

APPRENDRE ANGULAR

Développez facilement votre première application Angular avec TypeScript

DIENY, SIMON

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 suivants du Code pénal.

Droits d'auteur 2018 © Simon Dieny

Tous droits réservés

TABLE DES MATIERES

Table des matières	2
Préambule.....	3
Avant-propos	4
Avant de commencer le cours ... (Important !)	6
A propos de l'auteur.....	8
Partie 1.....	9
Découvrir Angular	9
Chapitre 1 : Présentation de Angular.....	10
Chapitre 2 : ECMAScript 6	18
Chapitre 3 : Découvrir TypeScript.....	28
Chapitre 4 : Les Web Components	36
Chapitre 5 : Premiers pas avec Angular.....	45
Questionnaire n°1	60
Partie 2	61
Acquérir les bases d'Angular	61
Chapitre 6 : Les composants.....	62

Préambule

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 pré-requis 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 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 réalisée avec Angular 6 est disponible en ligne à [cette adresse](#).

Voici un aperçu :

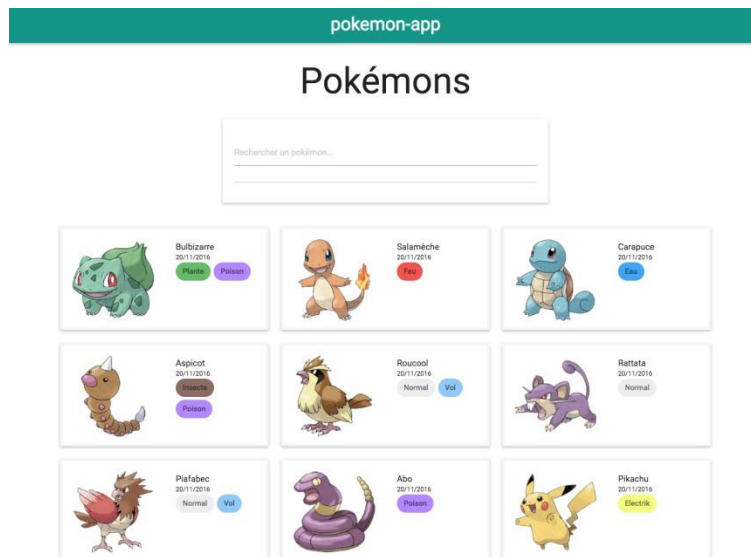


Figure 1 - Voici l'application que vous réaliserez pendant ce cours !

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 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.

PS : La version actuelle du cours correspond à la version 6.0.3, 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 comment 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, l'authentification, le déploiement...

IV. Annexes : Composées 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/ng6-pokemons-app>

Ce dépôt est organisé avec des tags Git, et vous donne accès à une correction complète à la fin de chaque chapitre :

```
Git checkout 4.directives // Correction du chapitre sur les directives
```

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

Si vous ne connaissez ou n'utilisez pas Git, consultez directement la correction en ligne grâce au lien ci-dessus.

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 :

<https://awesome-angular.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 !

Questionnaires

Il y a en tout trois questionnaires dans ce cours disponibles en ligne :

<https://awesome-angular.com/ebook/>

Google is everywhere

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

Règles typographiques

Ce cours respecte la typographie 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 question que vous pouvez vous poser, et à laquelle nous apporterons une réponse !

Ci-dessous se trouve un extrait de code. Les numéros de lignes sont indiqués à gauche :

```
01 function Vehicule(couleur, nombreRoueMotrice) {  
02   this.couleur = couleur;  
03   this.nombreRoueMotrice = nombreRoueMotrice;  
04   this.moteurAllumer = false;  
05 }
```

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 habituel.

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.

Ça bouge !

Angular évolue constamment, comme beaucoup de technologies dans le monde du développement Web. Ce cours est stabilisé pour la version 6.0.3 du Framework. Vous pouvez vous former sans problèmes sur des versions ultérieures (6.1.0, 6.2.0, etc), car les évolutions d'Angular concernent des éléments très spécifiques, et qui n'ont pas d'impact sur les éléments de base abordés dans ce cours.

Les différences apportées dans ce cours par rapport à la version 5.0.0, sont disponibles sur [mon blog](#).

Je suis bloqué, ... je ne trouve rien sur Google, et personne ne m'aide sur les forums !

La courbe d'apprentissage d'Angular n'est pas simple. Mais c'est en cherchant et en trouvant les réponses à vos questions par vous-même, que vous réussirez. Toutefois, il arrive à tout le monde d'être bloqué, et je peux vous donner un [petit coup de main](#) si nécessaire.

De plus, si vous avez une remarque sur l'ouvrage (exemple de code pas très parlant, coquilles, etc), faites en moi part, et je pourrai continuer à améliorer ce cours pour ceux qui apprendront après vous.

A propos de l'auteur



Figure 2 - L'auteur, Simon Diény

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. Et pourquoi pas d'aller plus loin avec Angular : Ionic, NativeScript, ElectronJS, etc.

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

<https://awesome-angular.com/blog/>

Bon, ce n'est pas tout, mais quand commence t'on ? Et bien, maintenant ! C'est parti !

Partie 1

Découvrir 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 ? 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.

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.

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 :

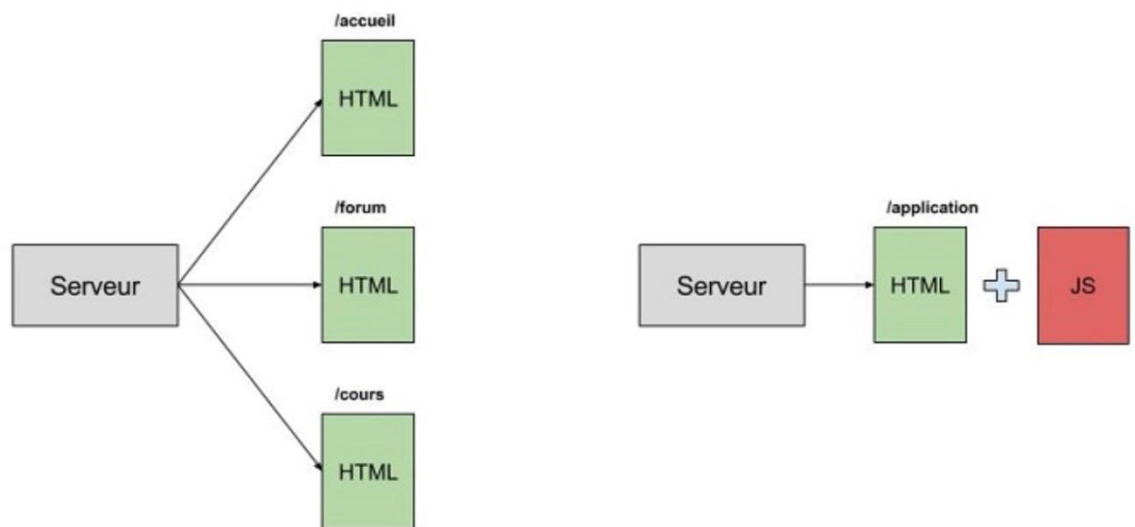


Figure 3 - L'architecture d'un site web et d'une application web

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 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 plaît 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 !" ».

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 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. **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, 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 qu'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 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 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.

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.

<i>Hé mais attends, ... depuis quand il y a des classes en JavaScript ?</i>

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.

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 vous demande pas de tout 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 bibliothèques, 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'utiliserai 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, la bibliothèque `@angular/core` est une bibliothèque 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 bibliothèque. Dans ce cas on renseigne un chemin relatif à la place du nom de la bibliothèque :

```
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.

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 améliorer l'organisation de ce code, et les développeurs se sont tournés 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.

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

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 toujours 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 reconnaissez 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 objet 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.
- **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, remplacez *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 une grande respiration, et à reprendre lentement la partie qui vous bloque, depuis le début. 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.

C'est pourquoi il est important de s'adopter au fur et à mesure et ne pas baisser les bras.

Pour les curieux, sachez que la documentation officielle d'Angular se trouve sur le site <https://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 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 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 transpilateur pour convertir leur code ES6 en du code ES5. Ainsi leur code peut être interprété 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 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.

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 !

Les classes

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

```
01 function Vehicle(color, drivingWheel) {
02   this.color = color;
03   this.drivingWheel = drivingWheel;
04   this.isEngineStart = false;
05 }
06 Vehicle.prototype.start = function start(){
07   this.isEngineStart = true;
08 }
09 Vehicle.prototype.stop = function stop(){
10   this.isEngineStart = false;
11 }
```

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](#), en français.

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 Vehicle {
02   constructor(color, drivingWheel) {
03     this.color = color;
04     this.drivingWheel = drivingWheel;
05     this.isEngineStart = false;
06   }
07   start() {
08     this.isEngineStart = true;
09   }
10   stop() {
11     this.isEngineStart = false;
12   }
13 }
```

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 !

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 Vehicle (color, drivingWheel) {
02   this.color = color;
03   this.drivingWheel = drivingWheel;
04   this.isEngineStart = false;
05 }
06 Vehicle.prototype.start = function start() {
07   this.isEngineStart = true;
08 }
09 Vehicle.prototype.stop = function stop() {
10   this.isEngineStart = false;
11 };
12 // Une voiture est un véhicule.
13 function Car(color, drivingWheel, seatings) {
14   Vehicle.call(this, color, drivingWheel);
15   this.seatings = seatings;
16 }
17 Vehicle.prototype = Object.create(Vehicle.prototype);
18 // Une moto est un véhicule également.
19 function Motorbike (color, drivingWheel, unleash) {
20   Vehicle.call(this, color, drivingWheel);
21   this.unleash = unleash;
22 }
23 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.

```
01 class Vehicle {
02   constructor(color, drivingWheel, isEngineStart = false) {
03     this.color = color;
04     this.drivingWheel = drivingWheel;
05     this.isEngineStart = isEngineStart;
06   }
07   start() {
08     this.isEngineStart = true;
09   }
10   stop() {
11     this.isEngineStart = false;
12   }
13 }
14 class Car extends Vehicle {
15   constructor(color, drivingWheel, isEngineStart = false, seatings) {
16     super(color, drivingWheel, isEngineStart);
17     this.seatings = seatings;
18   }
19 }
20 class Moto extends Vehicle {
21   constructor(color, drivingWheel, isEngineStart = false, unleash) {
22     super(color, drivingWheel, isEngineStart);
```

```
23  this.unleash = unleash;
24  }
25  }
```

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

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 :

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

```
01 function someFunction (defaultValue) {
02   defaultValue = defaultValue || undefined;
03   return defaultValue;
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 vaut un par défaut :

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

A 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 :

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

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 :

```
01 var x = 1;
02 if(x < 10) {
03   let v = 1;
04   v = v + 21;
05   console.log(v);
06 }
07 // v n'est pas définie, car v a été déclarée avec 'let' et non 'var'.
08 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 = 3.14 // Erreur : la valeur de PI ne peut plus être modifié
```

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 :

```
01 const MATHEMATICAL_CONSTANTS = {PI: 3.141592, e: 2,718281};
02 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* !

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 createCar() {  
02   let color = 'green';  
03   let wheel = 4;  
04   return { color, wheel };  
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 createCar() {  
02   let color = 'green';  
03   let wheel = 4;  
04   return { color: color, wheel: wheel };  
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 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 :

```
01 getUser(userId, function(user) {  
02   getFriendsList(user, function(friends) {  
03     showFriends(friends);  
04   });  
05 });
```

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 :

```
01 getUser(userId)  
02   .then(function(user) {  
03     getFriendsList(user);  
04   })  
05   .then(function(friends) {  
06     showFriends(friends);  
07   });
```


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 :

```
01 let getUser = function(userId) {
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     // j'extrais les données de l'utilisateur :
07     let user = response.data.user;
08     if(response.status === 200) {
09       resolve(user);
10     } else {
11       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 :

```
01 getUser(userId)
02   .then(function (user) {
03     console.log(user); // 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 de simplifier l'écriture des fonctions anonymes : cela se combine parfaitement avec l'utilisation des promesses.

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 :

```
01 class Person {
02   constructor(firstName, email, button) {
03     this.firstName = firstName;
04     this.email = email;
05     button.onclick = function() {
06       // ce 'this' fait référence au bouton,
07       // et non à une instance de Personne.
```

```

08     sendEmail(this.email);
09   }
10 }
11 }

```

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. 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, souvent nommé *that*, comme ceci :

```

01 class Person {
02   constructor(firstName, email, button) {
03     this.firstName = firstName;
04     this.email = email;
05     // 'this' fait référence ici à l'instance de Personne
06     var that = this;
07
08     button.onclick = function() {
09       // 'that' fait référence à la même instance de Personne
10       sendEmail(that.email);
11     }
12   }
13 }
14 }

```

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 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 :

```

01 someArray = [1, 2, 3, 4, 5];
02 let doubleArray = someArray.map((n) => n*2);
03 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 :

```

01 // un traitement asynchrone en quelques lignes !
02 getUser(userId)
03 .then(user => getFriendsList(user))
04 .then(friends => showFriends(friends));

```

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 = {rank: 1, name: 'Zlatan'};
02 // Je crée un nouveau dictionnaire
03 let players = new Map();
04 // J'ajoute l'objet 'zlatan' à la clé '1'
05 players.set(zlatan.rank, 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 players = new Set(); // Je crée une nouvelle liste
02 players.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 tout 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 // affiche le nombre d'éléments dans la collection
02 players.size;
03 // Dictionnaire: affiche si oui ou non,
04 // le dictionnaire contient la clé correspondant au rang de Zlatan.
05 players.has(zlatan.rang);
06 // Liste: affiche si oui ou non, la liste contient le joueur Zlatan.
07 players.has(zlatan);
08 // Dictionnaire: supprime un élément d'après une clef.
09 players.delete(zlatan.rang);
10 // Liste: supprime l'élément passé en paramètre.
11 players.delete(zlatan);
```

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 :

```
01 let someText = "duTexte";
02 let someOtherText = "unAutreTexte";
03 let embarrassingString = someText;
04 embarrassingString += " blabla";
05 embarrassingString += someText;
06 embarrassingString += "blabla";
07 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.

```
01 // On peut écrire des strings sur plusieurs ligne grâce au backtick
02 let severalLinesString = `bla
03   blablalbalblalballb
04   balblablalabla
05   b
06   ablablalabbl`;
07 // .. mais pas avec des guillemets !
08 // Regardez la couleur syntaxique, vous comprendrez que ce code risque
09 // de lever une erreur !
10 let severalLinesStringWithError = "bla
11   blba
12   blbla
13   blabla"
```

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

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

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

Conclusion

Nous avons vu dans ce chapitre plusieurs changements majeurs apportés par ES6 à JavaScript, et qui nous servirons 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-créé 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 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 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 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 tous 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 tous de suite le lien entre TypeScript et ce qui nous intéresse : Angular.

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 plusieurs versions de la documentation vous sont proposées :



Figure 4 - Il est possible d'utiliser Dart 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 le 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.

(Oubliez donc AngularDart pour le moment).

"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 sayHello(person) {
03   return "Hello, " + person;
04 }
05 // Création d'une variable "Jean"
06 var Jean = "Jean";
07 // On ajoute dans notre la page HTML un message
08 // pour affiche "Hello, Jean".
09 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 leur versions respectives grâce aux commandes suivantes :

```
node -v
v8.11.2
```

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

```
npm -v
5.9.0
```

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 8 pour Node. 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, 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 2.9.1
```

Voilà, c'est aussi simple que ça !

Là aussi, la version a de l'importance. Essayez d'avoir au moins la version 2.7, 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 :

```
01 // Fonction qui retourne un message de bienvenu
02 function sayHello(person) {
03   return "Hello, " + person;
04 }
05 // Création d'une variable "Jean"
06 var Jean = "Jean";
07 // On ajoute dans notre la page HTML un message
08 // pour affiche "Hello, Jean".
09 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 comment ça marche !

*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 !

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 lifePoint: number = 100;
03 // On déclare une chaîne de caractère
04 var name: string = 'Green Lantern';
```



```

05 // On déclare une variable qui correspond
06 // à une classe de notre application !
07 var greenLantern: Hero = new Hero(lifePoint, name);
08 // On peut créer un autre héros de type Hero
09 var superMan: Hero = new Hero(lifePoint, 'Superman');
10 // Je peux même créer un tableau de héros,
11 // qui contient tous les héros de mon application !
12 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 :

```

01 var lifePoint: number = 100;
02 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 :

```

01 // Ce code ne causera pas d'erreur car lifePoint est bien de type number.
02 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'.

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 Hero
02 // On spécifie le type de retour après les ':', ici Hero.
03 function createHero(lifePoint: number, name: string): Hero {
04   var hero = new Hero();
05   hero.lifePoint = lifePoint;
06   hero.name = name;
07   return hero;
08 }

```

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 '?' :

```
01 // Le '?' indique que le paramètre 'planet' est facultatif.
02 function createHero(lifePoint: number, name: string,
03 planet?: string): Hero {
04   var hero = new Hero();
05   hero.lifePoint = lifePoint;
06   hero.name = name;
07   if(planet) hero.planet = planet;
08
09   return hero;
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 *test.ts* quelque part sur votre ordinateur, et ajoutez-y le code TypeScript suivant :

```
01 // test.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 :

```
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 :

```
01 // Ma classe en ES5 (test.js)
02 var Test = (function () {
03   function Test() {
04   }
05   return Test;
06 }());
```

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* :

```
01 // Ma classe en ES6 (test.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 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 !

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 :

```
01 // // Exemple d'utilisation des décorateurs avec TypeScript  
02 @Component({  
03   selector: 'my-component',  
04   template: 'my-template.html'  
05 })  
06 export class MyComponent {}
```

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 autres, le typage à JavaScript, mais il s'agit également d'un transpileur, 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 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 est possible d'utiliser JavaScript, ou 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 !

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 !

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 :

```
01 <!-- Je crée une balise au hasard -->
02 <une-balise-inventee></une-balise-inventee>
03 <style>
04  /* Je rajoute un peu de style à ma nouvelle balise */
05  une-balise-inventee {
06    width: 200px;
07    height: 50px;
08    border: 1px solid orange;
09  }
10 </style>
11 <script>
12  // Ce script permet d'afficher le nombre de
13  // balises personnalisées sur la page
14  var elementPerso =
15    document.getElementsByTagName("une-balise-inventee");
16  var quantite = elementPerso.length;
17  alert("Il y a " + quantite +
18    " éléments <une-balise-inventee> dans ce document.");
19 </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, '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 :

```
01 <!doctype html>
02 <html>
```

```

03 <head>
04   <meta charset="UTF-8">
05   <title>Test</title>
06 </head>
07 <body>
08   <script>
09     // On crée une classe pour notre élément personnalisé
10     // et on lui ajoute un template avec la méthode 'innerHTML'.
11     class SuperHero extends HTMLElement {
12       constructor() {
13         super();
14         this.innerHTML = '<h1>Superman</h1>';
15       }
16     }
17
18     // On définit notre élément personnalisé avec la méthode 'define'.
19     customElements.define('super-hero', SuperHeros);
20     // On ajoute notre élément sur la page web !
21     document.body.appendChild(new SuperHero());
22   </script>
23 </body>
24 </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>

```

Figure 5 - Console du navigateur lors de l'inspection de mon titre 'Superman'

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

Document checking completed. No errors or warnings to show.

Figure 6 - 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 !

Attends-tu 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 se passe-t-il si vous avez sur votre page un élément A qui indique que tous ces paragraphes doivent être écrits en jaune, et un élément B qui spécifie que tous les paragraphes doivent être écrits 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.

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 !

Les spécifications du Shadow DOM

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 :

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

Et maintenant ajoutons un Shadow DOM sur ce paragraphe :

```
01 // On crée un nouveau Shadow DOM sur un élément de notre page,
02 // qui possède l'identifiant 'paragraphe-shadow'
03 var shadow =
04   document.querySelector('#paragraphe-shadow').attachShadow({mode: 'open'});
05 // Pour l'instant notre Shadow DOM est vide,
06 // mais nous pouvons lui ajouter du contenu.
07 shadow.innerHTML = "<p id='shadow'>Salut, Shadow DOM !</p>";
08 // Si on recherche notre contenu caché depuis la console du navigateur,
09 // alors la commande suivante ne retourne rien
10 // car le DOM n'a pas accès au Shadow DOM !
11 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 défini 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'* :

```
01 // On ajoute un peu de style à notre Shadow DOM
02 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 !

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 !

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 réutilisé 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 :

```
01 <template id="super-heros">
02 <style>
03   h1 { color: green; };
04 </style>
05   <h1>Green Lantern</h1>
06 </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* !

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 :

```
01 <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 Components, comme la librairie [Polymer](#) de Google, et qui vous permet de faire ça :

```

01 <!-- Polyfill pour les Composants Web,
02     afin de supporter les navigateurs plus anciens -->
03 <script
04   src="components/webcomponentsjs/webcomponents-lite.min.js"></script>
05 <!-- Un import de HTML -->
06 <link rel="import" href="components/google-map/google-map.html">
07 <!-- On utilise l'élément personnalisé importé ! -->
08 <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 !

Voici un [lien](#) sur lequel vous pouvez voir quelles spécifications sont actuellement supportées par les navigateurs les plus populaires.

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

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.

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 !

Choisir un environnement de développement

Alors de quoi allons-nous avoir besoin ? Et bien je vous conseille de lancer votre éditeur de texte favoris et de commencer à faire chauffer votre cerveau.

Il y a plusieurs IDEs qui supportent TypeScript et qui pourront vous aider lors de vos développements :

1. Visual Studio Code (Recommandé par Microsoft pour les développements Angular, évidemment)
2. WebStorm (Payant mais [licence de 1 an offerte](#) si vous êtes étudiant).
3. Sublime Text avec [ce plugin](#) à installer.
4. 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'utilise Atom dans ce cours, mais vous êtes libres de travailler avec l'outil que préférez. Bon, on commence quand, pour de vrai ?

On va commencer par créer un dossier vide, qui sera le dossier racine de notre projet. Sachez que ce que nous allons développer maintenant, servira de base pour le reste du cours. En gros, nous commençons notre projet dès maintenant ! Mais rassurez-vous, on va commencer par la base de la base.

Créons donc un dossier nommé *pokemon-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.

Démarrer un projet Angular

Il y a plusieurs moyens de démarrer un projet Angular, certains sont plus rapides que d'autres. Par exemple, le projet [angular CLI](#) vous permet de mettre en place un nouveau projet Angular en une seule ligne de commande ! Cependant, je préfère partir d'un dossier vide, pour que vous puissiez voir le processus de création d'une nouvelle application Angular en partant de zéro.

Pour commencer nous allons avoir besoin d'ajouter trois fichiers de configuration dans le dossier racine de notre projet. Je vais détailler le rôle de chacun d'entre eux.

Le fichier package.json

Ce fichier doit vous être familier : il permet de décrire les dépendances d'un projet pour le Node Package Manager : Ajoutez le fichier *package.json* à la racine de notre projet, avec le contenu ci-dessous.

```
01 {
02   "name": "ng6-pokemon-app",
03   "version": "1.0.0",
04   "description": "An awesome application to handle some pokemons.",
05   "scripts": {
06     "build": "tsc -p src/",
07     "build:watch": "tsc -p src/ -w",
08     "build:e2e": "tsc -p e2e/",
09     "serve": "lite-server -c=bs-config.json",
10     "serve:e2e": "lite-server -c=bs-config.e2e.json",
11     "prestart": "npm run build",
12     "start": "concurrently \"npm run build:watch\" \"npm run serve\"",
13     "pree2e": "npm run build:e2e",
14     "e2e": "concurrently \"npm run serve:e2e\" \"npm run protractor\" --kill-
others --success first",
15     "preprotractor": "webdriver-manager update",
16     "protractor": "protractor protractor.config.js",
17     "pretest": "npm run build",
18     "test": "concurrently \"npm run build:watch\" \"karma start karma.conf.js\"",
19     "pretest:once": "npm run build",
20     "test:once": "karma start karma.conf.js --single-run",
21     "lint": "tslint ./src/**/*.ts -t verbose"
22   },
```

```

23 "private": true,
24 "dependencies": {
25   "@angular/animations": "^6.0.3",
26   "@angular/common": "^6.0.3",
27   "@angular/compiler": "^6.0.3",
28   "@angular/core": "^6.0.3",
29   "@angular/forms": "^6.0.3",
30   "@angular/http": "^6.0.3",
31   "@angular/platform-browser": "^6.0.3",
32   "@angular/platform-browser-dynamic": "^6.0.3",
33   "@angular/router": "^6.0.3",
34   "core-js": "^2.5.4",
35   "rxjs": "^6.2.0",
36   "rxjs-compat": "^6.2.0",
37   "systemjs": "0.19.40",
38   "zone.js": "^0.8.26"
39 },
40 "devDependencies": {
41   "typescript": "~2.7.2",
42   "@angular/language-service": "^6.0.3",
43   "@types/jasmine": "~2.8.6",
44   "@types/jasminewd2": "~2.0.3",
45   "@types/node": "~8.9.4",
46   "codifyer": "~4.2.1",
47   "jasmine-core": "~2.99.1",
48   "jasmine-spec-reporter": "~4.2.1",
49   "karma": "~1.7.1",
50   "karma-chrome-launcher": "~2.2.0",
51   "karma-coverage-istanbul-reporter": "~2.0.0",
52   "karma-jasmine": "~1.1.1",
53   "karma-jasmine-html-reporter": "^0.2.2",
54   "protractor": "~5.3.0",
55   "ts-node": "~5.0.1",
56   "tslint": "~5.9.1"
57 }
58 }

```

Il y a trois parties intéressantes à remarquer dans ce fichier *package.json* :

1. **Les scripts** : un certain nombre de scripts prédéfinis sont listés à partir de la ligne 5. Par exemple, *start* permet de démarrer notre application et *lite* permet de démarrer un mini-serveur sur lequel tournera notre application Angular. Nous verrons comment utiliser certaines de ces commandes un peu plus tard dans ce chapitre.
2. **Les dépendances** : A partir de la ligne 24, apparaît une liste de toutes les dépendances de notre application. Par exemple, *@angular/router* a pour vocation de gérer les routes de notre application, et *@angular/forms* gère les formulaires.
3. **Les dépendances relatives au développement** : Ces dépendances sont listées à partir de la ligne 40, et concernent les dépendances dont nous n'aurons plus besoin quand notre application sera terminée. Par exemple, TypeScript (ligne 44) ne sera plus nécessaire une fois l'application terminée, étant donné que le navigateur ne pourra pas l'interpréter : le navigateur interprétera uniquement le JavaScript qui a été généré depuis le TypeScript compilé.

Si vous avez le temps, vous pouvez retrouver toutes les explications détaillées, ligne par ligne, de ce fichier, dans la [documentation officielle](#) sur le sujet.

Angular 6 nécessite au moins la version ~2.7.0 de TypeScript (et au minimum la version 6 de RxJS, qui est une librairie que je vous présenterai plus tard dans ce cours). C'est bien cette version de TypeScript que nous avons renseigné dans le fichier `package.json`.

Le fichier `bs-config.json`

Ce fichier contient la configuration du mini-serveur *lite-server* que nous utiliserons dans ce cours. Le fichier permet de déclarer le dossier qui contiendra le code de notre application, ici le dossier `src`, que nous allons créer bientôt, et indique le chemin vers les librairies installées dans le dossier `node_modules`. Créez donc le fichier `bs-config.json` à la racine de votre projet :

```
01 {
02   'server' : {
03     'baseDir' : 'src'
04     'routes' : {
05       '/node_modules' : 'node_modules'
06     }
07   }
08 }
```

Le fichier `systemjs.config.js`

System.js est la bibliothèque par défaut choisie par Angular pour charger les modules JavaScript, c'est-à-dire assembler de manière cohérente tous les fichiers de notre application : les fichiers propre à Angular, les librairies tierces et les fichiers de notre propre code.

Le plus important à retenir est l'élément *map* à la ligne 13. Cet objet nous permet de déclarer deux nouveaux éléments :

- Le dossier de notre application à la ligne 15, ci-dessous le dossier *app*.
- Le mapping de nos librairies : on lie un alias avec l'emplacement de la librairie. Par exemple à la ligne 18, on déclare l'alias `@angular/core`, puis on renseigne son emplacement dans le dossier `node_modules`. Cela nous permet dans notre application d'utiliser cet alias pour l'importation :

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

Voici le contenu du fichier `systemjs.config.js`, à placer, dans le dossier `src` de votre projet (Il s'agit d'une convention que respectent les projets Angular). Créez donc un dossier `src`, puis placez le fichier `systemjs.config.js` à l'intérieur :

```
01 /**
02  * La configuration de SystemJS pour notre application.
03  * On peut modifier ce fichier par la suite selon nos besoins.
04  */
05 (function (global) {
06   System.config({
07     paths: {
08       // définition d'un raccourcis, 'npm' pointera vers 'node_modules'
09       'npm:': 'node_modules/'
10     },
11     // L'option map permet d'indiquer à SystemJS l'emplacement
12     // des éléments à charger
13     map: {
14       // notre application se trouve dans le dossier 'app'
15       app: 'app',
```

```

16 // packets angular
17 '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
18 '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
18 '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
20 '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-
browser.umd.js',
21 '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
22 '@angular/common/http': 'npm:@angular/common/bundles/common-http.umd.js',
23 '@angular/http': 'npm:@angular/common/bundles/common-http.umd.js',
24 '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
25 '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
26 // autres librairies
27 'tslib': 'npm:tslib/tslib.js',
28 'rxjs': 'npm:rxjs',
29 'angular-in-memory-web-api': 'npm:angular-in-memory-web-api/bundles/in-
memory-web-api.umd.js'
30 },
31 // L'option 'packages' indique à SystemJS comment charger
32 // les paquets qui n'ont pas de fichiers et/ou extensions renseignés
33 packages: {
34   app: {
35     defaultExtension: 'js'
36   },
37   rxjs: {
38     main: 'index.js',
39     defaultExtension: 'js'
40   },
41   'rxjs/operators': {
42     main: 'index.js',
43     defaultExtension: 'js'
44   }
45 }
46 });
47 })(this);

```

A la ligne 15, nous indiquons que notre application est contenue dans le dossier *app*.

Le fichier tsconfig.json

Ce document est le fichier de configuration de TypeScript. Vous devez le placer dans le dossier *src* également. On peut définir un certain nombre d'éléments de configuration dans ce fichier, en voici quelques-uns :

- A la ligne 3, *target* a pour valeur *es5*, ce qui indique que notre code TypeScript sera compilé vers du code JavaScript ES5. On pourrait mettre *ES6* comme valeur pour générer du code JavaScript différent.
- A la ligne 12, *removeComments* indique à TypeScript que l'on ne souhaite pas que les commentaires de notre code soient retirés lors de la compilation. On pourrait mettre la valeur *true* pour obtenir le comportement opposé.

```

01 {
02   "compilerOptions": {
03     "target": "es5",
04     "module": "commonjs",
05     "moduleResolution": "node",
06     "sourceMap": true,
07     "emitDecoratorMetadata": true,

```

```
08 "experimentalDecorators": true,  
09 "lib": [ "es2015", "dom" ],  
10 "noImplicitAny": true,  
11 "suppressImplicitAnyIndexErrors": true,  
12 "removeComments": false  
13 }  
14 }
```

Les détails de la configuration de TypeScript sont indiqués sur la [documentation officielle](#), si vous souhaitez y revenir par la suite.

Pour plus de détails sur le fonctionnement des fichiers de configuration, je vous invite à consulter les annexes "Configuration de TypeScript" et "Dépendances d'un projet Angular".

Installer les dépendances

Maintenant, nous devons installer les bibliothèques que nous avons déclarées dans le `package.json`. Pour cela, utilisons la commande `npm install` à la racine de votre projet :

```
npm install
```

Cette commande devrait créer un dossier à la racine de notre projet nommé `node_modules`. Ce dossier contient toutes les dépendances dont nous avons besoin pour faire fonctionner notre projet.

Vérifiez que vous disposez d'au moins la version 8 de NodeJS, qui est la version minimum recommandée pour développer avec Angular 6.

Créer notre premier composant

Nous allons créer notre premier composant, depuis le temps que vous en entendez parler !

Cependant, nous allons organiser un peu notre code en mettant nos fichiers sources dans un dossier `app`, plutôt qu'avec nos fichiers de configuration à la racine de notre projet. Créer donc un dossier `app`, et placez-le dans le dossier `src`. À l'intérieur du dossier `app`, créer un fichier `app.component.ts`, qui contiendra notre premier composant :

```
01 import { Component } from '@angular/core';  
02  
03 @Component({  
04   selector: 'pokemon-app',  
05   template: `<h1>Hello, {{name}} !</h1>`,  
06 })  
07 export class AppComponent { name = 'Angular'; }
```

Nous allons prendre un peu de temps pour décrire ce fichier, car malgré sa taille réduite, ce fichier 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 :

```
01 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*.

Créer notre premier module

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.

Voici le code du module racine de notre application, *app.module.ts*, à placer dans notre dossier *app* :

```
01 import { NgModule } from '@angular/core';
02 import { BrowserModule } from '@angular/platform-browser';
03 import { AppComponent } from './app.component';
04
05 @NgModule({
06   imports: [ BrowserModule ],
07   declarations: [ AppComponent ],
08   bootstrap: [ AppComponent ]
```

```
09 })  
10 export class AppModule { }
```

Je vais présenter ce code rapidement. D'abord, on retrouve des importations en haut du fichier. Pour déclarer un module, on importe l'annotation *NgModule* contenue dans le cœur d'Angular lui-même; à la ligne 1.

Ensuite on importe le *BrowserModule*, 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 importe le seul composant *AppComponent* de notre application, que nous allons rattacher à ce module.

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

- **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).
- **declarations** : Une liste de tous les composants et directives qui appartiennent à ce module. Nous avons donc ajouté notre unique composant *AppComponent*.
- **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.

Créer un point d'entrée pour notre application

Vous vous rappelez que dans le fichier de configuration de *SystemJS*, nous avons indiqué à Angular que le point d'entrée de notre application serait le fichier *main.ts* ? Eh bien, c'est maintenant que nous allons le créer !

Créez donc un fichier *main.ts* avec le contenu suivant, dans le dossier *app* :

```
01 import { platformBrowserDynamic } from  
02 '@angular/platform-browser-dynamic';  
03 import { AppModule } from '../app/app.module';  
04 platformBrowserDynamic().bootstrapModule(AppModule)  
05 .catch(err => console.log(err));
```

Notre devons préciser dans ce fichier que notre application démarre dans un navigateur web et pas ailleurs : en effet, nous pourrions choisir d'utiliser Angular pour du développement mobile native avec [NativeScript](#) ou du développement *cross-platform* avec [Electron.js](#). Nous précisons donc que notre application est destinée aux navigateurs web, et que l'on désigne l'*AppModule* comme module racine, qui lui-même lancera l'*AppComponent*.

Sachez que le fichier *main.ts* sera très peu modifié lorsqu'on développera notre application par la suite, on peut dire qu'on le développe « une fois pour toute ».

Heu..., pourquoi créer main.ts, app.module.ts et app.component.ts dans 3 fichiers différents, tout ça pour lancer une application qui affiche un "Hello, Angular" ?

Votre question est légitime. Pour l'instant, ces trois fichiers sont assez simples et contiennent relativement peu de code.

Sachez pourtant que ces efforts supplémentaires nous ont permis de mettre en place notre application de la bonne manière. Le démarrage de notre application est indépendant de la description de notre module, qui est également indépendant des composants qui le constituent. Ces trois éléments doivent donc être séparés !

Héberger notre application

Il ne nous reste plus qu'un seul fichier pour pouvoir démarrer notre application, promis !

Vous vous rappelez ce que nous sommes en train de développer en terme d'architecture ? Une SPA (*Single Page Application*), c'est-à-dire que notre application ne sera composé que d'une page HTML avec beaucoup de code JavaScript pour dynamiser la page, récupérer des informations d'un serveur distant, etc... Et bien il nous manque cette fameuse page HTML. Créons-là tout de suite un fichier *index.html*, à la racine du projet (**et non dans le dossier app !**) :

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>Angular QuickStart</title>
05     <meta charset="UTF-8">
06     <meta name="viewport" content="width=device-width, initial-scale=1">
07     <!-- 1. Chargement des librairies -->
08     <!-- Polyfill(s) pour les anciens navigateurs -->
09     <script src="node_modules/core-js/client/shim.min.js"></script>
10     <script src="node_modules/zone.js/dist/zone.js"></script>
11     <script src="node_modules/systemjs/dist/system.src.js"></script>
12     <!-- 2. Configuration de SystemJS -->
13     <script src="systemjs.config.js"></script>
14     <script>
15       System.import('main.js').catch(function(err){ console.error(err); });
16     </script>
17   </head>
18   <!-- 3. Afficher l'application -->
19   <body>
20     <pokemon-app>Loading AppComponent content here ...</pokemon-app>
21   </body>
22 </html>
```

Ce fichier est une page HTML classique, qui contiendra toute notre application. J'ai essayé de tout décrire dans les commentaires. Remarquez la ligne 20, c'est à cet endroit que notre application va vivre, et que les templates de nos composants seront injectés !

Voici l'architecture de notre projet, à ce stade :

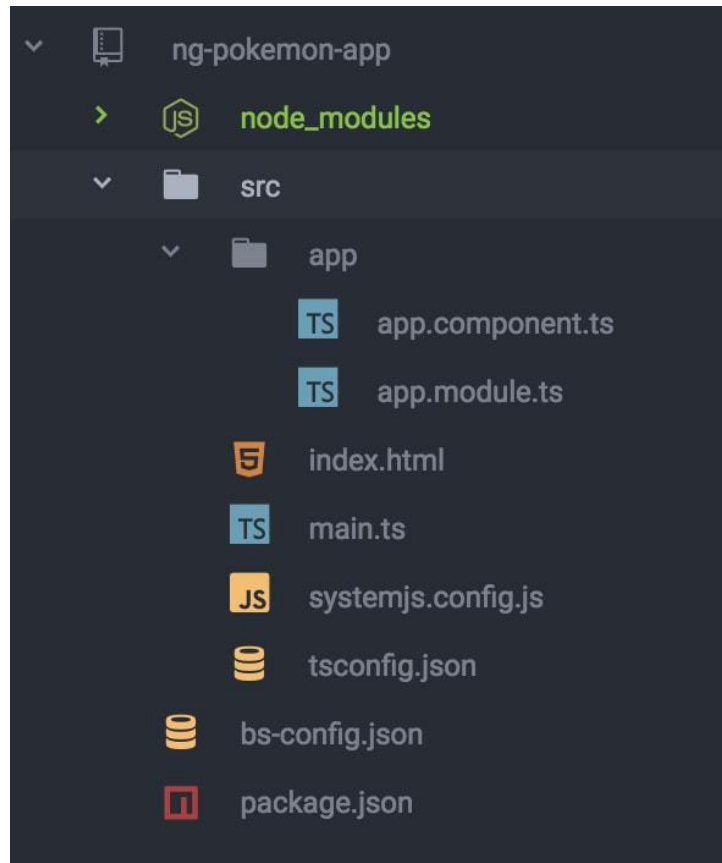


Figure 7 - L'architecture de notre projet

Bon, je vous rassure, on a fini ! Il ne nous reste plus qu'à démarrer notre application.

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 :

```
npm start
```

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, Angular2 !*" dans votre navigateur !

Hello, Angular !

Figure 8 - "Hello, World!" avec Angular !

Ca 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 !

Si toutefois vous avez des erreurs et que l'application ne se lance pas, afficher la console dans laquelle vous avez tapé la commande `npm start`, et scroller avec votre souris vers le haut jusqu'à tomber sur des messages d'erreurs. Ils devraient vous décrire à quelle ligne et dans quel fichier l'erreur se trouve.

En cas de pépin pour lancer la commande elle-même, essayer de lancer les deux commandes suivantes, dans deux consoles différentes :

1. `npm run tsc:w`

2. `npm run lite`

Alors je sais, tout ça pour ça !

Mais rassurez-vous, vous venez de faire quelque chose propre à beaucoup de Framework : installer le **socle** de notre application. Vous avez fait plus qu'afficher un message à vos utilisateurs, vous venez de mettre en place l'architecture de votre application, et ça, ça n'a pas de prix !

Mais revenons à la commande `npm start`, car sous des airs modestes, cette petite commande accomplit un travail important :

1. Elle compile tous nos fichiers TypeScript vers du JavaScript.
2. Elle lance l'application dans un navigateur et la met automatiquement à jour si nous modifions notre code !

Dans les librairies que nous avons installées au début de ce chapitre, *BrowserSync* s'occupe de mettre à jour notre application à chaque fois que nous modifions son code source, sans avoir à appuyer sur F5 !

Testons ça tout de suite, ouvrez le fichier de votre composant `app.component.ts`, et modifiez son code comme ceci :

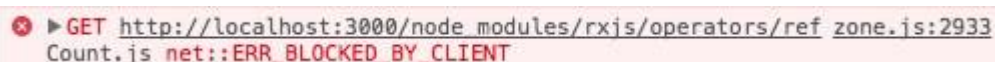
```
01 import { Component } from '@angular/core';
02
03 @Component({
04   selector: 'pokemon-app',
05   template: '<h1>Bonjour, {{name}} !</h1>'
06 })
07 export class AppComponent { name = 'Angular'; }
```

Si maintenant vous retournez dans le navigateur ou tournez votre application, vous verrez maintenant le message *"Bonjour, Angular !" s'afficher*, à la place de *"Hello, Angular !"*, le tout sans avoir à rafraîchir votre navigateur !

Pour que votre application soit mise à jour automatiquement, laissez la commande `npm start` tourner en continu pendant vos développements !

Le navigateur Chrome et l'extension Adblock

Si vous utilisez Chrome et que l'extension Adblock est activée, il se peut que votre application affiche une simple page blanche. Cela est dû à un conflit entre l'extension Adblock et la librairie `zone.js`, requise pour faire fonctionner Angular. En ouvrant la console de votre navigateur, voici ce que vous devriez obtenir :



```
✖ GET http://localhost:3000/node_modules/rxjs/operators/ref_count.js:2933
Count.js net::ERR_BLOCKED_BY_CLIENT
```

Figure 9 - Il existe un conflit entre l'extension Adblock et Zone.js

Pour remédier à ce problème, il suffit de désactiver l'extension Adblock lorsque vous développez sur votre machine. De toute façon, vous n'avez pas besoin de cette extension pour vos sites locaux, qui n'affichent pas de publicités.



Figure 10 - Pensez à désactiver Adblock lors de vos développements !

Nettoyer son dossier de développement

Si vous ouvrez votre IDE (le logiciel que vous utilisez pour effectuer vos développements), vous constatez qu'un certains nombres de fichiers se sont ajoutés dans le dossier *app*. Il s'agit de fichier *.js et *.js.map. Il s'agit des fichiers compilés par TypeScript et ce sont eux qui sont interprétés par le navigateur. Jusque-là tout va bien.

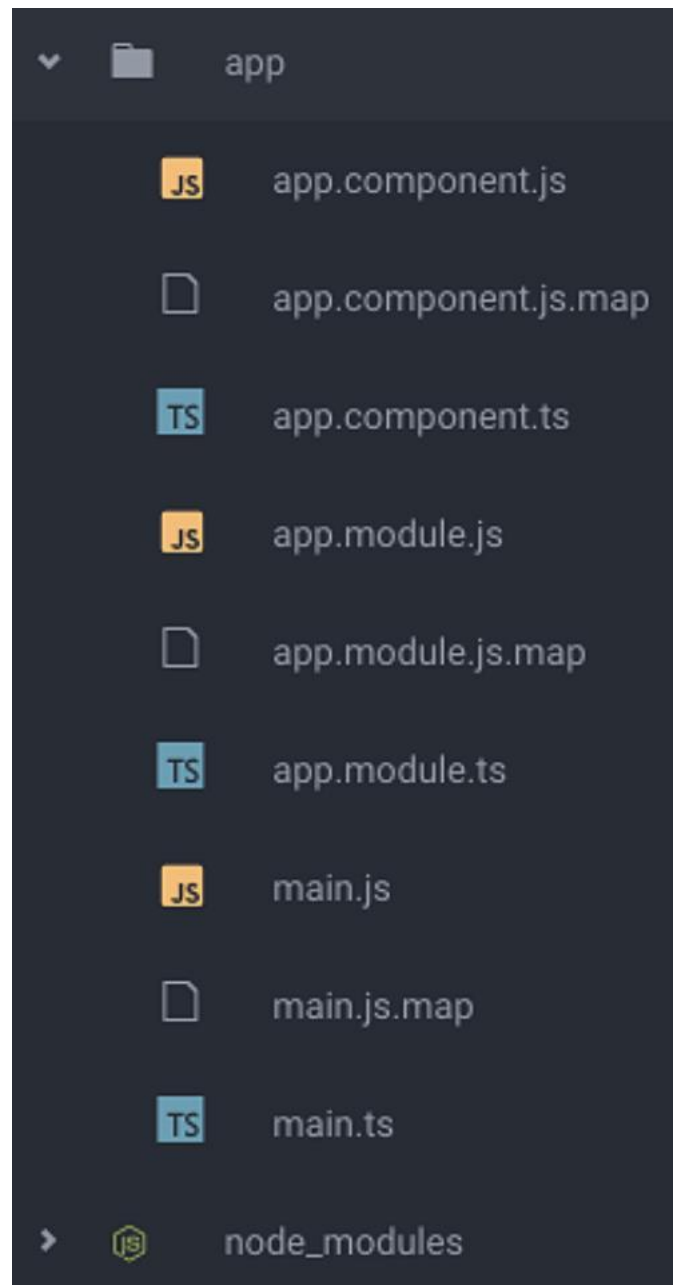


Figure 11 - Le dossier 'app' n'est très lisible

Le problème est que ces fichiers nous gênent : ils polluent notre dossier de développement *app* et le rendent moins lisible. Je vous propose de mettre tous les fichiers destinés au navigateur dans un dossier à part, le dossier */dist*. Les fichiers que nous utilisons pour nos développements (les fichiers TypeScript) resteront dans notre dossier de développement */app*.

Pour cela, nous allons dire à SystemJS de ne pas chercher à démarrer l'application depuis le dossier *app*, mais depuis le dossier *dist*. Mais avant, nous allons dire à TypeScript de compiler les fichiers vers ce nouveau répertoire, sinon il restera désespérément vide.

Premièrement, créer un dossier *dist* à la racine de votre application.

Ensuite, nous allons dire à TypeScript de pointer vers *dist*. Ouvrez le fichier de configuration *tsconfig.json* et ajoutez la ligne de configuration *outDir*, à la ligne 11 :

```

02 "compilerOptions": {
03   "target": "es5",
04   "module": "commonjs",
05   "moduleResolution": "node",
06   "sourceMap": true,
07   "emitDecoratorMetadata": true,
08   "experimentalDecorators": true,
09   "removeComments": false,
10   "noImplicitAny": false,
11   "outDir": "dist"
12 }
13 }

```

Maintenant, il ne nous reste plus qu'à dire à *System.js* de démarrer notre application par rapport aux fichiers JavaScript qui se trouvent dans le nouveau dossier *dist*. Ouvrez le fichier de configuration de *System.js* nommé *systemjs.config.js* et modifiez la ligne 5 ci-dessous :

```

01 (function (global) {
02   System.config({
03     // ...
04     map: {
05       app: 'dist/app', // <-- remplacez 'app' par 'dist/app' comme ceci !

```

Enfin, dans le fichier *index.html*, ajoutez la bonne inimportation pour SystemJS :

```

01 <!-- ... -->
02 <script>
03   System.import('dist/main.js').catch(function(err){ console.error(err); });
04 </script>
05 <!-- ... -->

```

Désormais relancez la commande *npm start*. Sans surprise, tout fonctionne comme avant (sinon reprenez les étapes ci-dessus). Si vous ouvrez à nouveau votre IDE favoris, vous constaterez que le dossier *dist* contient de nouveaux fichiers générés par TypeScript :

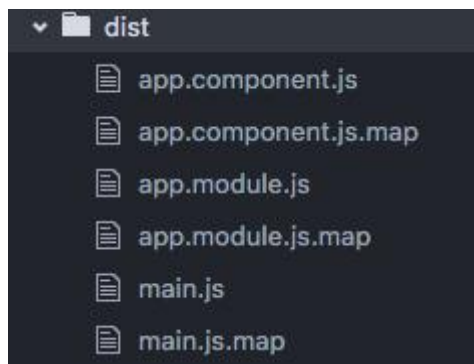


Figure 12 - Le dossier 'dist' contient de nouveaux fichiers

Parfait, les fichiers destinés au navigateur ne polluent plus notre dossier de développement *app*. Il ne nous reste plus qu'à supprimer les fichiers suivants dans *app*, car nous n'en avons plus besoin :

- *app.component.js*
- *app.component.js.map*
- *app.module.js*
- *app.module.js.map*

- *main.js*
- *main.js.map*

Ne vous inquiétez, l'application fonctionnera toujours !

D'ailleurs si vous ne me croyez pas, aller hop, un petit *npm start* !

Le nom de dossier '*dist*' signifie 'distribution' : le contenu de ce dossier est destiné à la production. En effet, une fois que nous aurons finis de développer notre application, nous n'allons pas déployer les fichiers TypeScript, qui seront inutiles. Nous verrons comment déployer une application Angular en production plus tard dans ce cours.

Pour ceux qui versionnent leur code avec [Git](#), voici le contenu du *.gitignore* pour notre application Angular :

/dist

/node_modules

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

- *SystemJS* est la bibliothèque par défaut choisie par Angular pour charger les modules.
- 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*.
- L'ordre de chargement de l'application est le suivant : *index.html* > *main.ts* > *app.module.ts* > *app.component.ts*.
- Le fichier *package.json* initial est fourni avec des commandes prêtes à l'emploi comme la commande *npm start*, qui nous permet de démarrer notre application web.

Questionnaire n°1

L'objectif de ce questionnaire est de valider les acquis théoriques de cette première partie, ainsi que la mise en place du socle pour une application Angular.

Le questionnaire est noté sur 10 points.

Le questionnaire est accessible en ligne à [cette adresse](#).

Bon courage !

Partie 2

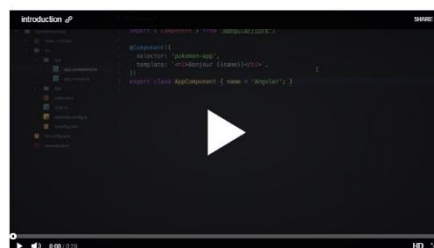
Acquérir les bases d'Angular

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.



CHAPITRE 3 : LES COMPOSANTS

1. Introduction aux composants
2. Qu'est-ce qu'un composant ?
3. Les cycles de vie d'un composant
4. Interagir sur le cycle de vie d'un composant
5. Gérer les interactions de l'utilisateur
6. Exercice
7. Correction
8. Conclusion

FORMATION VIDÉO

Les extraits de code à copier-coller, l'application de démonstration, et la correction de chaque chapitre sont disponibles sur [cette page](#).

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

Je vais vous envoyer un accès à la formation vidéo par email.

Je vous retrouve dans votre boîte de réception,

À tout de suite ! 😊

