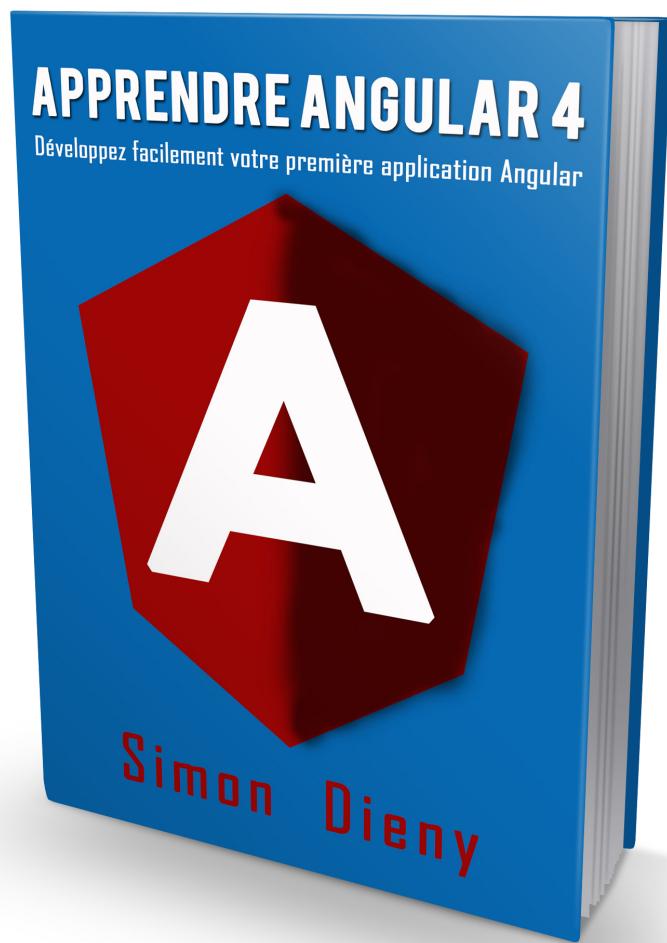

Apprendre Angular 4

« Développez facilement votre première application Angular »



Simon Diény

Reproduction totale ou partielle interdite sur quelque support que ce soit sans l'accord de l'auteur. Il est donc protégé par les lois internationales sur le droit d'auteur et la protection de la propriété intellectuelle. Il est strictement interdit de le reproduire, dans sa forme ou son contenu, totalement ou partiellement, sans un accord écrit de son auteur.

La loi du 11 mars 1957, n'autorisant, au terme des alinéas 2 et 3 de l'article 4, d'une part, que « les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, toute représentation ou reproduction , intégrale ou partielle, faite sans le consentement de l'auteur ou de ses ayants cause, est illicite » (alinéa premier de l'article 40). Cette représentation ou reproduction constituerait donc une contrefaçon sanctionnée par les articles 425 et suivant du Code pénal.

Droits d'auteur 2017 © Simon Diény

Tous droits réservés

Table des matières

Préambule.....	5
I. Vue d'ensemble de Angular	11
Présentation de Angular	12
ECMAScript 6	23
Découvrir TypeScript	38
Les Web Components.....	49
Premier pas avec Angular	50
II. Acquérir les bases de Angular	52
Les composants	53
Les templates	54
Les directives	55
Transformer les données avec les "Pipes".....	56
Les routes.....	57
Les modules	58
Les services et l'injection de dépendances	59
III. Aller plus loin avec Angular.....	60
Formulaires pilotés par le template	61
Formulaires pilotés par le modèle	62
La programmation réactive.....	63
Utiliser le module HTTP	64
Authentification.....	65
Tester votre application	66
Déployer votre application	67

IV. Annexes.....68

Préambule

Avant-propos

Bienvenue à tous (et à toutes !) dans ce cours consacré au développement d'applications avec Angular !

Ce cours s'adresse à tous les 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 pré-requis nécessaires. Mais pas d'inquiétude, il s'agit de connaissances élémentaires sur lesquelles vous pouvez vous formez, il y a des cours très bien faits là-dessus sur OpenClassrooms.

Je vous conseille donc de voir (ou de revoir) les cours suivants :

- 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 orienté 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 est disponible en ligne à cette adresse :

<https://ng2-pokemon-app.firebaseio.com>

Voici un aperçu :

pokemon-app

Pokémons

Rechercher un pokémon...

 <p>Bulbizarre 20/11/2016 Plante Poison</p>	 <p>Salamèche 20/11/2016 Feu</p>	 <p>Carapuce 20/11/2016 Eau</p>
 <p>Aspicot 20/11/2016 Insecte Poison</p>	 <p>Roucool 20/11/2016 Normal Vol</p>	 <p>Rattata 20/11/2016 Normal</p>
 <p>Piafabec 20/11/2016 Normal Vol</p>	 <p>Abo 20/11/2016 Poison</p>	 <p>Pikachu 20/11/2016 Electrik</p>

Voici l'application que vous réaliserez pendant ce cours !

IMPORTANT : 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 recommandé 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.

PS : La version actuelle du cours correspond à la version 4.0.0, qui est une notation sémantique du numéro de version. Mais on parle bien du "*nouvel Angular*", rassurez-vous ! 😊

Comment est structuré le cours ?

Ce cours est structuré en quatre parties distinctes :

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

II. 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...

III. Aller plus loin avec Angular : Nous lèverons le voile sur les formulaires, les requêtes HTTP, les tests, le déploiement...

IV. Annexes : Composés de quatre chapitres sur les bonnes pratiques de développement avec Angular, la configuration de TypeScript, les dépendances de base d'une application Angular et la gestion des titres pour vos pages.

Avant de commencer le cours ... (Important !)

Le code de l'application

L'ensemble du code de l'application est disponible librement à l'adresse suivante :

<https://github.com/moumoune/ng2-pokemon-app>

N'hésitez à vous y référer en cas de doute !

Les extraits de code

Ce cours est très orienté pratique et est donc composé de nombreux extraits de code. Plutôt que de les recopier manuellement, vous pouvez les copier coller depuis l'adresse suivante : <http://angular4.tumblr.com/ebook>

Attention : Le code disponible correspond à l'état du fichier lorsque vous le rencontrez pour la première fois dans le chapitre, sans les modifications ultérieures qui peuvent lui 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 ! 😊

Quiz

Il y a en tout trois quiz dans ce cours. Vous pourrez bien sûr noter vos réponses sur un papier et calculer votre score ensuite, mais une version numérique de ces quiz est proposé à la même adresse que les extraits de code :

<http://angular4.tumblr.com/ebook>

Google is everywhere

Les quiz et le chapitre sur le déploiement de l'application réalisé pendant le cours nécessite de posséder un compte Google, car nous effectuons le déploiement grâce à Firebase, une entreprise appartenant à Google, et les quiz sont des *Google Forms*.

Règles typographiques

Ce cours respecte la syntaxe suivante :

Ceci est une bulle contenant des informations supplémentaires sur le point qui vient d'être abordé.

Ceci est un message d'avertissement qu'il est recommandé de prendre en compte !

Cette bulle contient une réponse que vous pouvez vous poser, et auquel nous apporterons une réponse !

01 Ceci est un extrait de code, les numéros de lignes sont indiqués à droite !

Pratiquez, Pratiquez !

N'attendez pas de finir ce cours pour commencer à travailler, nous vous recommandons de faire vos tests au fur et à mesure sur votre poste de travail personnel.

De A à Z

Pour une première lecture, lisez le livre dans l'ordre, chapitre après chapitre.

Ensuite vous pourrez revenir sur certains chapitres sur lesquels vous souhaitez approfondir vos connaissances.

A propos de l'auteur



Simon Dieny

Je suis un jeune ingénieur logiciel, passionné par le développement web et mobile depuis mes études. J'ai eu un vrai coup de foudre pour Angular, depuis qu'il est en bêta !

Mon objectif est simple : Permettre à un maximum d'étudiants et de professionnels de se former facilement sur cette nouvelle technologie.

Vous pourrez retrouver sur mon blog des ressources complémentaires pour ce cours, ainsi que des articles techniques sur Angular, son environnement et son actualité :

<http://angular4.tumblr.com/>

Bon, c'est pas tout, mais on commence quand ? Et bien, maintenant ! C'est parti ! 😊

I. Vue d'ensemble de Angular

Chapitre 1

Présentation de Angular

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 ? Et bien, ça tombe bien, parce que 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 éclos jour après jour. Nous allons voir pourquoi vous devez faire le pari de vous lancer avec Angular 2, 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.

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 sympas d'avoir un cadre de travail commun, mais pour travailler sur quoi exactement ?* »

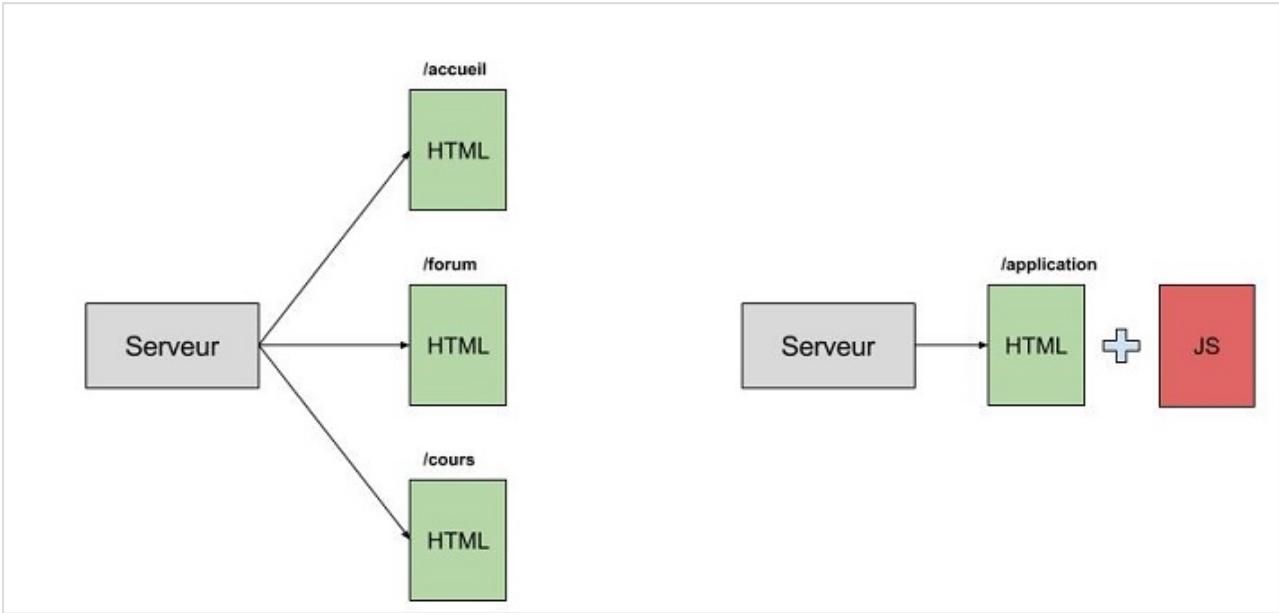
Effectivement, quels genres d'applications peuvent être développées avec Angular ? Et bien Angular permet de développer des **applications web**, de manière robuste et efficace. Nous allons voir la différence entre une application web, et un site web, car cette distinction est très importante pour bien comprendre dans quoi vous mettez les pieds.

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

A 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 une 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 ? Et 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'éco-système 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 Quizz 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 une message de bienvenue à l'utilisateur pour vérifier que tout marche bien ! On appelle cette petite application un "*Hello, World !* ».

Différences entre Angular et AngularJS

C'était quoi, AngularJS ?

AngularJS était très populaire et a été utilisé pour développer des applications clientes 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 le plus important :

- 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 frameworks (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. jQLite** : 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 mêmes raisons de performances, cette fonctionnalité n'est pas disponible de base. Cependant, 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. 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 importants. D'ailleurs, il a été annoncé que AngularJS ne seraient 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 *WebComponents* (nous y reviendrons) qui ont été développé après la sortie d'AngularJS, et qui fournissent des 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 WebComponents.

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 leur performance. 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 allégé pour Angular) ce qui résulte en une meilleure performance. Ces nouvelles performances font d'Angular 2 un outil parfaitement approprié pour développer des applications web mobiles.

La philosophie d'Angular

Angular est un framework **orienté composant**. Lors du développement de nos applications, nous allons coder une multitudes 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 attend, ... 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 chatter, contenu principale 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.

Le fonctionnement global d'une application

Sans rentrer dans les détails, je vais vous présenter quelques briques élémentaires qui forment les bases de toute application Angular qui se respecte. Cela fait beaucoup de nouvelles choses d'un coup, du vocabulaire et des concepts en pagaille, mais je ne nous demande pas de tous retenir, lisez simplement cette partie de haut en bas, et revenez-y quand vous ne vous rappelez plus de certains éléments et que vous avez besoin d'un rafraîchissement de mémoire ! 😊

Comme nous l'avons vu précédemment, toute votre application tiendra dans une simple page HTML : cela change radicalement la manière de concevoir vos sites web. Heureusement, Angular nous aide à organiser et à développer efficacement tout le code dont nous aurons besoin. Il fournit également plusieurs librairies, dont certaines d'entre elles forment le cœur du framework.

Nous allons voir quatre éléments à la base de toute application Angular :

- Le Module.
- Le Composant. (ou 'Component' en anglais)
- Le Template.
- Les métadonnées. (j'utiliserais le terme annotation dans la suite de ce cours)

Nous verrons chacun de ces éléments de manière plus poussée par la suite.

Le Module

Un module est une brique de votre application, qui une fois assemblée avec d'autres briques, forme une application à part entière. Chaque module est dédié à une fonctionnalité particulière.

Par exemple, le module `@angular/core` est un module Angular qui contient la plupart des éléments de base dont nous aurons besoin pour construire notre application. Si nous voulons créer un composant, il faudra importer la classe de base d'un composant depuis ce composant :

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

Souvent, on aura besoin d'importer des éléments depuis notre propre code, sans passer par une librairie. Dans ce cas on renseigne un chemin relatif à la place du nom de la librairie :

```
01 import { MaClasse } from './ce-dossier';
```

Heu ... c'est quoi cette syntaxe exactement ?

On importe et exporte des éléments grâce à la syntaxe des modules **ECMAScript 2015**. Si ça ne vous avance pas beaucoup, sachez que cette syntaxe a vu le jour pour répondre à un problème précis :

ECMAScript 2015, ES6 ou ES2015 désigne le même standard, que je vous présenterai dans un chapitre dédié.

Si vous avez déjà développé un site avec une grosse dose de JavaScript, vous vous êtes rendu compte que l'on se retrouve vite avec beaucoup de fichiers JavaScript qui dépendent tous les uns des autres, et on ne sait plus dans quel ordre les charger. Il a vite fallu organiser mieux ce code, et les développeurs se sont tourné vers des outils comme [require.js](#). Vous n'avez pas besoin de connaître ces outils pour continuer à suivre le cours, mais sachez que nous utiliserons [System.js](#) avec Angular, et que cet outil s'occupe pour nous de charger les modules et leurs dépendances.

Le Composant

Les composants sont la base des applications Angular. Vous allez écrire des composants tout le temps, pour gérer les interactions avec l'utilisateur, appeler des services, traiter les formulaires. Chaque composant contrôlera un petit bout de votre interface utilisateur, que nous appellerons une vue.

On définit la logique d'application d'un composant - ce qu'il doit faire pour supporter la vue - à l'intérieur d'une classe, et cette classe interagit avec la vue à travers un ensemble de propriétés et de méthodes.

Par exemple, un composant *FilmsComponent* pourrait avoir une propriété *films* , qui retourne un tableau de films. La vue associée à ce composant pourrait contenir un tableau HTML qui afficherait ses films.

Il est recommandé de suffixer le nom de vos composants par 'Component'. Vous verrez que ce sera utile pour vous y retrouver quand vous aurez des classes de services, de modèles, ...

Le Template

Un template ? C'est quoi ? Ne vous inquiétez pas, c'est simplement une "vue", associée à un composant. En fait, à chaque fois que l'on définit un composant, on lui associe tou-

jours un template. Un template est une forme de HTML spéciale qui dit à Angular ce que doit afficher le composant.

Parfois, un template contient simplement du HTML traditionnel. Par exemple, le code ci-dessous pourrait être un template Angular, sans aucun problème :

```
01 <h1>Blog sur Angular2</h1>
02 <p>Bienvenue sur mon blog, je m'appelle Jean Dupond !</p>
```

Mais parfois on a besoin d'injecter dans notre template des informations issues du composant lui-même :

```
01 <h1>Blog de Angular2</h1>
02 <p>Bienvenue sur mon blog, je m'appelle {{ prenomAuteur }} !</p>
```

Avez-vous compris ? - La syntaxe avec les accolades `{{ prenomAuteur }}` indique à Angular, que cette variable n'est pas disponible dans le template, et que la valeur de cette variable se trouve dans le composant qui gère ce template. En dehors de cette nouvelle syntaxe, que nous verrons dans un chapitre dédié, vous reconnaisserez les balises HTML classiques `<h1>` et `<p>`.

Les Annotations

Les métadonnées indiquent à Angular comment il doit traiter une classe. Par exemple, regardez la classe ci-dessous :

```
01 export class AppComponent { ... }
```

Comme son nom l'indique, il s'agit bien d'une classe de composant, puisqu'elle est suffixée par 'Component'.

Cependant, comment Angular sait qu'il s'agit vraiment d'un composant et pas d'un modèle, ou d'un service par exemple ? Et bien nous devons ajouter des annotations sur la classe `AppComponent`. Nous allons ajouter l'annotation `@component` pour indiquer à Angular que cette classe est un composant :

```
01 @Component({
02   selector: 'mon-app',
03   template: '<p>Ici le template de mon composant</p>'
04 })
05 export class AppComponent { ... }
```

Les annotations ont souvent besoin d'un paramètre de configuration. Le décorateur `@Component` n'échappe pas à cette règle, et prend en paramètre un object qui contient les informations dont Angular a besoin pour créer et lier le composant et son template. Il y a

plusieurs options de configuration possible, mais nous n'allons voir que les deux plus importants pour le moment, qui sont obligatoires lorsque vous définissez un composant :

- selector: un sélecteur est un identifiant unique dans votre application, qui indique à Angular de créer et d'insérer une instance de ce composant à chaque fois qu'il trouve une balise `<mon-app></mon-app>` dans du code HTML d'un composant parent. Le nom de cette balise vient de la ligne 2 du code ci-dessus :

```
01 selector: 'mon-app',
```

- template: Le code du template lui-même. Il est également possible d'écrire le code du template dans un fichier à part si vous le souhaitez. Dans ce cas-là, remplacer template par templateUrl, et indiquez chemin relatif vers le fichier du template.

Hey, mais ce n'est pas du HTML valide ça:

`<mon-app></mon-app> ?`

Et bien détrompez-vous, ce code est parfaitement valide, et vous pourrez même le faire valider par le [W3C](#). Nous verrons pourquoi dans la suite de ce cours.

Il existe bien sûr d'autres annotations : `@Injectable`, `@Input` ou encore `@Pipe` par exemple.

Conclusion

Il y a bien sûr beaucoup d'autres éléments qui peuvent constituer une application Angular, mais nous ne sommes qu'au début de notre apprentissage !

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

Pour les curieux, sachez que la documentation officielle d' Angular se trouve ici: <https://angular.io/> . Cette documentation n'est disponible qu'en anglais par contre. 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 éco-système 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 d'ECMAScript6 (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 "*prototypale*", 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.

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 sûrement 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 **transpilateur** pour convertir leur code ES6 en du code ES5. Ainsi leur code est lisible par tous les navigateurs.

« *Un transpilateur ?* »

Le transpilateur est un outil qui permet de publier son code pour les navigateurs qui ne supportent pas encore l'ES6 : le rôle du transpilateur 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 transpilateur : 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 transpilateur ! Patience 😊.

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

Voyons tout de suite ce que ES6 apporte et ce qu'il est possible de faire avec, en avant !

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

Le nouveau monde d'ES6

Commençons tout de suite avec une des meilleures nouveautés d'ES6 : il est désormais possible d'utiliser des classes en Javascript !

Les classes

Pour créer une simple classe *Vehicule* avec ES5, nous aurions dû faire comme ceci :

```
01 function Vehicule(couleur, nombreRoueMotrice) {
02   this.couleur = couleur;
03   this.nombreRoueMotrice = nombreRoueMotrice;
04   this.moteurAllumer = false;
05 }
06
07 Vehicule.prototype.demarrer = function demarrer(){
08   this.moteurAllumer = true;
09 }
10
11 Vehicule.prototype.couperMoteur = function couperMoteur(){
12   this.moteurAllumer = false;
13 }
```

Le code ci-dessus était le moyen détourné utilisé pour créer un objet, en utilisant la mécanique des prototypes, propre à JavaScript.

Si vous ne voyez pas ce qu'est l'*héritage prototypal*, vous pouvez regarder cette vidéo très bien faite [sur Youtube](#).

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. 😊

```
01 class Vehicule {
02
03   constructor(couleur, nombreRoueMotrice) {
04     this.couleur = couleur;
05     this.nombreRoueMotrice = nombreRoueMotrice;
06     this.moteurAllumer = false;
07   }
08
09   demarrer() {
10     this.moteurAllumer = true;
11   }
```

```
12
13  couperMoteur() {
14    this.moteurAllumer = false;
15  }
16 }
```

Voilà une classe JavaScript, bien différente de ce que l'on avait précédemment. Si vous avez bien remarqué à la ligne 3, on a même droit à un constructeur !

C'est-il pas merveilleux ! 😊

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 :

```
01 function Vehicule(couleur, nombreRoueMotrice) {
02   this.couleur = couleur;
03   this.nombreRoueMotrice = nombreRoueMotrice;
04   this.moteurAllumer = false;
05 }
06
07 Vehicule.prototype.demarrer = function demarrer() {
08   this.moteurAllumer = true;
09 }
10
11 Vehicule.prototype.couperMoteur = function couperMoteur() {
12   this.moteurAllumer = false;
13 }
14
15 // Une voiture est un véhicule.
16 function Voiture(couleur, nombreRoueMotrice, nombrePlaceAssise) {
17   Vehicule.call(this, couleur, nombreRoueMotrice);
18   this.nombrePlaceAssise = nombrePlaceAssise;
19 }
20
21 Voiture.prototype = Object.create(Vehicule.prototype);
22
23 // Une moto est un véhicule également.
24 function Moto(couleur, nombreRoueMotrice, debrider) {
25   Vehicule.call(this, couleur, nombreRoueMotrice);
```

```

26 this.debrider = debrider;
27 }
28
29 Moto.prototype = Object.create(Vehicule.prototype);

```

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

```

01 class Vehicule {
02   constructor(couleur, nombreRoueMotrice, moteurAllumer = false) {
03     this.couleur = couleur;
04     this.nombreRoueMotrice = nombreRoueMotrice;
05     this.moteurAllumer = moteurAllumer;
06   }
07
08   demarrer() {
09     this.moteurAllumer = true;
10   }
11
12   couperMoteur() {
13     this.moteurAllumer = false;
14   }
15 }
16
17 class Voiture extends Vehicule {
18   constructor(couleur, nombreRoueMotrice, moteurAllumer = false,
19               nombrePlaceAssise) {
20     super(couleur, nombreRoueMotrice, moteurAllumer);
21     this.nombrePlaceAssise = nombrePlaceAssise;
22   }
23 }
24
25 class Moto extends Vehicule {
26   constructor(
27     couleur, nombreRoueMotrice, moteurAllumer = false, debrider) {
28     super(couleur, nombreRoueMotrice, moteurAllumer);
29     this.debrider = debrider;
30   }
31 }

```

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

Les valeurs 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 , qui prend un nombre quelconque d'arguments en paramètre, les additionne, puis retourne le résultat :

```
01 function somme() {  
02   var resultat = 0;  
03   for (var i = 0; i < arguments.length; i++) {  
04     resultat += arguments[i];  
05   }  
06   return resultat;  
07 }
```

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. 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 :

```
01 function test (valeurParDefault) {  
02   valeurParDefault = valeurParDefault || undefined;  
03   return valeurParDefault;  
04 }
```

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 il vaut 1 par défaut :

```
01 function multiplier(a, b) {  
02   var b = typeof b !== 'undefined' ? b : 1; // b est facultatif  
03   return a*b;  
04 }  
05  
06 multiplier(2, 5); // 10  
07 multiplier(1, 5); // 5  
08 multiplier(5);    // 5
```

A la *ligne 2*, nous utilisons un opérateur ternaire pour attribuer la valeur `1` à la variable `b` si aucun nombre n'a été passé en paramètre n°2. Ce code semble très compliqué pour réaliser quelque chose de très simple.

Heureusement, ES6 nous permet d'utiliser une syntaxe plus élégante:

```
01 function multiplier(a, b = 1) {  
02   return a*b;  
03 }  
04 multiplier(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 ? 😊

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 quand on veut effectuer beaucoup d'opérations sur une variable et que vous ne voulez pas polluer d'autres contextes avec plus de variables que nécessaire :

```
01 var x = 1;  
02  
03 if(x < 10) {  
04   let v = 1;  
05   v = v + 21;  
06   v = v * 100;  
07   v = v / 8;  
08  
09   console.log(v);  
10 }  
11  
12 // v n'est pas définie, car v a été déclaré avec 'let' et non 'var'  
13 console.log(v);
```

Et cela fonctionne pour les boucles également !

```
01 for (let i = 0; i < 10; i++) {  
02   console.log(i); // 0, 1, 2, 3, 4 ... 9  
03 }
```

```
04 // i n'est pas défini hors du contexte de la boucle  
05 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 :

```
01 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 :

```
01 PI = 3.146; // 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 un tableau ou un 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 :

```
01 const MATHEMATIQUES_CONSTANTES = {PI: 3.141592, e: 2.718281};  
02 MATHEMATIQUES_CONSTANTES.PI = 3.146; // 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* !

Retrouvez la liste exhaustive des mots-clés réservés JavaScript (ES6 compris) [ici](#).

Un raccourci pour créer des objets

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

```
01 function creerVoiture() {  
02   let couleur = 'vert';  
03   let roue = 4;  
04   return { couleur, roue };  
05 }
```

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

```
01 function creerVoiture() {  
02   let couleur = 'vert';  
03   let roue = 4;  
04   return { couleur: couleur, roue: roue };  
05 }
```

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

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 générale, 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 :

```
01 recupererUtilisateur(idUtilisateur, function(utilisateur) {  
02   recupererListeAmis(utilisateur, function(amis) {  
03     afficherListe(amis);  
04   });  
05 });
```

L'exemple ci-dessus permet de récupérer un objet *utilisateur* 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:

```
01 recupererUtilisateur(idUtilisateur)  
02 .then(function(utilisateur) {  
03   recupererAmis(utilisateur);  
04 })  
05 .then(function(amis) {  
06   afficherListe(amis);  
07 });
```

Il n'y a même pas besoin d'explications : le code est assez clair pour parler 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 *callback de succès* et une *callback d'erreur*. Ainsi, lorsque la promesse est réalisée, c'est la callback de succès qui est appelé, 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 :

```
01 let recupererUtilisateur = function(idUtilisateur) {  
02   return new Promise(function(resolve, reject) {  
03     // appel asynchrone au serveur pour récupérer  
04     // les informations d'un utilisateur...  
05     // à partir de la réponse du serveur,  
06     // on extrait les données de l'utilisateur :  
07     let utilisateur = response.data.utilisateur;  
08  
09     if(response.status === 200) {  
10       resolve(utilisateur);  
11     } else {  
12       reject('Cet utilisateur n\'existe pas.');//  
13     }  
14   })  
15 }
```

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

```
01 recupererUtilisateur(idUtilisateur)  
02 .then(function (utilisateur) {  
03   console.log(utilisateur); // en cas de succès  
04 }, function (error) {  
05   console.log(error); // en cas d'erreur  
06 });
```

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

Les fonctions flèches, ou « Arrow Functions »

Les arrows functions (ou fonctions 'flèches' en français, mais avouez que ça fait moins pro ! 😊) sont une nouveauté d'ES6. Le premier cas d'utilisation des arrows functions, c'est avec *this*. L'utilisation de *this* peut être compliqué, surtout dans le contexte de fonctions à l'intérieur d'autres fonctions. Prenons l'exemple ci-dessous:

```
01 class Personne {  
02   constructor(prenom, email, bouton) {  
03     this.prenom = prenom;  
04     this.email = email;  
05  
06     bouton.onclick = function() {  
07       // le 'this' fait référence au bouton,  
08       // et non à une instance de Personne:  
09       envoyerEmail(this.email);  
10     }  
11   }  
12 }
```

Comme vous le voyez en commentaire, le *this* de la ligne 9 ne fait pas référence à l'instance de Personne mais au bouton sur lequel l'utilisateur a cliqué. Hors, 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, comme ceci:

```
01 class Personne {  
02   constructor(prenom, email, bouton) {  
03     this.prenom = prenom;  
04     this.email = email;  
05     // 'this' fait référence ici à l'instance de Personne  
06     var that = this;  
07  
08     bouton.onclick = function() {  
09       // 'that' fait référence à la même instance de Personne  
10       envoyerEmail(that.email);  
11     }  
12   }  
13 }
```

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:

```
01 bouton.onclick = () => { envoyerEmail(this.email); }
```

Les *arrow functions* ne sont donc pas tout à fait des fonctions classiques, parce qu'elle 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 tienne sur une seule ligne:

```
01 DeUnACinq = [1, 2, 3, 4, 5];
02 var DeDeuxADix = DeUnACinq.map((n) => n*2);
03 console.log(DeDeuxADix); // [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 passé en paramètre.

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

```
01 // un traitement asynchrone en une seule ligne !
02 recupererUtilisateur(idUtilisateur)
03 .then(utilisateur => recupereAmis(utilisateur))
```

Les Affectations déstructurées, ou le 'Destructuring'

Le nom barbare de '*déstructuration*' désigne en fait un concept assez simple. C'est un outil formidable pour extraire des données d'un tableau. Jetons un coup d'oeil à un exemple de déstructuration de tableau :

```
01 var [premier, deuxième, troisième] = ["Usain", "Yohan", "Justin"];
```

Voilà, vous venez de déstructurer un tableau ! Les valeurs du tableau affecté sont automatiquement attribuées à partir des valeurs du tableau de droite. Est-ce que ça marche pour les objets ? Bien sûr !

```
01 var Personne = {prenom: "Usain", nom: "Bolt" };
02 // La destructuration d'un objet !
03 var {prenom, nom} = Personne;
```

C'est quand même plus pratique que d'écrire :

```
01 var prenom = personne.prenom;
02 var nom = personne.nom;
```

La collection Map

Map est l'équivalent d'un dictionnaire dans lequel vous ajoutez des paires de *clé-valeur*. Imaginons par exemple le cas où vous souhaitez stocker le classement d'un joueur :

```
01 let zlatan = {rang: 1, name: 'Zlatan'};  
02 // Je crée une nouveau dictionnaire  
03 let joueurs = new Map();  
04 // J'ajoute l'objet 'zlatan' à la clé '1'  
05 joueurs.set(zlatan.rang, zlatan.name);
```

Vous disposez maintenant d'un dictionnaire contenant un joueur et son classement. Nous verrons ci-dessous les opérations que nous pouvons effectuer sur ce dictionnaire.

La collection Set

Il est également possible de créer des listes sur le même principe. Une liste se comporte comme un dictionnaire, mais sans clés.

```
01 let joueurs = new Set(); // Je crée une nouvelle liste  
02 joueurs.add(zlatan); // j'ajoute un joueur dans cette liste
```

Vous remarquez que j'utilise la méthode *add()* et non *set()* comme pour les dictionnaires.

Enfin, sachez que vous pouvez faire ensuite tous un tas d'opérations sur vos collections, qu'il s'agisse de dictionnaires ou de listes. Regardez le code ci-dessous, il est suffisamment explicite pour pouvoir se passer d'explications 😊.

```
01 joueurs.size; // affiche le nombre d'éléments dans la collection  
02 // Dictionnaire: affiche si oui ou non,  
03 le dictionnaire contient la clé correspondant au rang de Zlatan.  
04 joueurs.has(zlatan.rang);  
05 // Liste: affiche si oui ou non, la liste contient le joueur Zlatan.  
06 joueurs.has(zlatan);  
07 // Dictionnaire: supprime un élément d'après une clef.  
08 joueurs.delete(zlatan.rang);  
09 // Liste: supprime l'élément passé en paramètre.  
10 joueurs.delete(zlatan);
```

Les 'Template Strings'

Avant de terminer ce chapitre, je voulais vous comment utiliser les *template strings* : c'est une petite amélioration de ES6 bien sympas ! 😊

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

```
01 var duTexte = "duTexte";  
02 var unAutreTexte = "unAutreTexte";  
03
```

```
04 var chaineLonguePainible = duTexte;
05 chaineLonguePainible += " blabla";
06 chaineLonguePainible += unAutreTexte;
07 chaineLonguePainible += "blabla";
08
09 return chaineLonguePainible;
```

Comme vous pouvez le constatez, 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.

```
01 // On peut écrire des strings sur plusieurs ligne grâce au backtick...
02 var uneStringSurPlusieursLigne = `bla
03 blablabalbalbalballb
04 balblablablalabla
05 b
06 ablablabbabl`;
07
08 // .. mais pas avec des guillemets !
09 // Regarder la couleur syntaxique, vous comprendrez que ce code risque
10 // de lever une erreur !
11 var sans = "bla
12 blabla
13 blabla
14 blabla"
```

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

```
01 return `${this.name} a un mail: ${this.email}`;
```

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

Conclusion

Nous avons vu dans ce chapitre quelques changements majeurs qu'a apporté ES6 à JavaScript. Sachez que ES6 est rétro-compatible, donc vous pouvez toujours écrire 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 ? Et bien, direction le prochain chapitre, où nous allons voir le transpilateur 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 transpilateur pour convertir notre code de 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 2 nouveaux mots-clés: *let* et *const*. *Let* est la nouvelle manière 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 cocréé 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.

C'est quoi, "TypeScript" ?

Le language 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 commence à écrire beaucoup de code, on arrive 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 mis alors à développer des méta-langage 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 méta-langage de votre choix, vous devez la compiler, c'est à dire la 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-languages produisent du JavaScript au final, pourquoi je ne développerais pas directement en JavaScript ? (Pourquoi s'embêter avec cette étape intermédiaire ?)

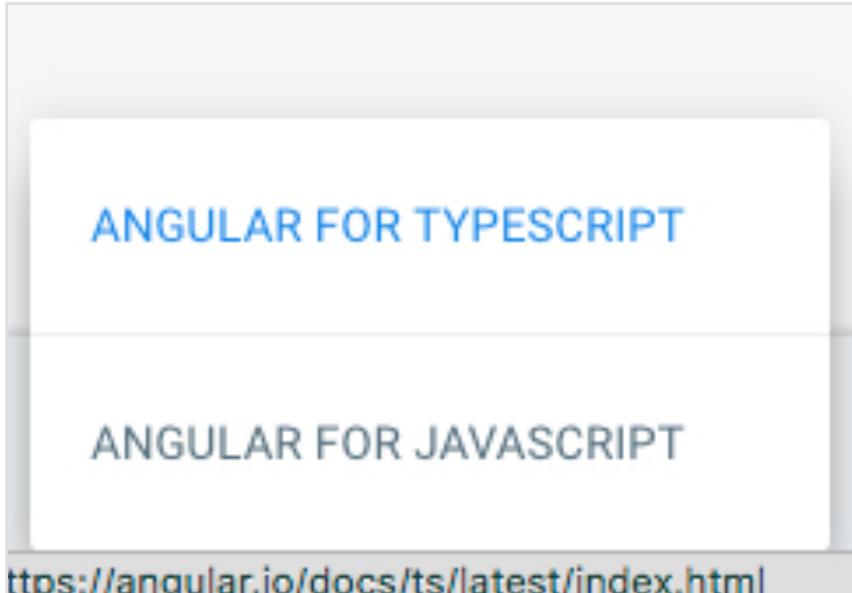
En effet, c'est une excellente question. En théorie, on pourrait effectivement se passer de méta-langage et réécrire nous-même les éléments dont nous avons besoin. En pratique, c'est très différent :

- D'abord vous n'allez pas réécrire tous ce que le méta-langage 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 tous de suite le lien entre TypeScript et ce qui nous intéresse : Angular.

Angular et TypeScript

Ce que vous devez savoir, c'est que Angular vous permet de développer vos applications en JavaScript (c'est normal vous me direz !), ou en TypeScript. Si vous allez sur la documentation officielle, vous verrez que plusieurs versions de la documentation vous sont proposées :



Il est possible d'utiliser JavaScript ou TypeScript pour développer des applications Angular

Alors je ne vais pas tourner autour du pot : il est chaudement recommandé d'utiliser TypeScript avec Angular, c'est comme ça que vous trouverez plus d'aide par la suite. Et il y a une bonne raison à cela : c'est que l'équipe de chez Google chargée de développer le framework recommande explicitement d'utiliser TypeScript.

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.

"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 :

```
01 // fonction qui retourne un message de bienvenue
02 function direBonjour(personne) {
03   return "Bonjour, " + personne;
04 }
05
06 // création d'une variable "Jean"
07 var Jean = "Jean";
08
09 // on ajoute dans notre la page HTML un message
10 // pour affiche "Bonjour, Jean".
11 document.body.innerHTML = direBonjour(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, voilà ce que je vous propose. On va installer ce dont on va avoir besoin pour transformer notre TypeScript en JavaScript, puis on va créer un page HTML classique qui nous permettra de tester notre code Javascript généré. 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 vérifier leur versions respectives grâce aux commandes suivantes :

```
01 node -v  
02 v6.4.0
```

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

```
01 npm -v  
02 3.9.3
```

Si vous n'avez pas les mêmes versions que moi, ce n'est pas grave. Essayer d'avoir au moins la version **5.x.x** pour **node** et la version **3.x.x** pour **npm**.

Si les deux commandes ci-dessus vous affichent une erreur, c'est probablement que vous n'avez pas installé [Node](#). Vous allez en avoir besoin pour la suite, revenez à ce chapitre une fois que vous l'aurez installé.

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

```
01 npm install -g typescript
```

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

```
01 tsc -v  
02 Version 1.8.9
```

Voilà, c'est aussi simple que ça !

Là aussi, la version a de l'importance. Essayez d'avoir au moins la version 1.8, au pire, la version 1.5, mais pas en-dessous, ou vous serez bloqués dans la suite du cours. Je vous avait prévenu que c'était un éco-système bourgeonnant !

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

```
01 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 :

```
01 // fonction qui retourne un message de bienvenu
02 function direBonjour(personne) {
03   return "Bonjour, " + personne;
04 }
05 // création d'une variable "Jean"
06 var Jean = "Jean";
07 // on ajoute dans notre la page HTML un message
08 // pour affiche "Bonjour, Jean".
09 document.body.innerHTML = direBonjour(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 comment ça marche ! 😊

A quoi serve les fichiers de définition dans TypeScript ?

*Ceux 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 !

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**.

Le typage avec TypeScript

La syntaxe pour déclarer une variable typée avec TypeScript est la suivante :

```
01 // Syntaxe pour déclarer une variable typé en TypeScript
02 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 :

```
01 // On déclare un nombre
02 var pointDeVie: number = 100;
03
04 // On déclare une chaîne de caractère
05 var surnom: string = 'Green Lantern';
06
07 // On déclare une variable qui correspond à une classe de notre
08 // application !
09 var greenLantern: Heros = new Heros(pointDeVie, surnom);
10
11 // On peut créer un autre héros de type
12 var superMan: Heros = new Heros(pointDeVie, 'Superman');
13
14 // Je peut même créer un tableau de héros,
15 // qui contient tous les héros de mon application !
16 var heros: Array<Heros> = [greenLantern, superMan];
```

Le typage de nos variables est très pratique, puisque cela vous permet d'être sûrs du type des variables que l'on utilise. 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 caractère à la variable point de vie :

```
01 pointDeVie = 'une-chaine-de-caractere';
```

Ensuite essayez de compiler ce code :

```
01 tsc test.ts
```

Vous obtiendrez alors cette magnifique erreur dans votre console :

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

L'erreur est explicite, à la *ligne 7* 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 :

```
01 // Ce code ne causera pas d'erreur car pointDeVie est bien de type number
02 pointDeVie = 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*'.

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* :

```
01 // Un constructeur pour notre classe Heros
02 // on spécifie le type de retour après les ':', ici un Heros
03 function creerHeros(pointDeVie: number, surnom: string): Heros {
04   var heros = new Heros();
05   heros.pointDeVie = pointDeVie;
06   heros.surnom = surnom;
07
08   return heros;
09 }
```

Cette fonction doit retourner une instance de la classe *Heros*, 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 '?'.

```
01 // Le '?' indique que 'planeteOrigine' est facultatif.  
02 function creerHeros(pointDeVie: number, surnom: string,  
03 planeteOrigine?: string): Heros {  
04  
05 var heros = new Heros();  
06 heros.pointDeVie = pointDeVie;  
07 heros.surnom = surnom;  
08 if(planeteOrigine) this.planeteOrigine = planeteOrigine;  
09 return heros;  
10 }
```

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 *ma-classe.ts* quelque part sur votre ordinateur, et ajoutez-y le code TypeScript suivant :

```
01 // ma-classe.ts  
02 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 :

```
01 mon-repertoire > tsc --t ES5 test.ts
```

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

```
01 // Ma classe en ES5 (ma-classe.js)  
02 var Test = (function () {  
03     function Test() {}  
04     return Test;  
05 }());
```

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 reconnaîtrez une IIFE.

IIFE: Immediately-invoked Function Expression. Ce sont des fonctions qui s'exécutent immédiatement, et dans un contexte privé. Jetez un coup d'oeil au cours de JavaScript sur OpenClassroom si vous ne vous rappelez plus de quoi il s'agit.

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

```
01 mon-repertoire > tsc --t ES6 test.ts
```

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

```
01 // Ma classe en ES6 (ma-classe.js)
02 class Test {
03 }
```

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 pourtant c'est parfaitement normal. Vous savez pourquoi ? Parceque 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 ! 😊

Les Décorateurs

Vous vous rappelez des méta-données 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 méta-données : 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:

```
01 // Ci-dessous les décorateurs TypeScript
02 @Component({
03   selector: 'mon-composant',
04   template: 'mon-template.html'
05 })
06 export class MonComposant {}
```

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.

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

Conclusion

TypeScript apporte évidemment d'autres fonctionnalités : les valeurs énumérées, les interfaces, la généricité, ...

TypeScript est donc un méta-langage qui est surtout connu pour apporter, entre autre, le typage à JavaScript, mais il s'agit également un transpilateur, capable de générer du code vers ES5 ou ES6 !

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](#). 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 retrouverez 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 est possible d'utiliser JavaScript, Dart, ou TypeScript pour ses développements avec Angular.
- Angular a été développé avec TypeScript et il est fortement recommandé d'adopter TypeScript.
- 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 *WebComponents*, 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 voyez de quoi il s'agit. Il s'agit d'un chapitre 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 ! 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



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

Chapitre 5

Premier pas avec Angular

Document checking completed. No errors or warnings to show.

Commencer par "Hello, world!"

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

Nous allons réaliser un superbe "*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 une application.
- Dire à Angular avec quels composants nous souhaitons démarrer notre application. Pour nous ce sera facile, car nous n'auront qu'un seul composant, le composant racine.
- Ecrire une simple page index.html qui contiendra notre application.

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.

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



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



II. Acquérir les bases de 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 chaque élément qui constitue une application Angular...



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



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 **yntaxe des templates** avec Angular...



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



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 ! 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



Chapitre 9

Transformer les données avec 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 2016 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...



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



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. 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



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. 😌



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



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 données et ces opérations.

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 ! 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



III. 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 ! 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



Chapitre 14

Formulaires pilotés par le modèle

Le chapitre précédent nous a donné un aperçu du processus de création des formulaires. Les formulaires pilotés par le template sont très pratiques pour développer des formulaires de petite taille, dans le cadre d'applications qui ne sont pas appelées à évoluer souvent.

Cependant le code coté template était plutôt verbeux, et il aurait été compliqué de modifier le formulaire sans y passer beaucoup de temps. De plus, le formulaire n'était pas vraiment testable.

On ne peut pas toujours justifier le temps de développer ces formulaires *fait à la main*. Surtout si vous avez besoin de développer beaucoup de formulaires, et qu'il faut les modifier régulièrement car les besoins de vos utilisateurs changent ! 😊

Il peut être plus économique de créer les formulaires dynamiquement, basés sur des modèles d'objets métiers de votre application. C'est ce que nous allons voir dans ce chapitre !



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



Chapitre 15

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 ! 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



Chapitre 16

Utiliser le module HTTP

Après un chapitre plutôt théorique sur la programmation réactive, nous allons maintenant mettre en pratique nos apprentissages pour améliorer notre application de pokémons ! En effet, pour le moment, cette application est assez simple et permet seulement d'édition des pokémons.

Ce que nous aimerais faire, c'est communiquer avec un serveur distant pour récupérer les pokémons, les éditer et sauvegarder ces changements sur le serveur. En prime, on pourrait même ajouter un champ de recherche sur la page d'accueil, pour permettre à l'utilisateur de retrouver un pokémon plus facilement. Qu'est-ce que vous en pensez ? 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



Chapitre 17

Authentification

Pour le moment, notre application n'est pas très fiable en terme 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 à 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 ! 😊



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



Chapitre 19

Tester votre application

La réalisation d'une application comprend plusieurs phases : recueillir le besoin, réfléchir aux technologies que l'on va utiliser, réaliser le développement du logiciel, et ensuite les tests ! Que vont dire vos utilisateurs si votre application est pleine de bugs, et qu'il est impossible de s'en servir ?

Il existe aussi une autre phase, la mise en production, que nous verrons au chapitre suivant.

La phase de développement d'une application n'est donc qu'une des étapes, bien qu'elle constitue l'essentiel de ce qu'on a vu dans ce cours ! 😊 Maintenant, nous allons voir comment tester de manière rigoureuse une application.



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



Chapitre 19

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 reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/



IV. Annexes



Le reste de ce contenu n'est pas disponible car il s'agit d'une version gratuite ! :/

