À la découverte d'Angular 4

Cet article n'a pas pour vocation d'être un tutoriel mais plutôt une présentation d'Angular 4 et des possibilités que nous offre ce framework.

Ne vous en faites pas, les notions que je vous montrerais tout au long de cette article seront facilement utilisables et modifiables pour que vous puissiez facilement les intégrer à votre application web.

Table des matières

Table des matières

Les prérequis

Avant de se lancer dans la découverte du framework Angular, nous avons besoin d'avoir certaines notions pour être capable de saisir toutes les subtilités et la syntaxe des exemples de code que vous pourrez voir dans cet article.

Il vous faudra connaître :

-Le HTML et le CSS

-Le javascript, c'est le langage que nous allons utiliser ici

-Et bien entendu, connaître la POO ! (Programmation Orientée Objet pour les intimes).

Angular, mais qu'est-ce que c'est ?

Angular est un framework Javascript. Il est l'évolution d'AngularJs, développé par google.

(A partir de maintenant, quand je parlerai d'Angular, ce sera pour désigner les versions postérieures à AngularJs, de Angular2 jusqu'à Angular4).

Un framework, c'est une sorte de boite à outils ! Quand vous utilisez des frameworks, vous utilisez également les fonctions mises à dispositions pour vous simplifier la vie !

Mais à quoi ça sert Angular ?
Question légitime, c'est vrai qu'avant aujourd'hui, on se débrouillait très bien sans. Angular va nous permettre de nous mettre à développer des **applications web** et non-plus des **sites web.**
La différence ? Bon en bref... :

- Le site web : 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.

- L'application web : 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 chattant avec ses amis par exemple ! (Comme la version web de Facebook par exemple).

Bon, maintenant on qu'on a expliqué le contexte dans lequel on utilise Angular, rentrons dans le vif du sujet ! Comment ça fonctionne ?

Angular est un framework orienté composant.
C'est à dire que nous allons développer tout un panel de composants qui une fois mis bout à bout forment une application.
Un composant c'est la combinaison du HTML et d'une classe javascript (nous verrons cela plus tard dans le cours), et chaque composant de l'application fonctionne de façon autonome.

Passons maintenant à la présentation du fonctionnement de notre application Angular.

Il y a quatre éléments que je dois vous présenter, ce sont les éléments principaux de notre application :

- -Le Module
- -Le Composant
- -Le Template
- -Les annotations

Le Module :

Un module, c'est un morceau de votre application.
Une application est constituée de plusieurs modules qui se complètement.
Petit conseil de propreté du code : Chaque module a une utilité qui lui est propre, on ne fait pas de mélange !

Par exemple le module @angular/core est un module d'Angular qui contient beaucoup d'éléments de base.

Par exemple, si l'on souhaite créer un composant il faudra importer la classe Composant.

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

Cela dit, il est possible d'importer des éléments de notre propre code.
Le chemin d'accès à renseigner sera donc un chemin relatif :

```
import { UneClassePersonnelle } from './unDossierPersonnel'
```

Le Composant

Les composants sont la pierre angulaire d'Angular !
Ils vous serviront absolument à tout ! Interagir avec l'utilisateur, gérer des formulaires et j'en passe !

N'oubliez pas, un composant est voué à effectuer une action bien précise ; Si vous voulez faire deux actions, utilisez deux composants.

Un composant est relié à une classez dans laquelle des actions et méthodes ont été définies.
Les composants vont agir sur la « vue » (la vue, c'est ce que l'utilisateur voit), ils vont permettre la modification de la vue (totale ou partielle).
Par exemple, un composant peut être chargé d'afficher un logo de chargement le temps que la page charge, de sorte à ce que l'expérience utilisateur soit meilleure et qu'il n'ait pas l'impression d'attendre bêtement devant une page blanche.

Bon, c'est bien beau tout ça, mais ça ressemble à quoi un composant ?

Bonne question, voici un petit exemple !

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

@Component({
    selector: 'mon-composant',
    template: `
    <div>
      <h1>Titre 1</h1>
      <p>Un petite paragraphe</p>
    </div>`
})
export class MyComponent {}
```

Voici un composant somme toute basique.

« Mais qu'est-ce que c'est que ce « @Component » ? »

Patience ! Nous allons dégrossir tout ceci dans quelques instants !

Le template

Un template qu'est-ce que c'est ? Je suis certain que vous avez déjà entendu ce terme si vous vous intéressez un tant soit peu à la réalisation de site web.
Un template, pour faire simple, c'est un thème, l'enveloppe graphique de votre site.

Dans notre cas, un template, c'est simplement une « vue » qui est associé à un de nos composants. Et ce sont les templates qui dictent ce que les composants devront afficher.

Quelques exemples de template :

```
<h1>Un template</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit..</p>
```

Et oui, ce n'est que du html !
Mais c'est bel et bien un template, croyez-moi !

Attardons rapidement sur ce code :
Cette syntaxe indique à Angular que l'on souhaite incorporer à notre template une

variable qui se trouve dans le composant qui gère le template.
Attention, cette variable n'est pas accessible dans notre template, elle provient seulement du composant !

Allez, voici un template un peu plus original ….

```
<h1>Un template</h1>
<p>Âge : {{ ageutilisateur}}</p>
```

Les annotations

Vous souvenez vous du composant que nous avons vu plus haut ?
Et bien ici nous allons expliquer ce qu'il contient !

Angular se sert de composants pour pouvoir fonctionner, mais comment faire pour reconnaître un composant ?

Rien de plus simple !

Il faut juste utiliser le décorateur « @Component » et remplir ses diverses propriétés :

```
@Component({
    selector: 'mon-composant',
    template: `
    <div>
      <h1>Titre 1</h1>
      <p>Un petite paragraphe</p>
    </div>`
})
```

Ici, notre composant a deux propriétés, un selector et un template.

Le « selector », c'est tout simplement ce qui va nous permettre d'appeler notre composant (qui comporte ici un template), dans notre page html ou nos templates.

```
<body>
    <mon-composant></mon-composant>
</body>
```

Voyez ? Rien de plus simple ! On dirait tout simplement une balise html classique ! Mais qu'en est-il du template associé au composant ? Je suis certain que vous savez à quoi il va servir … Vous ne voyez pas ?
Et bien la réponse est très simple, à l'intérieur de notre « selector » « <mon-composant></mon-composant> », nous retrouverons le code html du template que nous avons créé.
Il nous reste cependant un point à éclaircir concernant notre composant, vous vous souvenez de l'« export » que nous avons vu à la dernière ligne ?

```
export class MyComponent {}
```

Et bien cet « export », sert simplement à rendre accessible notre composant par les autres composants de notre application afin que ces derniers puissent l'importer comme nous avons importé le composant « Component » depuis '@angular/core'.

Voici un petit conseil que je peux vous donner.
Lorsque vous codez un composant, suffixez le nom du fichier de votre composant par « component », de cette façon, vous saurez qu'est-ce qui est un composant et qu'est-ce qui ne l'est pas dans votre application.

TypeScript

Allez, encore un peu de théorie et promis on s'y met ! Il faut bien que vous soyez conscient du contexte dans lequel vous aller travailler !

Vous avez pu voir que l'extension du fichier comportant notre composant est un « .ts », et bien c'est l'extension des fichiers dans lesquels on code en TypeScript.

De sorte à ce que ce soit plus claire, voici la définition de TypeScript que nous propose Wikipedia :

« *TypeScript est un langage de programmation libre et open-source développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript. C'est*

un sur-ensemble de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript. Il a été cocréé par Anders Hejlsberg, principal inventeur de C#)»

Allez, une petite démo s'impose, on va coder un « Hello World » en TypeScript, rien ne vaut un bel exemple !
Créez un fichier HellowWolrd.ts et collez-y le code suivant ;

```
function HelloWolrd()
{
  return "Hello World";
}
  document.body.innerHTML = HelloWolrd();
```

On dirait un du javascript, mais non, c'est bien du TypeScript !
On compilera le Typecript en javascript pour que notre navigateur puisse l'interpréter (sans quoi, le navigateur ne peut interpréter du TypeScript).

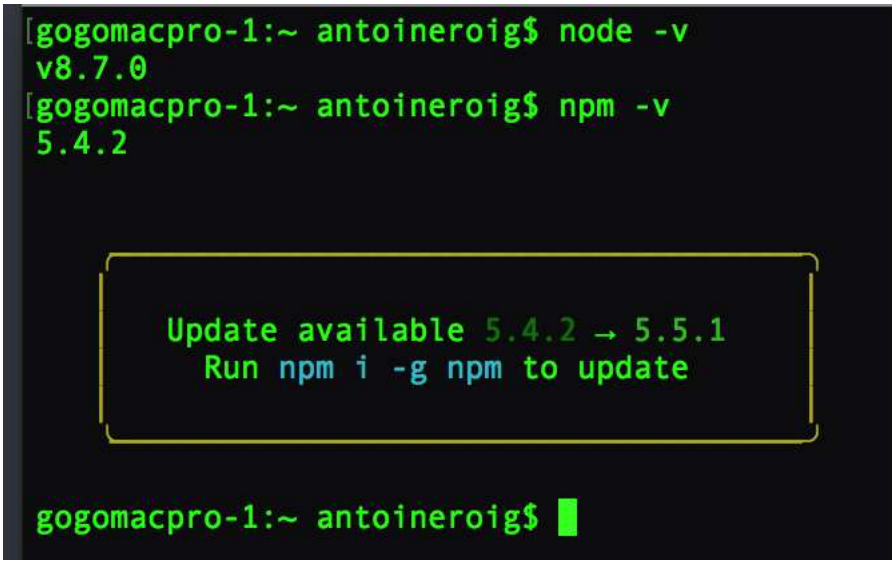
Installation

C'est parti ! On va commencer à faire marcher notre TypeScript !

Il va falloir installer quelques petites choses avant.

Dans un premier temps, veillez à ce que Node.js et npm (le gestionnaire de packets pour javascript) soient installés.

Ouvrez donc un terminal et tapez ces commandes :



Il vous faut au moins la version 5.x.x de node et 3.x.x de npm.

Si vous ne les avez pas, téléchargez-les.

Maintenant il faut installer le compilateur TypeScript (qui je rappelle, compilera notre typescript en javascript) :

```
gogomacpro-1:~ antoineroig$ npm install -g typescript
```

Et maintenant on compile notre fichier test.ts.

```
gogomacpro-1:~ antoineroig$ tsc test.ts
```

Et voilà ! Vous avez compilé votre typescript en javascript (notez l'apparition du fichier test.js).
Rien n'a changé, c'est le même code, mais avec une extension différente. Normal, nous avions écrit du javascript dans notre fichier.
Mais n'oubliez pas, notre navigateur n'interprétera que des fichiers javasript, n'oubliez pas de compiler !

TypeScript et sa syntaxe

Maintenant nous allons voir à quoi ressemble un code en typescript.

Les types

Voici la syntaxe pour déclarer une variable en typescript :

```
UnNombre : number = 100;
```

Vient d'abord le nombre de la variable, les deux points puis **le type de la variable**.
Ensuite vous pouvez affecter une valeur à votre variable comme dans l'exemple ci-dessus.

Il en va de même pour les fonctions !

```
retournerUnMot(mot: string) : string
{
  return mot;
}
```

Voyez, nous avons dans un premier temps le nom de notre fonction, son paramètre (on précise également le type du paramètre) et ensuite le type de retour de la fonction qui ici est un string.

Enfin, on commence à voir à quoi ressemble un code en typescript.

Maintenant, nous allons créer un petit projet angular dans lequel nous allons pouvoir coder et surtout, voir le résultat de notre travail sur une page web.

Un projet Angular

Ouvrez donc votre éditeur de texte préféré et créez un dossier nommé « Mon-app », c'est dans celui-ci que nous allons développer.

Nous allons ajouter à la racine de notre projet des fichiers de configuration qui seront nécessaires pour que Angular fonctionne.

Créez un fichier package.json et ajoutez-y le code suivant

```
{
  "name": "ng2-pokemon-app",
  "version": "1.0.0",
  "description": "A awesome app for managing pokemons, to train at Angular",
  "scripts": {
    "start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",
    "e2e": "tsc && concurrently \"http-server -s\" \"protractor protractor.config.js\" --kill-others --success first",
    "lint": "tslint ./app/**/*.ts -t verbose",
    "lite": "lite-server",
    "pree2e": "webdriver-manager update",
    "test": "tsc && concurrently \"tsc -w\" \"karma start karma.conf.js\"",
    "test-once": "tsc && karma start karma.conf.js --single-run",
    "tsc": "tsc",
    "tsc:w": "tsc -w"
  },
  "keywords": [],
  "author": "",
  "license": "MIT",
  "dependencies": {
    "@angular/common": "~4.0.0",
    "@angular/compiler": "~4.0.0",
    "@angular/core": "~4.0.0",
    "@angular/forms": "~4.0.0",
    "@angular/http": "~4.0.0",
    "@angular/platform-browser": "~4.0.0",
    "@angular/platform-browser-dynamic": "~4.0.0",
    "@angular/router": "~4.0.0",
    "angular-in-memory-web-api": "~0.3.0",
    "systemjs": "0.19.40",
    "core-js": "^2.4.1",
    "rxjs": "5.0.1",
    "zone.js": "^0.8.4"
  },
  "devDependencies": {
    "concurrently": "^3.2.0",
    "lite-server": "^2.2.2",
    "typescript": "~2.1.5",
    "canonical-path": "0.0.2",
    "http-server": "^0.9.0",
    "tslint": "^3.15.1",
    "lodash": "^4.16.4",
    "jasmine-core": "~2.4.1",
    "karma": "^1.3.0",
    "karma-chrome-launcher": "^2.0.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~4.0.14",
    "rimraf": "^2.5.4",
    "@types/node": "^6.0.46",
    "@types/jasmine": "2.5.36"
  },
  "repository": {}
}
```

Ce fichier contient les scripts qui sont déjà disponibles dans notre application, par exemple :

- start permet le démarrage de l'application
- lite permet de démarrer un serveur sur lequel Angular tournera.

Ensuite il y a les dépendances de notre application et leurs versions

Et enfin, les dépendances de développement, ce sont les dépendances que nous n'utiliserons plus une fois l'application mise en production.

Créez ensuite un fichier systemjs.config.js contenant ce code:

```
/**
 * La configuration de SystemJS pour notre application.
 * On peut modifier ce fichier par la suite selon nos besoins.
 */
(function (global) {
  System.config({
    paths: {
      // définition d'un raccourcis, 'npm' pointera vers 'node_modules'
      'npm:': 'node_modules/'
    },
    // L'option map permet d'indiquer à SystemJS l'emplacement des éléments charger les éléments
    map: {
      // notre application se trouve dans le dossier 'dist'
      app: 'dist',

      // packets angular
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',
      '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
      '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',

      // autres librairies
      'rxjs':                      'npm:rxjs',
      'angular-in-memory-web-api': 'npm:angular-in-memory-web-api/bundles/in-memory-web-api.umd.js'
    },
    // L'option 'packages' indique à SystemJS comment charger les paquets qui n'ont pas de fichiers et/ou extensions renseig
    packages: {
      app: {
        main: './main.js',
        defaultExtension: 'js'
      },
      rxjs: {
        defaultExtension: 'js'
      }
    }
  });
})(this);
```

System.js est la libraire qu'Angular utilise par défaut, ici elle va permettre de charger les modules JavaScript dont nous avons besoin.

Enfin, créez le fichier tsconfig.json qui est tout simplement le fichier de configure de TypeScript.

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [ "es2015", "dom" ],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true,
    "removeComments": false,
    "outDir": "dist"

  }
}
```

Et pour finir l'installation, on va installer les dépendances déclarées dans le package.json de notre application.
Entrez donc cette commande à la racine de votre application :

```
gogomacpro-1:app antoineroig$ npm install
```

Vous observerez l'apparition du dossier node_modules qui contiendra toutes les dépendances qui serviront à faire fonctionner notre application.

Hello World

C'est parti, nous allons créer un composant.

Tout d'abord, créez un dossier app, c'est dans celui qui que nous coderons nos composants !

Ceci fait, créez le fichier app.component.ts et ajoutez-y le code suivant :

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

@Component({
  selector: 'un-composant',
  template: `<h1>Hello, Angular  !</h1>`,
})
export class AppComponent { }
```

Je pense que si vous avez suivi les parties précédentes, rien ne vous échappe ici !

Maintenant nous allons créer notre module.

Créez alors un fichier app.module.ts, toujours dans le dossier app.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Qu'y a-t-il de nouveau ici ?
C'est l'annotation @NgModule, c'est grâce à cette annotation que l'on créé un module.

Dans ce module, on importe BrowserModule qui contient des éléments essentiels au fonctionnement de notre application.

Viennent ensuite les déclarations. Il s'agit des composants qui appartienent à ce module. Ici, nous avons donc ajouté le composant AppComponent que nous avons créé plus tôt.

Et enfin, la partie bootstrap, c'est tout bonnement la définition du composant racine de notre application, c'est à dire le composant qui sera appelé au démarrage de notre application.

Il est maintenant temps de créer un « point d'entrée » à notre application.
Dans notre fichier de configuration de systemjs il était question d'un main.js.
Créons donc un main.ts (qui sera compilé en main.js par la suite).

Ajoutez-y le code suivant :

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Et voilà, ce fichier lancera l'application (vous retrouvez ici notre « AppModule » qui sera donc appelé au démarrage.

Il nous manque une seule page et on y va !
Créez donc une page html, d'ailleurs je vous la donne (elle devra se trouver à la racine de votre application dans le dossier Mon-app).

```
<!DOCTYPE html>
<html>
  <head>
    <title>Angular QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- 1. Chargement des librairies -->
    <!-- Polyfill(s) pour les anciens navigateurs -->
    <script src="node_modules/core-js/client/shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <!-- 2. Configuration de SystemJS -->
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>
  <!-- 3. Afficher l'application -->
  <body>
    <un-composant></un-composant>
  </body>
```

Ceci est une simple page html.
Voyez la balise <un-composant></un-composant>, c'est ici que notre application se trouvera !

Démarrage de l'application

Enfin ! On va pouvoir voir le fruit de notre travail !

Ruez-vous sur votre terminal à la racine de votre application (Le dossier « Mon-app) et lancez cette commande :

```
gogomacpro-1:Mon-app antoineroig$ npm start
```

Une page web va s'ouvrir dans votre navigateur et là, que voyez-vous....

Hello, Angular !

Bravo ! Votre application s'est lancée !

Essayez de modifier votre template, vous verrez qu'il n'y pas besoin de relancer la commande npm start, les changements sont visibles automatiquement !

Exemples de possibilités avec Angular

A partir de maintenant, je vais vous montrer plusieurs exemples de codes que vous pourrez intégrer à votre code facilement (mais en réfléchissant un peu tout de même !). Je vous rappelle que cet article a pour but de vous présenter ce que propose le framework Angular et non un tutoriel de projet. Il faudra vous creuser la tête pour créer quelque chose d'original par la suite !

Par exemple, nous allons voir comment déclencher un événement quand on clique sur un élément de notre page.

Créons une classe Personnage dans un fichier personnage.ts dans notre dossier app.

```
export class Personnage
{
  age : number;
  nom : string;
  prenom : string;
}
```

On retourne dans notre fichier app.component.ts pour y effectuer des ajouts

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

Ici on importe notre classe personnage et on ajoute également le module OnInit qui va permettre l'initialisation des propriétés de notre composant.

On ajoute ensuite un peu de code dans notre composant.

```
export class AppComponent
{
 perso : Personnage;

 ngOnInit()
 {
    this.perso.age = 10;
    this.perso.prenom = "John";
    this.perso.nom = "Doe";
 }

 selectPerso(personnage:Personnage)
 {
   console.log('Le personnag selectionné se nomme' + personnage.prenom + ' ' +
   personnage.nom + ' et a ' + personnage.age + ' ans.' );
 }
}
```

Ici, on va créer un personnage et remplir ses attributs avec des valeurs.
On définit ensuite un méthode selectPerso qui affichera dans la console les informations d'un personnage prit en paramètre de cette même méthode.

Maintenant modifions notre template.

Par soucis de propreté, nous allons créer un fichier personnage.template.html, nous coderons notre template dans ce fichier et nous l'appellerons dans notre composant avec cette syntaxe :

```
templateUrl: `app/personnage.template.html`
```

On rajoute également un lien pour bénéficier d'effets graphiques dans notre index :

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.97.8/css/materialize.min.css">
```

Ici, on ajoute la librairie Materialize.

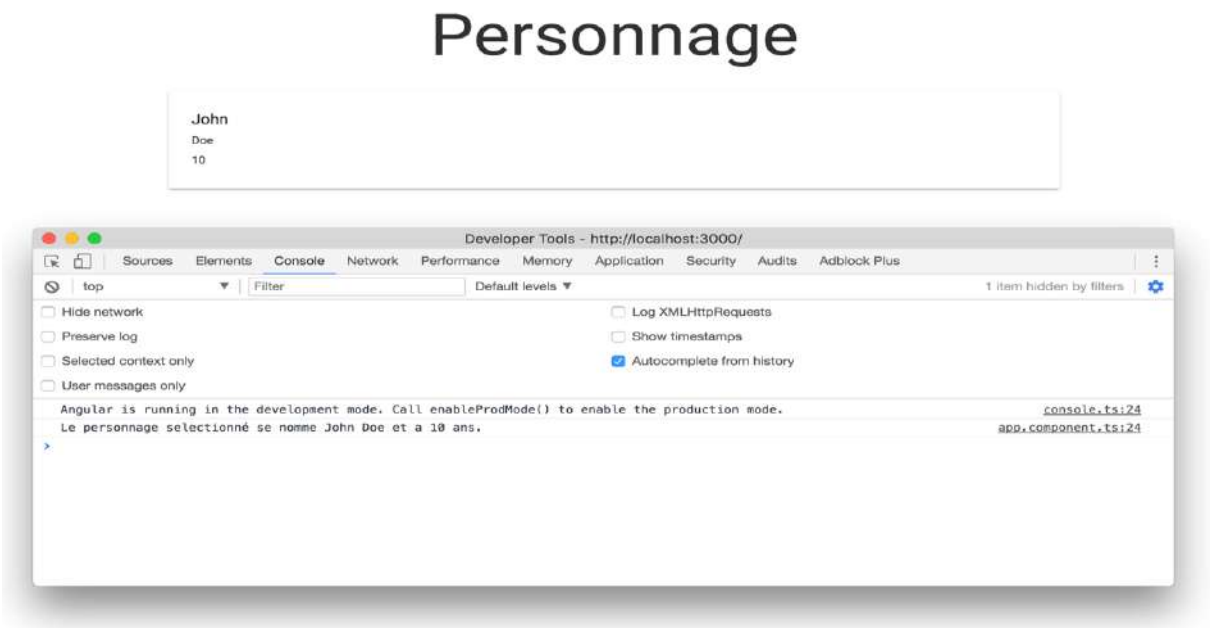
Une va ensuite créer notre template dans ce fichier.

```
<h1 class='center'>Personnage</h1>
<div class='container'>
<div class="row">
  <div class="card horizontal waves-effect" (click)="selectPerso(perso)">
    <div class="card-stacked">
      <div class="card-content">
        <p>{{ perso.prenom }}</p>
        <p><small>{{ perso.nom }}</small></p>
        <p><small>{{ perso.age }}</small></p>
      </div>
    </div>
  </div>
</div>
</div>
```

C'est déjà un peu plus fourni !
Ne paniquez pas, je vous explique.

Nous avons une div qui comporte la propriété « click » qui va donc appeler la méthode selectPerso définit plus haut en lui envoyant « perso », c'est le personnage que l'on retrouve dessous. Magique non ? Angular sait que nous utilisons la variable perso qui se trouve dans notre Composant et n'a aucun souci à le récupérer pour appliquer une méthode qui utilise ce personnage.

Allez-y, cliquez sur la carte de votre personnage et regardez dans la console.



`

Félicitations ! Vous avez développé votre première action !

Alors, ça vous a plu ? Ne vous en faites pas vous n'avez pas fini de vous amuser.

La prochaine particularité d'Angular que nous allons étudier sont les « Directives ».

Une directive c'est une classe d'Angular, cependant elle n'a pas de template.

Elle permet d'interagir avec les éléments html d'une page en leur attribuant un comportement spécifique.

Utilisons une directive déjà existante, la directif ngIf.

La directive ngIf permet d'ajouter des conditions dans notre code HTML, voyons ici un exemple qui va vous permettre de travailler un petit peu.

```
<h1 class='center'>Personnage</h1>
<div class='container'>
<div class="row">
  <div class="card horizontal waves-effect" (click)="selectPerso(perso)">
    <div class="card-stacked">
      <div class="card-content">
        <p>{{ perso.prenom }}</p>
        <p><small>{{ perso.nom }}</small></p>
        <p *ngIf="perso.age > 18" ><small>{{ perso.age }} </small></p>
        <p *ngIf="perso.age < 18">Ce personnage est mineur, il ne peut pas apparaitre à l'écran.</p>
      </div>
    </div>
  </div>
</div>
</div>
```

Ajoutez les deux lignes qui comportent *ngIf et regardez votre page web...
Mince, on ne voit plus l'âge de John !
À vous de modifier votre code pour afficher son âge.

Ça y est c'est fait ? Je vous l'accorde, changer une variable à la main ce n'est pas très drôle, et si on ajoutait un peu de code ?

Ajoutez donc ce bouton dans votre template :

```
<p *ngIf="perso.age >= 18" ><small>{{ perso.age }} </small></p>
<p *ngIf="perso.age < 18">Ce personnage est mineur, il ne peut pas apparaitre à l'écran.</p>
<button type="button" class="btn btn-default" (click)="ChangerAge()">Changer âge</button>
```

Puis ajoutez cette fonction dans votre composant.

```
ChangerAge()
{
  this.perso.age = 19;
}
```

Allez-y, cliquez sur le bouton ? Et bravo ! Nous voyons de nouveau l'âge de notre personnage.

Et si nous créions notre propre directive ?

Créez un fichier « MaDirective.directive.ts »

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

@Directive({selector: '[shadow-card]'})

export class MaDirective {
  constructor(private el: ElementRef, private renderer: Renderer)
  {
    this.setBorder('#f5f5f5');
    this.setHeight('180px');
  }

  @HostListener('mouseenter') onMouseEnter() {
    this.setBorder('#009688');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.setBorder('#f5f5f5');
  }

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

  private setHeight(height: string)
  {
    this.renderer.setElementStyle(this.el.nativeElement, 'height', height);
  }
}
```

Que fait ce code ? Il aura pour but de modifier l'apparence des bordures de la carte qui contient les informations de notre personnage à l'entrée et la sortie de la souris dans notre carte.

Notez qu'à la construction de la directive, nous créons les variables « el » et « renderer » que nous allons définir puis modifier dans la classe « MaDirective »

Dans un premier temps, nous définissons la largeur et la couleur de nos bordures.

Puis, grâce au décorateur @HostListener, nous appliquons des changements sur nos bordures.

Plus qu'une étape et notre directive sera opérationnelle.

Rendez-vous dans le fichier app.module.ts et faite l'import correspondant.

Pour ceux qui n'auraient pas trouvé, il suffit de faire ceci :

Enfin, appliquez le selecteur de notre directive sur la div comportant la carte de notre personnage :

Et le tour est joué !

Passez donc votre souris sur la carte et admirez le changement !

Conclusion

J'espère que cet article aura pu démystifier ce qu'est Angular et que vous continuerez votre apprentissage et le perfectionnement de ce merveilleux framework qui propose une autre approche du développement web.

Maintenant, vous savez ce qu'est une application web et comment elle est composée. Elle s'articule à l'aide de divers composant qui sont en somme les briques qui la composent.

Prenez conscience qu'Angular permet également de gérer des connexions à des serveurs grâce à sa palette de composant. Que ce soit par des systèmes d'authentification ou la récupération d'éléments qui sont stockés dans des bases de données.

Cet article s'appuie sur le cours sur Angular 4 qui est disponible sur openclassroom à l'adresse suivante : https://openclassrooms.com/courses/3647511?status=waiting-for-publication

Cette version du framework est toute récente et pourra sans doute évoluer au fil du temps.