Premier pas avec Angular

Un « Hello, World ! » Angular

Par Simon DIENY

Date de publication : 2 septembre 2017

Ce tutoriel va vous permettre de vous familiariser avec la dernière version du framework de Google en commençant par réaliser une superbe application « *Hello, World !* ».

Commentez

I - Premier pas avec Angular 4

Commencer par un « Hello, World ! »

Bonjour et bienvenue dans ce tutoriel sur Angular 4 !

Je vous propose de vous familiariser avec la dernière version du framework de Google en commençant par réaliser une superbe application « *Hello, World !* ». Comme prérequis pour suivre ce tutoriel, vous aurez besoin de :

- connaître le HTML, CSS et un peu de JavaScript ;
- avoir Node et NPM installés sur votre machine (téléchargeable **ici**) ;
- c'est tout !

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 adapté ;
- écrire le composant racine de notre application : rappelez-vous que votre application Angular n'est qu'un assemblage de composants, et donc il faut au moins un composant pour faire une application ;
- dire à Angular avec quel composant nous souhaitons démarrer notre application, c'est-à-dire le composant par défaut. Pour nous ce sera facile, car nous n'aurons qu'un seul composant, le composant racine ;
- écrire 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 traîner et de commencer tout de suite ! Allez, au boulot ! ☺

I-A - Choisir un environnement de développement

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

Il y a plusieurs IDE 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 d'un 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 IDE 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. Retenez simplement que le support de TypeScript est appréciable.

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

Pour ma part j'utilise **Atom** dans ce cours, mais vous êtes libre de travailler avec l'outil que préférez.

On va commencer par créer un dossier vide, qui sera le dossier racine de notre projet. Sachez que ce que nous allons développer maintenant, est à la base de toute les applications Angular.

Créons donc un dossier nommé *hello-world*.

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

I-B - 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 projet avec une seule ligne de commande ! Cependant, je préfère partir d'un dossier vide, pour que vous puissiez voir le processus de création en entier.

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.

I-B-1 - 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.

```
1. {
2.     "name": "ng2-pokemon-app",
3.     "version": "1.0.0",
4.     "description": "An awesome Angular app for managing pokemons !",
5.     "scripts": {
6.         "start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",
7.         "e2e": "tsc && concurrently \"http-server -s\" \"protractor protractor.config.js\" --
kill-others --success first",
8.         "lint": "tslint ./app/**/*.ts -t verbose",
9.         "lite": "lite-server",
10.         "pree2e": "webdriver-manager update",
11.         "test":
12.         "tsc && concurrently \"tsc -w\" \"karma start karma.conf.js\"",
13.         "test-once": "tsc && karma start karma.conf.js --single-run",
14.         "tsc": "tsc",
15.         "tsc:w": "tsc -w"
16.     },
17.     "keywords": [],
18.     "author": "",
19.     "license": "MIT",
20.     "dependencies": {
21.         "@angular/common": "~4.3.4",
22.         "@angular/compiler": "~4.3.4",
23.         "@angular/core": "~4.3.4",
24.         "@angular/forms": "~4.3.4",
25.         "@angular/http": "~4.3.4",
26.         "@angular/platform-browser": "~4.3.4",
27.         "@angular/platform-browser-dynamic": "~4.3.4",
28.         "@angular/router": "~4.3.4",
29.         "angular-in-memory-web-api": "~0.3.0",
30.         "systemjs": "0.19.40",
31.         "core-js": "^2.4.1",
32.         "rxjs": "5.0.1",
33.         "zone.js": "^0.8.4"
34.     },
35.         "devDependencies": {
36.         "concurrently": "^3.2.0",
```

```
37.         "lite-server": "^2.2.2",
38.         "typescript": "~2.1.5",
39.         "canonical-path": "0.0.2",
40.         "http-server": "^0.9.0",
41.         "tslint": "^3.15.1",
42.         "lodash": "^4.16.4",
43.         "jasmine-core": "~2.4.1",
44.         "karma": "^1.3.0",
45.         "karma-chrome-launcher": "^2.0.0",
46.         "karma-cli": "^1.0.1",
47.         "karma-jasmine": "^1.0.2",
48.         "karma-jasmine-html-reporter": "^0.2.2",
49.         "protractor": "~4.0.14",
50.         "rimraf": "^2.5.4",
51.         "@types/node": "^6.0.46",
52.         "@types/jasmine": "2.5.36"
53.     },
54.     "repository": {}
55. }
```

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

- **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 ;
- **les dépendances** : à partir de la ligne 21, apparaît une liste de toutes les dépendances de notre application. Par exemple, *@angular/router* a pour vocation de gérer les routes de notre formulaire et *@angular/forms* les formulaires ;
- **les dépendances relatives au développement** : ces dépendances sont listées à partir de la ligne 36, et concernent les dépendances dont nous n'aurons plus besoin quand notre application sera terminée. Par exemple, Typescript (ligne 39) 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.

I-B-2 - 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 bibliothèques 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 choses :

- le dossier de notre application à la ligne 15, ci-dessous le dossier *app* ;
- le *mapping* de nos bibliothèques : on lie un alias avec l'emplacement de la bibliothèque. 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 :

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

Voici le contenu du fichier *systemjs.config.js* :

```
1. /**
2. * La configuration de SystemJS pour notre application.
3. * On peut modifier ce fichier par la suite selon nos besoins.
4. */
5. (function (global) {
6.     System.config({
```

```
7.          paths: {
8.              // définition d'un raccourci, 'npm' pointera vers 'node_modules'
9.              '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',
19.              '@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/http': 'npm:@angular/http/bundles/http.umd.js',
23.              '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
24.              '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
25.              // autres librairies
26.              'rxjs': 'npm:rxjs',
27.              'angular-in-memory-web-api':'npm:angular-in-memory-web-api/bundles/in-memory-web-
api.umd.js'
28.          },
29.          // L'option 'packages' indique à SystemJS comment charger
30.          // les paquets qui n'ont pas de fichiers et/ou extensions renseignés
31.          packages: {
32.              app: {
33.                  main: './main.js',
34.                  defaultExtension: 'js'
35.              },
36.              rxjs: {
37.                  defaultExtension: 'js'
38.              }
39.          }
40.      });
41. })(this);
```

À la ligne 38, nous indiquons que le fichier pour démarrer notre application est main.js. Comme nous développons en TypeScript, cela signifie que nous devrons développer un fichier main.ts qui sera ensuite transcompilé en main.js, et qui servira de point d'entrée à notre application !

I-B-3 - Le fichier tsconfig.json

Ce document est le fichier de configuration de TypeScript. On peut définir un certain nombre d'éléments de configuration dans ce fichier, en voici quelques-uns :

- à 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 ;
- à 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.

```
1. {
2.     "compilerOptions": {
3.         "target": "es5",
4.         "module": "commonjs",
5.         "moduleResolution": "node",
6.         "sourceMap": true,
7.         "emitDecoratorMetadata": true,
8.         "experimentalDecorators": true,
9.         "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.*

I-B-4 - 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 nommé node_modules à la racine de notre projet. Ce dossier contient toutes les dépendances dont nous avons besoin pour faire fonctionner notre projet.

I-C - Créer notre premier composant

Nous allons créer notre premier composant, enfin !

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. Donc, créer un dossier *app*, et placer à l'intérieur un fichier *app.component.ts*, qui contiendra notre premier composant :

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

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

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

```
import { unElement } from { quelquePart }
```

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 notre 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).

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.

I-D - 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 dédiés à l'authentification, tous ceux qui servent à construire un blog, 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 compris !

Au minimum, votre application doit contenir un module : le **module racine**. Au fur et à mesure que votre application se développera, vous rajouterez 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* :

```
1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { AppComponent } from './app.component';
4.
5. @NgModule({
6.     imports: [ BrowserModule ],
7.     declarations: [ AppComponent ],
8.     bootstrap: [ AppComponent ]
9. })
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 directives *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 monomodule, puis nous ajouterons d'autres modules pour que vous voyiez comment se passe le développement d'une application plus complexe.

I-E - 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* :

```
1. import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2. import { AppModule } from './app.module';
3. platformBrowserDynamic().bootstrapModule(AppModule);
```

Nous 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 hybride avec **NativeScript** ou du développement *cross-plateform* 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*.

Heu ! Pourquoi créer *main.ts*, *app.module.ts* et *app.component.ts* dans trois fichiers différents, tout ça pour lancer une application qui affiche un « Hello, World » ?

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 ; son démarrage est indépendant de la description de notre module, qui est également indépendant des composants qui le constituent. Ces trois éléments doivent donc être séparés !

I-F - 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 termes d'architecture ? Une SPA (*Single Page Application*), c'est-à-dire que notre application ne sera composée que d'une page HTML avec beaucoup de code JavaScript pour dynamiser la page, récupérer des informations d'un serveur distant, etc. Eh bien, il nous manque cette fameuse page HTML. Créons-la tout de suite avec un fichier *index.html*, à la racine du projet **(et non dans le dossier app !)** :

```
1. <!DOCTYPE html>
2. <html>
3.   <head>
4.     <title>Angular QuickStart</title>
5.     <meta charset="UTF-8">
6.     <meta name="viewport" content="width=device-width, initial-scale=1">
7.     <!-- 1. Chargement des librairies -->
8.     <!-- Polyfill(s) pour les anciens navigateurs -->
9.     <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('app').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 22, 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 :

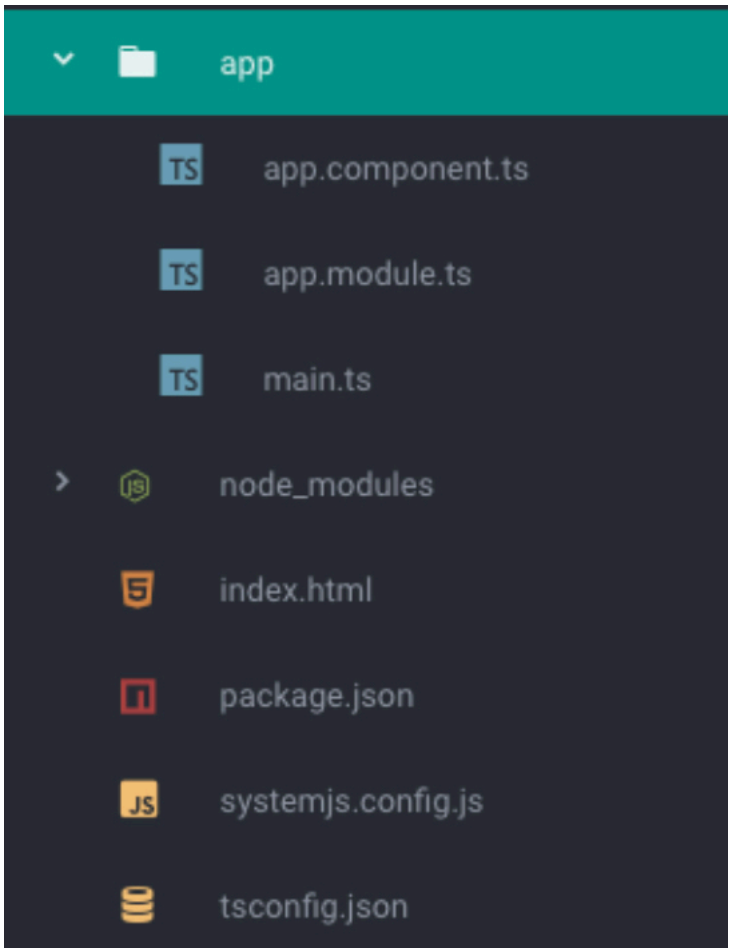

Figure 1 - L'architecture de notre projet

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

I-G - Démarrez l'application

Démarrer l'application va être un jeu d'enfant. Il y a déjà une commande disponible pour nous permettre cela. Ouvrez donc un terminal qui pointe vers le dossier de votre projet et tapez 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, Angular 4 ! » dans votre navigateur !

Ça y est, nous y sommes arrivés !

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

Si toutefois vous avez des erreurs et que l'application ne se lance pas, affichez la console dans laquelle vous avez tapé la commande npm start, et scrollez 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, essayez 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 frameworks : installer le **socle** de notre application. Vous avez fait plus qu'afficher un message à vos utilisateurs, vous venez de mettre en place la base 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 :

- elle compile tous nos fichiers TypeScript vers du JavaScript ES5 ;
- elle lance l'application dans un navigateur et la met automatiquement à jour si nous modifions notre code !

Dans les bibliothèques 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 !

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

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.     selector: 'pokemon-app',
5.     template: '<h1>Bonjour, Angular 4 !</h1>'
6. })
7. export class AppComponent {}
```

Si maintenant vous retournez dans le navigateur ou tourne votre application, vous verrez alors le message « *Bonjour, Angular 4 !* » s'afficher, à la place de « *Hello, Angular 4 !* », le tout sans avoir à appuyer sur F5 !

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

I-H - Bonus : nettoyer son dossier de développement

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

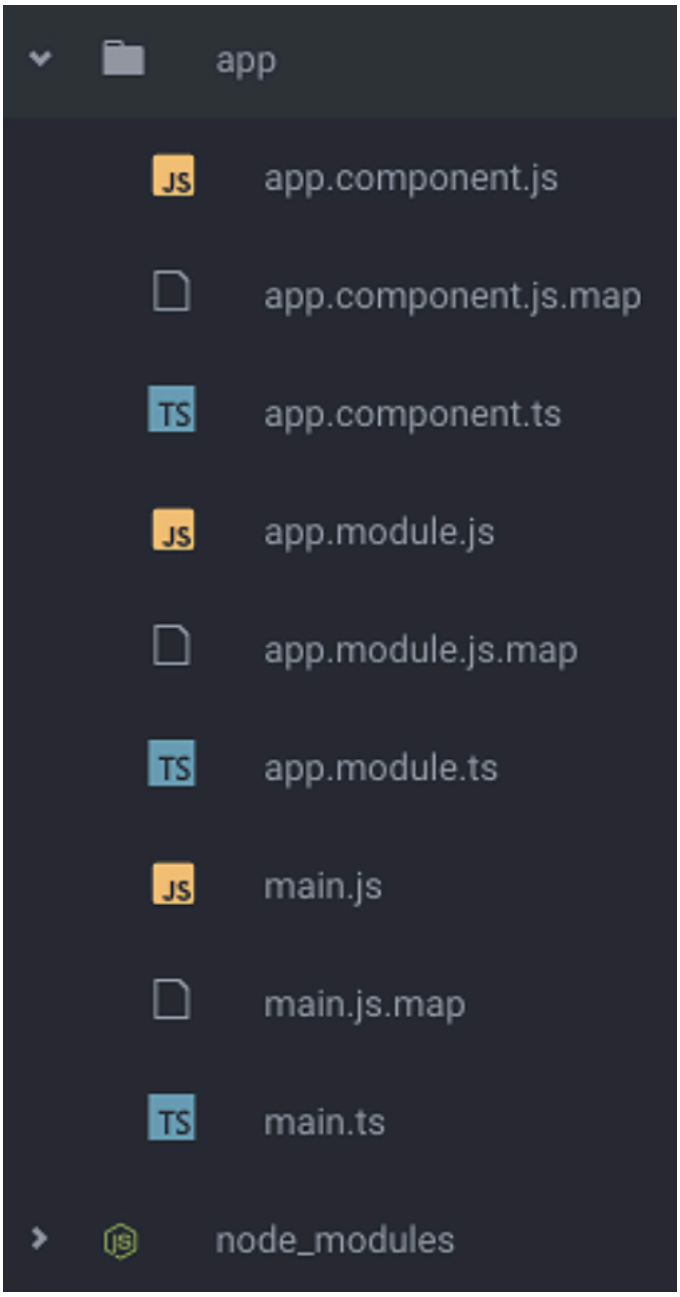

Figure 2 - Le dossier 'app' n'est très lisible

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) resterons 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, créons un dossier *dist* à la racine de notre 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 :

```
1. {
2.     "compilerOptions": {
3.         "target": "es5",
4.         "module": "commonjs",
5.         "moduleResolution": "node",
6.         "sourceMap": true,
7.         "emitDecoratorMetadata": true,
8.         "experimentalDecorators": true,
9.         "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 *system.config.js* et modifiez la ligne 5 ci-dessous :

```
1. (function (global) {
2.     System.config({
3.         // ...
4.         map: {
5.             app: 'dist', // <-- remplacez 'app' par 'dist' comme ceci !
```

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

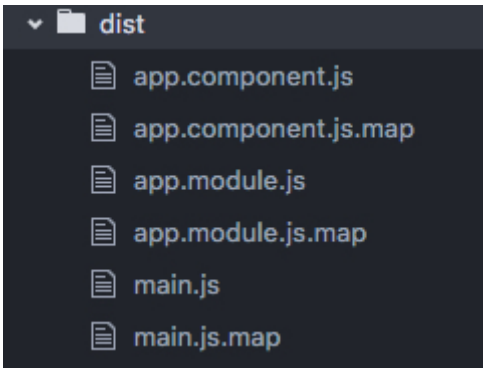

Figure 3 - Le dossier 'dist' contient de nouveaux fichiers

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 fini 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 minimum du .gitignore pour une application Angular :*

```
/dist
/node_modules
```

I-I - Conclusion

Voilà, vous l'avez fait, 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'est pas inutile de s'y intéresser.

I-I-1 - 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 *package.json* initial est fourni avec des commandes prêtes à l'emploi comme la commande *npm start*, qui nous permet de démarrer notre application sans trop d'efforts.

II - Note de la rédaction de Developpez.com

Nous tenons à remercier **Winjerome** pour la mise au gabarit et **Jacques_Jean** pour la relecture orthographique.