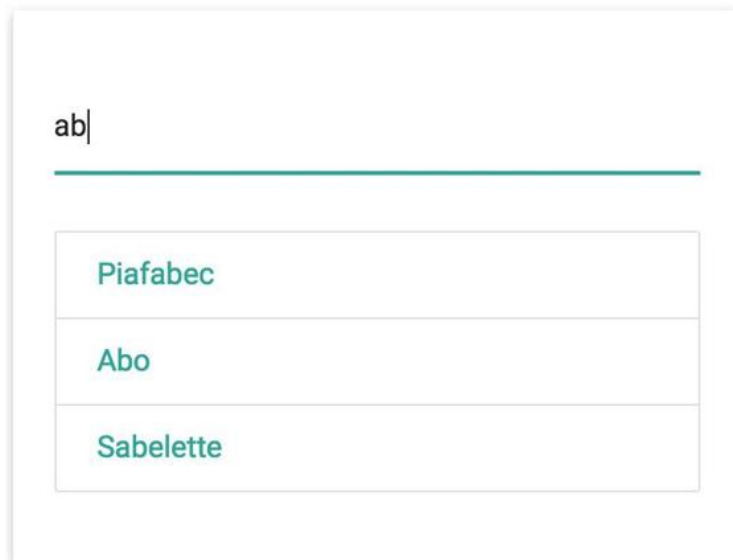
Ce que nous aimerions faire, c'est ajouter un champ de recherche sur la page d'accueil, pour permettre à l'utilisateur de retrouver un Pokémon plus facilement. Il suffira de taper le nom du Pokémon (ou juste le début de son nom), et notre application proposera au fur et à mesure les Pokémon de notre API, correspondant au terme de la recherche de l'utilisateur. Qu'en pensez-vous ?

En tous cas c'est un excellent entraînement pour progresser sur la programmation réactive et RxJS.

1. Construire un champ de recherche

Je vous propose d'ajouter une nouvelle fonctionnalité pour pouvoir rechercher des Pokémons via leur nom, sous la forme d'un champ de recherche.

Ce champ de recherche devra implémenter l'**auto-complétion**. Au fur et à mesure que l'utilisateur tapera un terme de recherche, nous afficherons immédiatement une liste de Pokémons correspondant à ce critère.



The image shows a search input field with the text 'ab|' inside. Below the input field, there is a dropdown list containing three items: 'Piafabec', 'Abo', and 'Sabelette'. The items are listed vertically and are highlighted with a light blue background.

Notre système d'auto-complétion pour rechercher un pokémon.

Pour le moment, nous avons effectué des requêtes http relativement simple. On peut parler de requêtes *one-shot*, on effectue la requête et on récupère un résultat.

Mais les requêtes ne sont pas toujours *one-shot*. Dans certains cas, on peut lancer une requête, puis l'annuler, pour finalement faire une requête différente, avant que le serveur ait répondu à la première ! Cette séquence *requête-annulation-nouvelle requête* est plus difficile à implémenter. C'est ce que nous allons voir maintenant !

1.1. Un champ de recherche

Pour commencer, créons une nouvelle méthode *searchPokemons* pour le service *pokemons.service.ts* :

```
searchPokemons(term: string): Observable<Pokemon[]> {  
  if (!term.trim()) {  
    // Si le terme de recherche n'existe pas,  
    // on renvoie un tableau vide sous la forme d'un Observable avec 'of'.  
    return of([]);  
  }  
  
  return this.http.get<Pokemon[]>(`api/pokemons/?name=${term}`).pipe(  
    tap(_ => this.log(`found pokemons matching "${term}"`)),  
    catchError(this.handleError<Pokemon[]>('searchPokemons', []))  
  );  
}
```

La fonction *searchPokemons* retourne un Observable qui renvoie un tableau de Pokémons. Dans la méthode *get* j'utilise une url spéciale, qui permet de filtrer les Pokémons d'après leur nom (la propriété *name*), en fonction d'un terme de recherche entré par l'utilisateur.

1.2. Consommer un Observable

Comment utiliser l'Observable renvoyé par notre méthode *searchPokemons*, depuis un composant ? Pour illustrer le fonctionnement, nous allons créer un nouveau composant *PokemonSearchComponent*. Ce composant aura pour rôle d'afficher et de gérer le champ de recherche des Pokémons, et l'auto-complétion.

Le template de ce composant sera simple : un champ de recherche, et une liste de résultats correspondants aux termes de la recherche. Voici le code à ajouter dans le fichier *search-pokemon.component.html* :

```
<div class="row">
  <div class="col s12 m6 offset-m3">
    <div class="card">
      <div class="card-content">
        <div class="input-field">
          <input #searchBox
            (keyup)="search(searchBox.value)" <!-- ligne 7 -->
            placeholder="Rechercher un pokémon" />
        </div>

        <div class="collection">
          <!-- ligne 12 -->
          <a *ngFor="let pokemon of pokemons$ | async"
            (click)="gotoDetail(pokemon)" class="collection-item" >
            {{ pokemon.name }}
          </a>
        </div>
      </div>
    </div>
  </div>
</div>
```

Hormis les balises nécessaires pour les feuilles de style, les lignes de code qui nous intéressent se situent :

- **À la ligne 7** : Au fur et à mesure que l'utilisateur tape dans la barre de recherche, la liaison de l'événement *keyup* appelle la

méthode *search* du composant avec la dernière valeur présente dans la barre de recherche.

- **À partir de la ligne 12** : La directive *NgFor* affiche une liste de Pokémons, issue de la propriété *pokemons* du composant associé. Mais, comme nous le verrons bientôt, **la propriété *pokemons* est maintenant un Observable et non plus un simple tableau** ! La directive *NgFor* ne peut rien faire avec un Observable à moins que nous n'appliquions le pipe *async* dessus. Le pipe *async* appliqué à l'Observable permet de n'afficher le résultat que lorsque l'Observable a retourné une réponse, et s'occupe de synchroniser la propriété du composant avec la vue !

Ce pipe *async*, qui signifie « asynchrone », peut également s'appliquer sur les Promesses !

Et pourquoi est-ce qu'il y a un \$ à la fin de la propriété pokemons\$?

Il s'agit d'une convention pour indiquer que la propriété *pokemons* n'est pas une simple propriété, mais un Observable, c'est-à-dire un flux.

Et maintenant, passons à la classe du composant ! Créer le fichier *search-pokemon.component.ts* dans le dossier *app/pokemons* :

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

import { debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operators';
import { Observable, Subject } from 'rxjs';

import { PokemonsService } from '../pokemons.service';
```

```

import { Pokemon } from './pokemon';

@Component({
  selector: 'pokemon-search',
  templateUrl: './search-pokemon.component.html',
})
export class PokemonSearchComponent implements OnInit {

  private searchTerms = new Subject<string>();

  pokemons$: Observable<Pokemon[]>;

  constructor(

    private pokemonsService: PokemonsService,

    private router: Router) {}

```

```

// Ajoute un terme de recherche dans le flux 'searchTerms'
search(term: string): void {
  this.searchTerms.next(term);
}

ngOnInit(): void {
  this.pokemons$ = this.searchTerms.pipe(
    // attendre 300ms de pause entre chaque requête
    debounceTime(300),
    // ignorer la recherche en cours si c'est la même que la précédente
    distinctUntilChanged(),
    // pour chaque terme de recherche
    // on renvoie la liste des Pokémons correspondants à cette recherche.
    switchMap((term: string) => this.pokemonsService.searchPokemons(term))
  );
}

gotoDetail(pokemon: Pokemon): void {

```

```
let link = ['/pokemon', pokemon.id];  
this.router.navigate(link);  
}  
}
```

Nous allons voir les explications de code dans les prochaines étapes.

1.3. Les termes de la recherche

Commençons par mettre l'accent sur la propriété *searchTerms* :

```
private searchTerms = new Subject<string>();  
// ...  
  
// Ajoute un terme de recherche dans le flux de l'Observable 'searchTerms'  
search(term: string): void {  
  this.searchTerms.next(term);  
}
```

La classe *Subject* n'appartient pas à Angular, mais à la librairie RxJS. C'est une classe particulière qui va nous permettre de stocker les recherches successives de l'utilisateur dans un tableau de chaînes de caractères, **sous la forme d'Observable**, c'est-à-dire avec une notion de *décalage dans le temps*. Nous allons voir en quoi cela va nous être utile. Retenez que la classe *Subject* hérite d'*Observable*, et donc *searchTerms* est bien un Observable.

C'est la propriété *searchTerms* qui permet de stocker ces recherches successives. Chaque appelle à *search* ajoute une nouvelle chaîne de caractère à *searchTerms*.

On utilise la fonction *next* car *searchTerms* n'est pas un tableau, auquel cas on aurait pu simplement utiliser la fonction *push*.

1.4. Initialiser un Observable

Un *Subject* est un Observable : c'est ce qui va nous permettre de transformer le flux de termes de recherche en un flux de tableaux de pokémons, et d'affecter le résultat à la propriété *pokemons*. Voici la méthode qui s'occupe de ça :

```
ngOnInit(): void {  
  this.pokemons$ = this.searchTerms.pipe(  
    // Attendre 300ms de pause entre chaque requête  
    debounceTime(300),  
    // Ignorer la recherche en cours si c'est la même que la précédente  
    .distinctUntilChanged(),  
    // Pour chaque terme de recherche,  
    // on renvoie la liste des Pokémons correspondants à cette recherche.  
    .switchMap((term: string) => this.pokemonsService.searchPokemons(term))  
  );  
}
```

Pourquoi le code paraît aussi « compliqué » ?

En fait, la problématique est la suivante : si nous passons chaque nouvelle saisie de l'utilisateur au *PokemonService*, nous allons déclencher une tempête de requêtes HTTP. Et ce n'est pas forcément une bonne idée ! Nous ne voulons pas surcharger notre API inutilement !

Heureusement, nous pouvons réduire ce flux de requêtes grâce à un certain nombre d'opérateurs. Nous faisons moins d'appels au *PokemonsService*, tout en obtenant toujours des résultats aussi rapides. Voici comment :

- **debounceTime(300)** : Met en pause l'exécution de nouvelles requêtes tant qu'une nouvelle recherche a été lancée il y a moins de 300 ms. Ainsi, il n'y a jamais de requêtes lancées plus fréquemment que toutes les 300ms.

- **distinctUntilChanged** : S'assure que nous envoyons une requête uniquement si le terme de la recherche a été modifié. En effet, il est inutile d'effectuer une autre requête pour la même recherche !
- **switchMap** : Appelle la méthode *searchPokemons* pour chaque terme de recherche qui passe à travers le filtre *debounceTime* et *distinctUntilChanged*. Cet opérateur annule et rejette les termes précédents de la recherche, et retourne seulement le plus récent, c'est-à-dire celui que l'utilisateur a saisi en dernier. Pratique !

1.5. Afficher le champ de recherche

Nous aimerions afficher la barre de recherche au-dessus de la liste de tous les Pokémons. Pour cela, rien de plus simple, il suffit d'utiliser la balise du composant : `<pokemon-search></pokemon-search>` ! Modifiez donc le template du composant `list-pokemon.component.ts`, en ajoutant la balise `<pokemon-search>` à la ligne 5 :

```
<h1 class='center'>Pokémons</h1>
<div class='container'>
  <div class="row">

    <pokemon-search></pokemon-search>

    <div *ngFor='let pokemon of pokemons' class="col s6 m4">
      <div class="card horizontal" (click)="selectPokemon(pokemon)" pkmnBorderCard>
        <div class="card-image">
          <img [src]="pokemon.picture">
        </div>
        <div class="card-stacked">
          <div class="card-content">
            <p>{{ pokemon.name }}</p>
            <p><small>{{ pokemon.created | date:"dd/MM/yyyy" }}</small></p>
            <span *ngFor='let type of pokemon.types' class="{{ type | pokemonTypeColor }}">
              {{ type }}
            </span>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

Enfin, on importe `PokemonSearchComponent`, et on l'ajoute au tableau `declarations` du module des Pokémons :

```
// Dans pokemons.module.ts, importez ceci :
import { PokemonSearchComponent } from './search-pokemon.component';

// ...
declarations: [
  // ...
```



```
PokemonSearchComponent // <= ... et ajoutez cela !  
],
```

Lancez l'application à nouveau, vous serez alors redirigé vers la liste de tous les Pokémons. Entrez du texte dans le nouveau champ de recherche qui est apparu. Vous devriez voir le système d'auto-complétion à l'œuvre !

Pas mal, non ? Et on peut encore améliorer l'application !

2. Ajouter une icône de chargement

Avez-vous remarqué qu'il y a un petit délai de chargement entre les pages ? Depuis que notre application utilise une API plutôt que les données issues d'un fichier, elle est un peu plus lente !

Par défaut, notre API simulée met une demie seconde pour retourner une réponse, ce qui correspond à peu près au temps que mettra une « vraie » API pour vous répondre.

Rassurez-vous, c'est un comportement parfaitement « normal ». Notre application est plus lente car nous récupérons les données depuis un service tiers, et nous devons attendre une réponse avant de pouvoir afficher les résultats. Cependant, nous pourrions rendre ce temps d'attente plus agréable pour l'utilisateur en ajoutant une icône de chargement.

L'icône de chargement que nous allons utiliser est un élément fourni par la librairie Materialize.

Pour cela, je vous propose de créer un nouveau composant pour représenter l'icône de chargement. Créer donc un fichier *loader.component.ts* dans le dossier */app* :

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

@Component({
  selector: 'pkmn-loader',
  template: `
    <div class="preloader-wrapper big active">
      <div class="spinner-layer spinner-blue">
        <div class="circle-clipper left">
          <div class="circle"></div>
```

```

    </div><div class="gap-patch">
      <div class="circle"></div>
    </div><div class="circle-clipper right">
      <div class="circle"></div>
    </div>
  </div>
  </div>
  </div>
  ,
})
export class LoaderComponent {}

```

Ce composant n'a pas de logique propre, il se contente d'afficher une petite icône circulaire. Il peut ensuite être injecté à tel ou tel endroit dans l'application, selon vos besoins. Nous allons l'afficher sur la page de détail d'un Pokémon, en attendant d'avoir récupéré les données nécessaires depuis l'API.

Dans le template *detail-pokemon.component.html* :

```

<!-- Avant -->
<h4 *ngIf="!pokemon" class="center">Aucun pokémon à afficher !</h4>

<!-- Après -->
<h4 *ngIf="!pokemon" class="center">
  <pkmn-loader></pkmn-loader>
</h4>

```

Dans le fichier *pokemon-form.component.html*, remplacer à la fin :

```

<!-- Avant -->

<h3 *ngIf="!pokemon" class="center">Aucun pokémon à éditer...</h3>

<!-- Après -->

<h4 *ngIf="!pokemon" class="center">
  <pkmn-loader></pkmn-loader>
</h4>

```

Il ne nous reste plus qu'à déclarer le nouveau composant dans le module Pokémons :

```
// pokemons.module.ts, ajouter ceci...  
  
import { LoaderComponent } from '../loader.component';  
  
// ...  
  
declarations: [  
  
// ...  
  
  LoaderComponent // ... et cela !  
  
],
```

Voilà, c'est bon ! Vous pouvez redémarrer l'application et profiter des icônes de chargement !

3. Conclusion

Nous avons pu voir un aperçu de l'utilisation des *Observables*. Même si les *Observables* peuvent être difficile à appréhender, essayez de vous habituer à la programmation réactive, et au changement de paradigme que cela représente.

Le développement web va vraiment dans ce sens, voyez-le comme un investissement pour l'avenir.

En résumé

- Les *Observables* permettent de faciliter la gestion des événements asynchrones.
- Les *Observables* sont adaptés pour gérer des séquences d'événements.
- Les opérateurs RxJS ne sont pas tous disponibles dans Angular. Il faut étendre cette implémentation en important nous-même les opérateurs nécessaires.
- Les opérateurs RxJS permettent de traiter des flux.
- Le traitement des flux de RxJS permet de transformer un flux de chaîne de caractères en un flux d'objets métiers. Dans notre cas, on a pu transformer les termes de recherches de l'utilisateur en une liste de Pokémons correspond aux critères de la recherche. Plutôt puissant !

Chapitre 17 : Authentification

Pour le moment, notre application n'est pas très fiable en termes de sécurité. N'importe quel utilisateur peut consulter ou modifier des Pokémons dans l'application. Or ce n'est pas forcément ce que nous voulons !

Il serait plus sûr de demander à l'utilisateur de s'authentifier avant de pouvoir interagir avec des Pokémons. Si l'utilisateur entre les bons identifiants, alors il est redirigé vers la liste des Pokémons, sinon il reste bloqué sur le formulaire de connexion.

Pour mettre en place une authentification, nous allons avoir besoin des *Guards*. Un *guard* est un mécanisme de protection utilisé par Angular pour mettre en place l'authentification, mais pas seulement. En avant !

1. Un Guard

Les Guards peuvent être utilisés pour gérer toutes sortes de scénarios **liés à la navigation** : rediriger un utilisateur qui tente d'accéder à une route, l'obliger à s'authentifier pour accéder à certaines fonctionnalités, etc.

Pour autant, les Guards reposent sur un mécanisme relativement simple, ils retournent un booléen, qui permet de contrôler le comportement de la navigation. Par exemple :

- Si le Guard retourne *true*, alors le processus de navigation continue.
- Si le Guard retourne *false*, alors le processus de navigation cesse, et l'utilisateur reste sur la même page.

Un Guard peut retourner un booléen de manière synchrone ou asynchrone. Mais dans la plupart des cas, un Guard ne peut pas renvoyer un résultat de manière synchrone, car il doit attendre une réponse. En effet, il peut être nécessaire de poser une question à l'utilisateur, de sauvegarder des changements, ou de récupérer des données plus récentes sur le serveur. Toutes ces actions sont asynchrones !

Dans la plupart des cas, le type de retour d'un Guard est `Observable<boolean>` ou `Promise<boolean>`, et le Router attendra la réponse pour agir sur la navigation.

Même si un Guard est seulement conçu pour interagir avec la navigation, il en existe des types différents :

- Le Guard *CanActivate* peut influencer sur la navigation d'une route, et notamment la bloquer. C'est ce type de Guard que

l'on va utiliser pour construire un système d'authentification dans notre application.

- Le Guard *CanActivateChild* peut influencer sur la navigation d'une route fille.
- Le Guard *CanDeactivate* peut empêcher l'utilisateur de naviguer en dehors de la route courante. Cela peut s'avérer utile lorsque l'utilisateur a oublié de valider un formulaire, avant de continuer à naviguer.
- Le Guard *Resolve* peut effectuer une récupération de données avant de naviguer.
- Le Guard *CanLoad* peut gérer la navigation vers un sous-module, chargé de manière asynchrone.

Dans ce chapitre, nous n'aurons besoin que du Guard de type *CanActivate*, pour construire le système d'authentification. La documentation officielle explique le fonctionnement de chacun des autres types de Guards, mais les explications sont en anglais !

On peut avoir différents Guards, à tous les niveaux du système de navigation. Cependant, si à un moment un Guard retourne false, tous les autres Guards en attente seront annulés, et la navigation entière sera bloquée.

2. Mise en place d'un Guard

Les applications ont souvent besoin de restreindre l'accès à certaines fonctionnalités, en fonction de l'utilisateur. Par exemple, obliger l'utilisateur à s'authentifier pour accéder à son espace membre. Il faut alors bloquer ou limiter l'accès jusqu'à ce que l'utilisateur se soit connecté.

Afin d'illustrer la mise en place d'un Guard, nous allons nous contenter d'afficher un message dans le navigateur lorsque l'utilisateur essaye d'accéder à la page d'édition d'un Pokémon, car il s'agit d'une route *sensible*, on ne veut pas que tout le monde puisse modifier les Pokémon.

Comme nous l'avons vu précédemment, nous utiliserons un Guard de type CanActivate.

Nous allons développer ce Guard dans le fichier *auth-guard.service.ts*. Ce fichier sera placé dans le dossier *app* :

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  canActivate() {
    alert('Le guard a bien été appelé !');
    return true;
  }
}
```

Bien sûr, pour le moment nous sommes seulement intéressés de voir comment mettre en place un Guard, donc cette première version ne fait rien de très d'utile. Il affiche un message d'information à l'utilisateur et renvoie immédiatement true, ce qui permet à la navigation de s'effectuer normalement par la suite, pour ne pas être bloquée.

On injecte notre *guard* à la racine de notre application grâce à la propriété *providedIn*, comme pour les services.

Ensuite nous devons déclarer ce Guard auprès de notre système de navigation. Comme nous voulons surveiller la route d'édition d'un Pokémon, ouvrez le fichier *pokemons-routing.module.ts* et modifiez le comme suit :

```
// Les autres importations ...
// Le guard de l'authentification :
import { AuthGuard } from '../auth-guard.service';

// Nos routes :
const pokemonsRoutes: Routes = [
  { path: 'pokemons', component: ListPokemonComponent },
  { path: 'pokemon/add', component: AddPokemonComponent },
  { path: 'pokemon/edit/:id', component: EditPokemonComponent, canActivate:
    [AuthGuard] }, // <-- Ajouter le guard.
  { path: 'pokemon/:id', component: DetailPokemonComponent }
];
```

Lancez l'application avec *ng serve -o* si ce n'est pas encore fait, et essayez de vous rendre sur la page d'édition d'un pokémon. Une pop-up s'ouvre et vous indique que le Guard a bien été appelé !

Mais, pour protéger dix routes, je vais devoir déclarer dix fois le même guard ?

Non, rassurez-vous ! En fait on peut déclarer nos routes différemment dans *pokemons-routing.module.ts* :

```
const pokemonsRoutes: Routes = [
  {
    path: 'pokemon',
```

```

canActivate: [AuthGuard],
children: [
  { path: 'all', component: ListPokemonComponent },
  { path: 'add', component: AddPokemonComponent },
  { path: 'edit/:id', component: EditPokemonComponent },
  { path: ':id', component: DetailPokemonComponent }
]
}
];

```

Ainsi, toutes nos routes sont protégées d'un coup. A la ligne 3, l'élément *path* permet de préfixer toutes les routes de notre module. Je me suis permis de renommer la route */pokemons* en */pokemon/all* pour avoir un préfixe commun à toute les routes du module. C'est plus « élégant » au niveau du code.

Avant de lancer l'application, nous devons encore faire deux choses :

- Retirer la pop-up d'alert dans le guard et la remplacer par un message dans la console. Sinon, des pop-up s'ouvriront en permanence lorsque l'utilisateur naviguera dans l'application ! Cela pourrait ressembler à du harcèlement publicitaire.
- Modifier la configuration des routes du module principale pour faire pointer la route par défaut sur la nouvelle route */pokemon/all*. Et aussi dans les composants qui pointent vers l'ancienne url (*PageNotFoundComponent* et *DetailPokemonComponent*).

D'abord, commençons par modifier notre Guard :

```

// Avant
alert('Le guard a bien été appelé !');

// Après
console.info('Le guard a bien été appelé !');

```

La modification est simple. Il ne nous reste plus qu'à modifier la route par défaut dans *app-routing.module.ts* :

```
// ...  
const appRoutes: Routes = [  
  { path: "", redirectTo: 'pokemon/all', pathMatch: 'full' },  
  { path: '**', component: PageNotFoundComponent }  
];  
// ...
```

... ainsi que les redirections du composant *page-not-found-component.ts* ...

```
<!-- Utiliser la nouvelle url dans le template du composant -->  
<!-- ... -->  
  <a [routerLink]="['/pokemon/all']" class="waves-effect waves-teal btn-flat">  
    Retourner à l' accueil  
  </a>  
<!-- ... -->
```

... et *detail-pokemon.component.ts* :

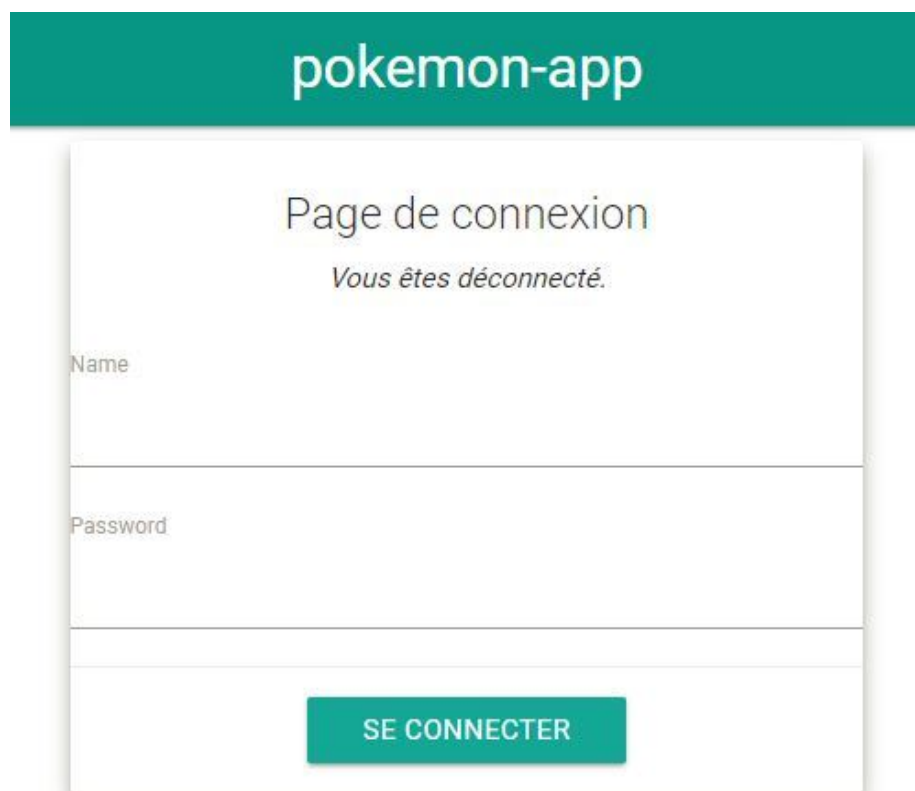
```
// Mettez à jour la méthode goBack du composant :  
goBack(): void {  
  this.router.navigate(['pokemon/all']);  
}
```

Parcourez à nouveau l'application, vous verrez le message du *Guard* s'afficher dans la console . Voilà, nos pokémons sont protégés par le Guard, ... mais bien mal protégés en fait !

3. Vers un système plus complet

Actuellement toutes les routes de notre module Pokémons sont accessibles, à tous le monde. Mais nous voulons rendre ce module accessible uniquement aux utilisateurs authentifiés, et avec un système fiable !

Nous allons donc améliorer notre Guard afin de rediriger les utilisateurs anonymes vers un formulaire de connexion lorsqu'ils essayeront d'accéder à l'application. Ce formulaire sera constitué d'un champ « *name* » et « *password* », comme la plupart des formulaires :



pokemon-app

Page de connexion

Vous êtes déconnecté.

Name

Password

SE CONNECTER

Notre formulaire de connexion pour accéder aux Pokémons.

Notre Guard va avoir besoin de l'aide d'un nouveau service, dédié à l'authentification. Cela nous permettra de bien découper le Guard et le service, dédié aux éléments métiers de la connexion.

Rappelez-vous : un fichier, une tâche !

Ce service sera chargé de connecter ou de déconnecter l'utilisateur courant, et de retenir l'url à laquelle il souhaitait accéder avant d'être bloqué par le Guard, afin d'être redirigé au bon endroit après son authentification !

Créez donc un fichier *auth.service.ts* dans le dossier app :

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

import { Observable, of } from 'rxjs';
import { tap, delay } from 'rxjs/operators';

@Injectable({
  providedIn: 'root',
})
export class AuthService {
  isLoggedIn: boolean = false; // L'utilisateur est-il connecté ?
  redirectUrl: string; // où rediriger l'utilisateur ?
  // Une méthode de connexion
  login(name: string, password: string): Observable<boolean> {
    // Faites votre appel à un service d'authentification si besoin
    let isLoggedIn = (name === 'pikachu' && password === 'pikachu');

    return of(true).pipe(
      delay(1000),
      tap(_ => this.isLoggedIn = isLoggedIn)
    );
  }

  // Déconnexion
  logout(): void {
    this.isLoggedIn = false;
  }
}
```

Ce service met à disposition plusieurs propriétés et méthodes :

- La propriété **isLoggedIn** (ligne 10) : C'est un booléen qui permet de savoir si l'utilisateur courant est connecté ou non. Par défaut, l'utilisateur est déconnecté lorsque l'application démarre.
- La propriété **redirectUrl** (ligne 11) : elle stocke l'url demandé par l'utilisateur quand il n'était pas encore connecté, pour pouvoir le rediriger vers cette page après son authentification.
- La méthode **login** (ligne 14) : cette méthode simule une connexion à une API externe en retournant un Observable qui se résout avec succès si l'utilisateur a entré les identifiants « *admin/admin* », après un délai d'une seconde. Bien sûr, vous pouvez personnaliser cette méthode pour correspondre à vos propres besoin de connexion par la suite.
- La méthode **logout** (ligne 25) : sans surprise, cette méthode permet de déconnecter immédiatement l'utilisateur courant. D'ailleurs, attention ! L'utilisateur se verra alors redirigé vers la page de connexion, sans qu'on lui demande son avis !

Il ne s'agit pas d'une « vraie » connexion ici, mais cela permet de faire fonctionner notre application de démonstration. En revanche, vous avez tout ce dont vous avez besoin pour construire un formulaire d'accès pour vos utilisateurs, avec votre propre API, grâce aux chapitres précédents.

Maintenant modifions notre guard *auth-guard* .service.ts, pour consommer ce nouveau service :

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

import { CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot } from
'@angular/router';

import { AuthService } from './auth.service';
```



```
@Injectable({  
  providedIn: 'root',  
})  
  
export class AuthGuard implements CanActivate {
```

```
  constructor(  
    private authService: AuthService, private router: Router) {}  
  
  // La méthode du Guard :  
  // détermine si l'utilisateur peut se connecter ou non !  
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {  
    let url: string = state.url;  
    return this.checkLogin(url);  
  }  
  
  // Méthode d'aide pour le Guard, qui interroge notre service.  
  checkLogin(url: string): boolean {  
    if (this.authService.isLoggedIn) { return true; }  
    this.authService.redirectUrl = url;  
    this.router.navigate(['/login']);  
  
    return false;  
  }  
}
```

Désormais, on fait appel à notre nouveau service pour déterminer si un utilisateur est authentifié ou non. Remarquez qu'on injecte en plus le *AuthService* et le *Router* dans le constructeur.

En paramètre de la méthode *canActivate*, l'objet *route*, de type *ActivatedRouteSnapshot*, contient la future route qui sera appelée. L'objet *state*, de type *RouterStateSnapshot*, contient le futur état du Routeur de l'application, qui devra passer la vérification du Guard.

Nous n'avons pas encore défini de fournisseur pour le *AuthService*, mais rien ne vous empêche d'injecter des services dans vos Guards.

Ce Guard retourne un booléen de manière **synchrone** comme résultat. Si l'utilisateur est connecté, alors le guard retourne *true* et la navigation continue. Sinon, nous stockons l'url de la route à laquelle l'utilisateur a tenté d'accéder dans la propriété *redirectUrl*, à la ligne 21. Ensuite on redirige l'utilisateur vers la page de connexion. Une page que nous allons créer tout de suite !

4. Une page de connexion

Il nous manque encore une chose : nous avons besoin d'un composant qui se charge d'afficher une page de connexion à l'utilisateur. Créez donc un fichier nommé *login.component.ts*, dans le dossier *app*. Pour être plus concis, j'ai mis le template et le composant dans le même fichier, que vous pourrez retrouver sur la page des ressources du cours.

Vous remarquerez que j'ai ajouté deux nouvelles propriétés et une méthode :

- Une propriété *name* de type *string*.
- Une propriété *password* de type *string* également.
- Une méthode *setMessage* pour prévenir l'utilisateur s'il a réussi à s'authentifier ou non.

Après l'authentification, l'utilisateur sera redirigé vers l'url stockée dans la propriété *redirect* (si cette propriété est non nulle, sinon on utilisera l'url par défaut « */pokemon/all* », qui redirigera l'utilisateur vers la liste des pokémons).

Ensuite, nous ajoutons le fichier de configuration de la connexion dans le dossier *app*, *login-routing.module.ts* :

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

@NgModule({
  imports: [
    RouterModule.forChild([
```

```
{ path: 'login', component: LoginComponent }  
])
```

```
],  
exports: [  
  RouterModule  
],  
providers: []  
})  
export class LoginRoutingModule {}
```

On utilise ce module pour enregistrer la route */login* à la ligne 8.

Pour terminer, dans le fichier *app.module.ts*, nous devons importer le composant *LoginComponent* et l'ajouter dans les déclarations du module racine *app.module.ts*. On importe aussi *LoginRoutingModule* dans les *imports* du *AppModule*, sans oublier d'ajouter le *FormsModule* car nous avons ajouté un nouveau formulaire dans notre application. Les commentaires ci-dessous indiquent les lignes que vous devez ajouter :

```
// ... autres importations ...  
  
import { FormsModule } from '@angular/forms';  
import { LoginComponent } from './login.component';  
import { LoginRoutingModule } from './login-routing.module';  
  
@NgModule({  
  imports: [  
    // ...  
    FormsModule, // <- ici, le FormsModule  
    PokemonsModule,
```

```

    LoginRoutingModule, // <- ici, au-dessus du AppRoutingModule !
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    LoginComponent, // <- ... et ici !
    PageNotFoundComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }

```

Les Guards et les fournisseurs de service dont ils ont besoin doivent être fournis au **niveau du module racine** ! Cela permet au *Routeur* de récupérer ces services à partir de l'injecteur pendant le processus de navigation.

Maintenant, essayer d'accéder à votre application. Automatiquement, vous serez redirigé vers la page de connexion, et vous devrez vous authentifier avant de pouvoir continuer ! On dirait que ça fonctionne !

5. Conclusion

Nous savons désormais comment mettre en place un vrai système d'authentification dans notre application ! Cela a été possible grâce aux *Guards*. Ils en existent d'autres, mais tous ont en commun de pouvoir modifier le comportement par défaut de la navigation.

Dans le prochain chapitre, nous allons voir un point important : comment **déployer** notre application ? C'est une problématique qui se pose dans tous les projets, et nous allons voir comment cela se passe pour Angular.

En résumé

- L'authentification nécessite la mise en place d'un système fiable : on utilise pour cela les Guards.
- Les Guards permettent de gérer toutes sortes de scénarios liés à la navigation : redirection, connexion, etc.
- Les Guards reposent sur un mécanisme simple. Ils retournent un booléen de manière synchrone ou asynchrone, qui permet d'influencer le processus de navigation.
- Il existe plusieurs types de Guards différents. Le type utilisé pour l'authentification est *CanActivate*.
- Il faut toujours déclarer les Guards au niveau du module racine, ainsi que les services tiers qu'ils utilisent.

Chapitre 18 : Déployer votre application

C'est la dernière étape, nous allons voir comment préparer notre application pour la production ! Pour le moment votre application est seulement sur votre machine, et personne d'autre que vous ne peut la voir. Il serait dommage de ne pas en faire profiter les autres !

Le terme « *Production* » désigne un environnement spécifique en informatique. Il y a l'environnement de *développement* lorsque vous développez, l'environnement de *test* lorsque vous testez, ... et l'environnement de *production*, lorsque votre application est prête pour être montrée aux utilisateurs. C'est d'ailleurs le seul environnement auquel vos utilisateurs ont accès normalement !

Dans ce chapitre, nous verrons comment déployer notre application sur Firebase. C'est une entreprise, appartenant à Google, qui propose d'héberger gratuitement des sites statiques en dessous d'un certain quota d'utilisation. C'est exactement ce dont nous avons besoin !

1. Le processus de déploiement

L'application fonctionne déjà sur notre machine locale, nous pouvons donc déployer l'application simplement en copiant tous les fichiers du projet vers le serveur distant. Ce serait sympas, non ?

Malheureusement, vous vous doutez que ce ne sera pas aussi simple. En informatique, le fait de passer du développement à une application utilisable par les utilisateurs s'appelle le **déploiement en production**.

Et pour déployer en production, nous avons plusieurs choses à faire. Il faut compresser nos fichiers pour qu'ils se chargent plus vite dans le navigateur des utilisateurs, s'assurer que tout fonctionne sur la nouvelle machine... Bref, nous devons effectuer un certain nombre d'opérations sur notre projet avant de le déployer.

Avant de commencer, voici une courte liste des tâches que nous devons accomplir pour déployer notre projet :

- Préparer notre projet en local avant le déploiement. Nous verrons que Angular a déjà tout prévu, et que nous n'aurons pas grand chose à faire de notre côté. C'est toujours ça de pris.
- Créer le projet sur le site de *Firebase Hosting*. C'est une entreprise appartenant à Google, qui propose d'héberger gratuitement des sites statiques en dessous d'un certain quota d'utilisation. C'est exactement ce dont nous avons besoin ! Cela vous donnera accès à une console d'administration d'où nous pourrions consulter l'historique de vos déploiements, revenir à une version plus ancienne de votre projet, etc.
- Enfin, déployer notre application sur le site de Firebase, afin qu'il soit accessible au monde entier. Rien que ça !

Le monde entier va bientôt pouvoir découvrir votre travail !

Allez hop, c'est parti !

Le seul pré-requis est de disposez d'un compte Google. Si vous n'en n'avez pas, vous pouvez créer un compte gratuitement.

2. Activer le mode production

Comme on l'a vu précédemment, on ne peut pas se contenter de déployer notre application en copiant tous les fichiers locaux vers le serveur.

Il faut d'abord optimiser tout un tas de chose en local, comme supprimer les dépendances inutiles en production, compresser les fichiers pour qu'ils ne soient pas trop long à charger dans le navigateur de vos utilisateurs, etc. Rien que dossier *nodes_modules* de notre projet contient beaucoup plus de code que nécessaire pour exécuter notre application. À titre d'exemple, le dossier *node_modules* d'une simple application « *Hello, World !* » peut contenir plus de 20 000 fichiers et peser plus de 180 MB ! Cela fait beaucoup de fichiers inutiles...

Je vous rassure tout de suite, Angular CLI est très bien fait et il va faire beaucoup de travail à notre place... à une condition. Il faut que notre application soit bien réglée en mode production. En effet, les applications Angular s'exécutent en mode développement par défaut. D'ailleurs, ce message doit s'afficher dans la console de votre navigateur depuis le début du cours :

“Angular is running in the development mode. Call enableProdMode() to enable the production mode.”

Le passage en mode production permet de rendre l'application plus rapide en désactivant des vérifications spécifiques au développement, et en optimisant tout un tas de fichiers. Le mode production sera automatiquement activé pour nous lorsque nous créerons le « livrable » de notre application, dans la prochaine étape. Donc nous n'avons rien de spécifique à faire.

Mais derrière la magie apparente, je voulais vous montrer que tout cela est bien concret. Rendez-vous dans le fichier *main.ts* :

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Comme vous pouvez le voir à la ligne 6, en fonction de l'environnement dans lequel se trouve notre application, Angular va passer en mode production (*enableProdMode*), ou non.

Mais, comment Angular sait dans quel environnement on se trouve ?

En fait, lorsqu'on démarre la commande *ng serve -o* pour développer en local, Angular se doute que nous sommes dans un environnement de développement. Et lorsque nous déploierons notre application, nous allons lancer une commande différente. Angular déduira que cette fois que nous allons en production.

Du coup, l'application tournera en mode production quand elle s'exécutera sur le serveur distant, et continuera à fonctionner en mode développement sur votre machine locale.

Pas bête !

3. Créer un livrable pour notre application

Avant de passer au déploiement, nous allons optimiser notre application locale pour la production.

Il faut vérifier que nous n'avons pas d'erreurs de syntaxe dans notre code, il faut minimiser et compresser nos fichiers, supprimer les dépendances inutiles en production comme TypeScript, car le navigateur ne peut interpréter que le code JavaScript, faire tourner notre application comme si nous étions en production pour s'assurer que tout fonctionne correctement, etc.

Heu, mais attend on n'en a pour combien de jours encore ?

Et bien j'ai une excellente nouvelle pour vous, tout cela peut se faire en une seule ligne de commande grâce à Angular CLI !

Si votre application est actuellement démarré dans un navigateur avec la commande `ng serve -o`, coupez là (Ctrl + C).

Ensuite, exécutez la commande suivante à la racine de votre projet :

```
ng build
```

Puis laissez tourner la commande. N'interrompez pas le script jusqu'à obtenir quelque chose de similaire :

```

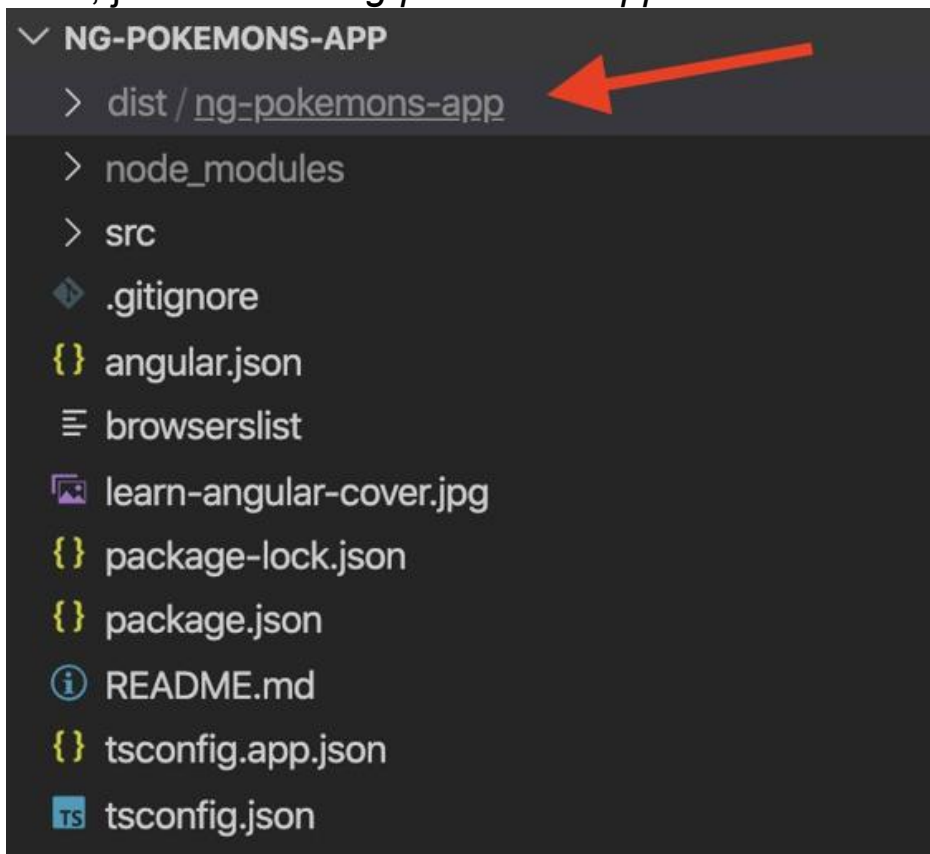
imac-de-simon-1:ng-pokemons-app simon$ ng build
Generating ES5 bundles for differential loading...
ES5 bundle generation complete.

chunk {polyfills} polyfills-es2015.js, polyfills-es2015.js.map (polyfills) 141 kB [initial] [rendered]
chunk {runtime} runtime-es2015.js, runtime-es2015.js.map (runtime) 6.16 kB [entry] [rendered]
chunk {runtime} runtime-es5.js, runtime-es5.js.map (runtime) 6.16 kB [entry] [rendered]
chunk {styles} styles-es2015.js, styles-es2015.js.map (styles) 9.78 kB [initial] [rendered]
chunk {styles} styles-es5.js, styles-es5.js.map (styles) 10.9 kB [initial] [rendered]
chunk {main} main-es2015.js, main-es2015.js.map (main) 124 kB [initial] [rendered]
chunk {main} main-es5.js, main-es5.js.map (main) 140 kB [initial] [rendered]
chunk {polyfills-es5} polyfills-es5.js, polyfills-es5.js.map (polyfills-es5) 656 kB [initial] [rendered]
chunk {vendor} vendor-es2015.js, vendor-es2015.js.map (vendor) 3.16 MB [initial] [rendered]
chunk {vendor} vendor-es5.js, vendor-es5.js.map (vendor) 3.7 MB [initial] [rendered]
Date: 2020-04-06T09:23:22.031Z - Hash: 9653cc2ba28b1aafcb2b - Time: 25133ms

```

Angular CLI s'occupe pour nous de créer un livrable pour notre application.

Ensuite, retournez dans l'arborescence de votre projet. Un nouveau dossier `dist/<nom-de-votre-projet>` devrait avoir fait son apparition. De mon côté, j'obtiens `dist/ng-pokemons-app` :



Un nouveau dossier « dist/ng-pokemons-app » a fait son apparition.

Ce dossier a été créé par Angular CLI, et il contient le « livrable » de notre application. C'est le contenu de ce dossier que nous allons déployer en production, car il s'agit d'une version ultra optimisée de

notre application. Tout y est minifié et compressé au maximum, afin d'offrir la meilleure expérience possible à nos utilisateurs.

Notre application peut maintenant être déployée !

4. Déployer sur Firebase Hosting

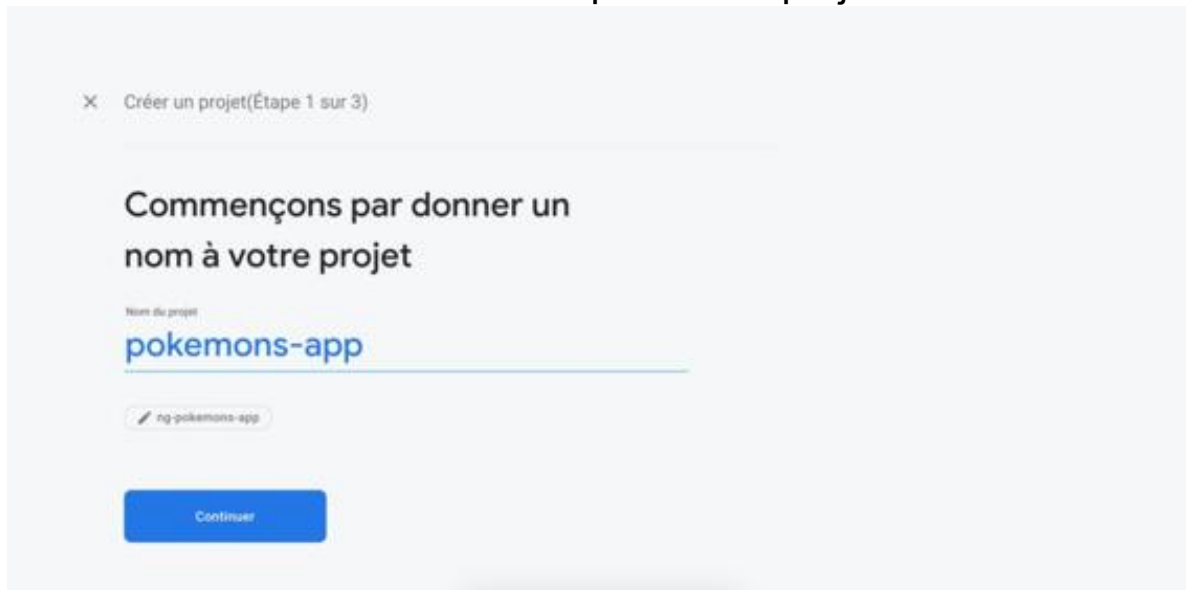
Le déploiement sur Firebase Hosting est assez simple, mais nécessite certaines tâches préliminaires lors du premier déploiement :

- **Créer un projet dans Firebase** : cela vous donnera accès à une console d'administration pour votre projet.
- **Installer Firebase-CLI en local** : il s'agit d'une sorte de boîte à outil, mise à disposition par Firebase, pour vous permettre de déployer votre application en une seule ligne de commande sur leurs serveurs !
- **Configurer votre projet auprès de Firebase** : pour préparer les éléments spécifiques au déploiement.
- **Déployer votre application de pokémons sur Firebase** : c'est la dernière étape. Ensuite, on pourra accéder à notre application de pokémons depuis n'importe où sur Internet !

5. Créer le projet dans Firebase

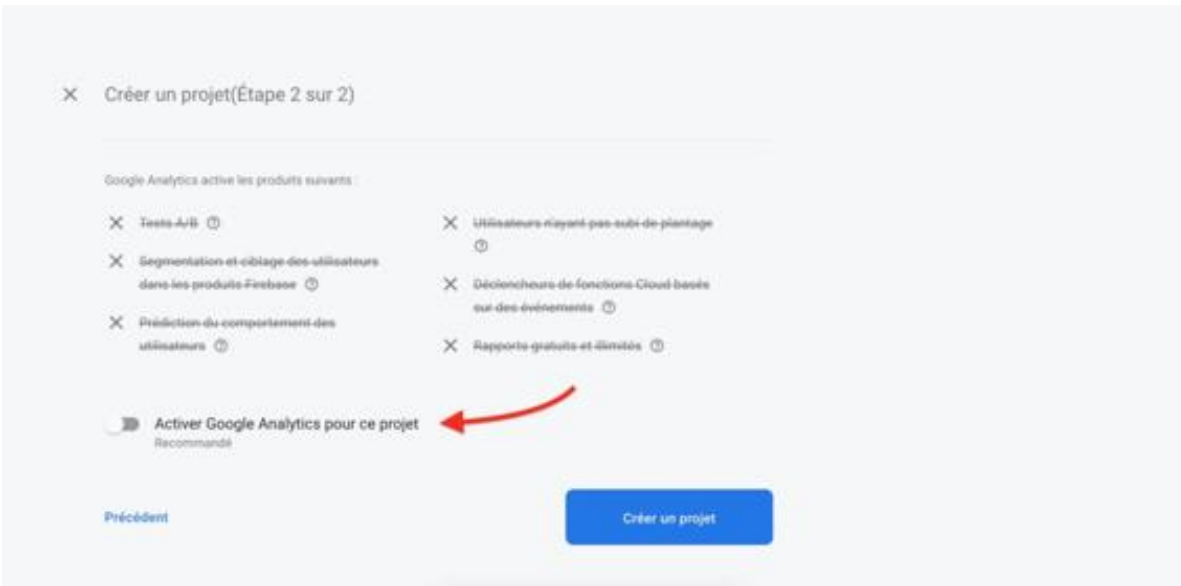
Pour commencer, connectez-vous à la console d'administration de Firebase sur *console.firebase.google.com*. Vous devez entrer vos identifiants Google si vous n'étiez pas déjà connecté.

Vous verrez un bouton, au milieu à gauche de la page, intitulé « *Ajouter un projet* ». Cliquez dessus. Une nouvelle page va s'ouvrir. Vous allez devoir choisir un nom pour votre projet :

The screenshot shows the 'Créer un projet' (Create project) screen in the Firebase console, specifically 'Étape 1 sur 3' (Step 1 of 3). The main heading is 'Commençons par donner un nom à votre projet' (Let's start by giving a name to your project). Below this, there is a text input field labeled 'Nom du projet' (Project name) containing the text 'pokemons-app'. Underneath the input field is a preview of the project ID, 'ng-pokemons-app', with a small edit icon. At the bottom of the form is a blue button labeled 'Continuer' (Continue).

Entrez un nom pour votre nouveau projet.

De mon côté, j'ai renseigné « *pokemons-app* » comme nom de projet. Ensuite, cliquez sur « Continuer », et vous devriez arriver sur l'écran suivant :



Décochez bien l'activation de Google Analytics !

Sur ce deuxième écran, Google vous demande gentilement d'ajouter le calcul de statistique sur votre site. Si vous laissez cette case cochée, vous devrez ensuite vous créer un nouveau compte Google Analytics. Comme ce n'est pas notre objectif, décochez bien cette case.

Finalement, cliquez sur "*Créer un projet*".

Vous arrivez ensuite dans un espace dédié à l'administration de votre application. Dans le menu, à gauche de l'interface, vous trouvez un onglet intitulé *Hosting*. Il s'agit du service que nous allons utiliser, et c'est à cet endroit que vous pourrez consulter l'historique de vos déploiements sur Firebase.

Maintenant, vous avez un projet disponible sur Firebase, prêt à accueillir les fichiers de votre application.

6. Installer la boîte à outil de Firebase

Nous devons maintenant installer l'utilitaire fournis par Firebase : *Firebase-CLI*. Cet outil permet de déployer facilement des applications sur les serveurs de Firebase.

CLI est l'acronyme de « Command-line Interface » ou « Interface en ligne de commande », en Français. Cela vous permet d'exécuter certaines commandes de l'utilitaire de Firebase depuis un terminal.

L'installation de Firebase-CLI est plutôt simple, tapez la commande suivante dans votre console :

```
npm install -g firebase-tools
```

Ensuite patientez un peu ... La commande peut prendre quelques minutes à s'exécuter, le temps de télécharger tous les éléments nécessaires.

Pour mettre à jour Firebase-CLI plus tard, il suffit de taper à nouveau la même commande.

Une fois l'installation terminée, on peut exécuter les commandes de FirebaseCLI. Avant de commencer à utiliser Firebase-CLI, exécutez ces deux commandes :

- *firebase --version* : Cela permet de vérifier que l'utilitaire est bien installé sur votre machine. Vous devriez voir un numéro de version qui s'affiche.
- *firebase login* : Cette instruction permet de lier votre compte Google (et donc vos projets Firebase) à l'utilitaire que vous venez d'installer. Suivez donc les instructions dans votre

terminal jusqu'à obtenir un message du type « *You are logged in as <l'adresse email de votre compte Google>* » .

Si ce message apparaît, cela signifie que vous êtes bien connecté ! Vous pouvez passer à l'étape suivante !

Si vous avez déjà utilisé la méthode firebase login il y a quelques temps, il est possible que vous receviez un message d'erreur « 401 – Unauthorized ». Cela est dû au fait que vos identifiants ont dû périmé avec le temps. Pour résoudre ce problème, rien de très compliqué, il suffit de lancer la commande firebase logout, puis firebase login à nouveau.

7. Configurer votre projet pour le déploiement

Avant de configurer votre projet pour le déploiement, relancez les commandes *npm install* et *npm start* pour être sûr que tout fonctionne correctement.

Maintenant, nous allons à nouveau utiliser Firebase-CLI pour configurer notre projet avant le déploiement. Voilà ce que l'on veut pouvoir dire à Firebase : *“Ok, ce dossier local sur ma machine doit correspondre au projet que j’ai créé sur Firebase tout à l’heure, et je veux déployer mes fichiers sur tes serveurs distants”*.

La commande magique qui permet de faire tout cela est :

```
firebase init
```

Vous devez vous placer à la racine de votre projet avant d'exécuter cette commande.

Un certain nombre d'éléments vous seront demandés lorsque vous exécuterez cette commande :

simplement « *dist/ng-pokemons-app* » , où adapter en fonction du nom de votre projet.

- **Configure as a single page app (rewrite all urls to /index.html) ?** Parfait, Firebase nous propose de s'occuper de configurer les redirections sur le serveur ! Taper donc « y » pour dire que vous êtes d'accord. Nous verrons où est-ce que Firebase a écrit ces règles de redirection.
- **File ./index.html already exists. Overwrite ?** Surtout pas ! Firebase nous demande gentiment s'il peut écraser notre fichier *index.html*. Bien sûr, ce n'est pas ce que nous voulons. Tapez « N » puis appuyer sur Entrée pour refuser.

Ça y est, nous avons fini de répondre aux questions. L'utilitaire a créé deux fichiers pour nous : *.firebaserc* et *firebase.json*. Seul le dernier fichier nous intéresse, car il permet de voir les fichiers qui ne seront pas déployés. Ouvrez donc le fichier *firebase.json* :

```
{
  "hosting": {
    "public": "src",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ]
  }
}
```

Ce fichier a été créé en fonction des réponses que nous avons données au-dessus. On peut vérifier que les règles de redirection vers *index.html* sont bien présentes de la ligne 9 à 12. C'est une configuration du serveur indispensable pour que votre application fonctionne.

Sur la documentation pour la configuration du fichier *firebase.json*, on apprend que l'option *ignore* indique à Firebase-CLI les fichiers ou dossiers que nous ne voulons pas déployer sur les serveurs de production, de la ligne 4 à 7.

Voilà, nous pouvons déployer. Enfin !

8. Déployer l'application

C'est maintenant que Firebase-CLI va se révéler être vraiment génial. Nous allons déployer notre application en une seule commande :

```
firebase deploy
```

Une fois que la commande a fini de s'exécuter, votre site est disponible en ligne immédiatement ! L'adresse par défaut pour accéder à votre site est : *https://<nom-du-projet>.firebaseapp.com*.

Si vous le souhaitez, Firebase vous propose d'associer un autre nom de domaine à votre application, si celui donné par défaut ne vous convient pas.

Si vous êtes vraiment fainéant, vous pouvez même taper une autre commande pour demander à Firebase d'afficher le site à votre place dans un navigateur :

```
firebase open
```

Ensuite sélectionner l'option *Hosting : Deployed Site*. Firebase-CLI va s'occuper pour vous d'ouvrir un navigateur à la bonne adresse !

Sinon, regardez dans votre invite de commande, la propriété *Hosting Url* correspond à l'url de votre site en ligne.

9. Conclusion

Et bien, félicitations ! Votre site est en ligne et accessible au monde entier !

Vous êtes parvenus au bout de ce cours sur Angular avec succès, et vous avez déployé votre application sur un serveur distant. Et ça, ce n'est pas rien. Encore bravo !

Avant de vous laisser, je me dois de vous rappeler que nous sommes loin d'avoir optimisé tout ce que l'on pouvait. Je vous redonne le lien vers la documentation officielle qui offre plusieurs pistes pour un déploiement plus optimisé.

Et pour finir sur une note plus positive, sachez que vous pouvez regarder les autres services que Firebase propose pour améliorer votre application : ajouter une base de données, associer un nouveau nom de domaine pour votre site... Bonne continuation dans vos futurs apprentissages, et merci d'avoir suivi ce cours en entier. Pour terminer en beauté, un dernier questionnaire vous attend !

En résumé

- Le déploiement est une étape dans un projet qui consiste à faire fonctionner une application sur l'environnement de production.
- Avant de déployer un projet local sur une machine de développement, il est nécessaire de prévoir une étape d'optimisation.
- Le site unpkg.com permet de charger des paquets npm depuis le web.
- Firebase propose un utilitaire en ligne de commande nommé Firebase-CLI, afin de déployer en ligne facilement des applications web statiques. Cela fonctionne aussi bien pour les applications qui ne sont pas développées avec Angular.
- Je vous recommande de lire la documentation officielle pour mieux connaître toutes les optimisations possibles que vous pouvez faire en local, avant le déploiement.