Notes sur le module : *Angular.*

Armel Pitelet

1er février 2022

Table des matières

Lien de la doc Angular https://angular.io/docs.
Lien de la doc Mozilla https://developer.mozilla.org/en-US/ (html, css, javascript, typescript).
Lien de la doc TypeScript : https://www.typescriptlang.org/docs/.
Pour un tuto complet sur angular : https://angular.io/tutorial.

1 Bases

Définition : **Angular**.
framework (cadre de travail) permettant de faire des applications bureau mais surtout web. Historiquement c'était Angularjs et maintenant Angular 13. Maintenue par google, il est réttocompatible et sécurisé. C'est un framework principalement front-end.

Remarque : *Sur les commandes angular.*
Ces commandes commencent par `ng` (dans un terminal).

Remarque : *Any desk.*
Un genre de team viewer (mais en ligne).

Remarque : *Pour créer un projet.*
`ng new NOM_PROJET`. puis (n : pas dans la dernière version), y (routing) et sélectionner un language de style. Va créer un projet dans le dossier nommer par NOM_PROJET.

Remarque : *Sur html et css.*
html sert à faire du formatage de page et css (fiche de style en cascade) à faire du formatage de style. Angular utilise les deux pour faire des pages web.

Définition : **SCSS**.
CSS avec une écriture moins lourde que css. Attention dans un navigateur on ne peut voir que

du css. SCSS doit être compiler vers CSS avec Angular pour être interprété par un naviguateur.

Note :
Pour l'édition de code il faut ensuite passer par Visual Studio Code ou bien PhpStorm (payant mais un mois de licence gratuite...)

Remarque : *Pour excécuter des scripts (angular mais pas que).*
`set-executionpolicy unrestricted` dasn un powershell en mode administrateur (2 conditions obligatoires). Nécessaire pour travailler avec Angular.

Note :
Les projets Angular sont super Fat en terme de place sur le disque.

Note :
Il faut ouvrir le dossier de projet entier (dans VSCode) pour que le serveur puisse fonctionner par la suite.

Remarque : *Sur l'arborescence d'un projet.*

- `tsconfig.json` : config pour la compilation du Typescript vers du javascript (javascript est par contre un Typescript valide). Ce fichier est nécessairepour faire du typescript (et ce n'est pas propre à Angular).
- `package.json` : fichier de dépendance du projet. Si l'on utilise ce fichier la première chose à faire est un `npm install` : pour installer les dépendances d'un projet (va charger le package json dans le nodemodule).
- Dans le dossier `app` : `app.component.xxx` permet de partitionner l'application en component. Un compenent à un rôle précis → il gère un seule et unique action. Chaque componenet doit être l eplus indépendant possible. Exemple : le html gère le rendu d'affichage (la vue), le css les style, spec.ts sert aux test et le ts est le *controlleur* en lui même qui gère la logique. Chaque nouveau component aura 4 fichier mon_component_component associés.
- Dans le dossier `app` : `app.module.ts` permet d'indiquer les librairies nécessaires à l'application (unqiuement ceux inclues dans l'application et pas tt les modules du node module).
- Dans le dossier `app` : `app-routing.modules.ts` : pour gérer les routes associées aux différents components. Ce fichier lie chaque component à un nom.
- Dans le dossier `index.html` : est la page principale du projet (dans laquel sera importer le projet après compilation).
- Dans le dossier `styles.scss` : fichier scss mère qui sera utilisé partout ailleurs dans le projet. Par contre le scss des components est généré en premier (ce qui est important car l'ordre de redéfinition des classe est important.).

Remarque : *Sur la diff JavaScript Typescript.*
TypeScript est typé alors que JavaScript non.

Remarque : *En css.*
L'ordre des importation est très important. On peut modifier un import précédent avec un import suivant.

Définition : **Bootstrap**.
Une bibliothèque CSS qui fournit un ensemble d'objets.

Note :
Pour récupérer des données en Angular il faut forcément passer par une API, il n'est pas possible d'interroger directement une base de donnée.

Remarque : *Sur la compilation.*
Une fois la compilation et le serveur lancé (`ng serve –open`) il n'est pas nécessaire de l'arréter. En JavaScript le code est rechargé en permanence.

Remarque : *Sur l'objet en Angular.*
En Angular (et typescript) tt est objet. On en revient donc au un fichier, une classe. Au niveau de la syntax d'une variable nom : type = valeur.

Mot clef : **let**.
permet de déclarer une varibale locale modifiable (javascript). Le typage n'est pas obligatoire mais est en théorie obligatoire.

Mot clef : **const**.
permet de déclarer une varibale locale non-modifiable (javascript). Le typage n'est pas obligatoire mais est en théorie obligatoire.

Mot clef : **any**.
à la place d'un type, permet de déclarer n'importe quel type. A oublier le plus vite possible (car l'intrêt du typescript est de typé au maximum le code.)

Note :
A partir du moment ou un attribut ou une fonction (dans le component.ts) est public (visibilité par défaut) il est possible de l'utiliser dans la partie html du component.

Mot clef : **export**.
devant la déclaration d'une classe. permet de dire que la classe va pouvoir etre utilisée en dehors du fichier de déclaration. Sans `export` le componet ne peut être utilisé nulle part.

Remarque : *ng-template.*
est un template qui est masqué sur une page tant qu'il n'a pas été appelé. Peut servir avec un *ngIf. Voir Cours et exemple

Mot clef : **Doc html/css**.
regarder ce site en cas de questions sur les languages balises à la con.

Remarque : *Pour créer un nouveau component.*
ng generate component component_name (ou ng g c component_name si on passe par les alias).

Mot clef : **implements**.
permet d'implémenter une interface.

Mot clef : **extends**.
permet de déclarer un héritage.

Remarque : *Sur l'interface OnInit.*
fait partie du cycle de vie des objets de type component. OnInit indique un traitement appliqués

après l'initialisation d'un objet (constructeur, injection de dépendance). Le OnInit est fait avant que la vue soit générer. En générale on fait dans le OnInit les requêtes à l'api, gérer les paramètres de root.

Sur le cycle de vie des objets angular.

Remarque : *Sur OnDestroy.*
Est appelé lorsque la fenêtre associée au component disparait.

Remarque : *pour installer une dépendance/librairie.*
`npm install un_truc` ou `npm i un_truc` si l'on passe par les alias. Met à jour le package.json.

Remarque : *Sur le SCSS.*
On peut définir des variables en SCSS en préfixant le nom d'un d'un dollar : $variable . & rappel l'élément courant (parent plutôt, un peu l'équivalent du this). voir https://www.code-boost.com/beginners-guide-to-scss/#nested-selectors. Le symbol '.' déclare une classe. &. est donc une classe dans l'élément parent. Le symbol ' :' déclare un évènement. Ces deux symboles sont des sélecteurs.

Remarque : *console.log(string).*
permet d'afficher une string dans la console log. Si on lui passe un objet la console affichera tt les attributs de la console

Remarque : *[ngStyle]="{'x'; source}".*
est une directive Angular qui permet de chercher une propriété dans une classe source pour l'affecter ici un Style html. Permet de binder à la propriété Style la valeur x extraite de source. Pour plus de détails voir : https://angular.io/api/common/NgStyle.

Mot clef : **super**.
permet de rappeler le constructeur de la classe mère. Dans le constructeur c'est la première instruction qu'il faut appeler. En dehors on peut l'appler à tt moment.

Remarque : *rem.*
unité de mesure relative qui se base sur la police du body (environnement le plus extérieur dans une page). Pour plus de détail sur les tailles : https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Values_and_units

Remarque : *Sur le padding.*
Il s'agit de l'écart entre entre le placement d'objet dans unvironnement et les bords de l'environnement. Il faut regarder sur une page (dans les outils de debug) pour visualiser sa.

Remarque : *sur le* ****ngFor**.*
Ce dernier peut fonctionner sur des container entier tant qu'ils sont contenue à l'intérieur de ces derniers. Voir aussi le **ngIf**

Remarque : *Sur le dossier* `asset`.
Dans ce dossier ; tt les ressources nécessaires pour le site (images, musique, video, gif...).

Remarque : *Pour afficher une image.*
balise <img src = "chemin">. Attention la balise ne se ferme pas comme les autres. On peut également utiliser le Binding d'Angular [img]=hero.image au lieu de src = "{{hero.image}}".

Remarque : *Pour régler les problèmes de taille d'image.*

Il faut faire un environnement ou container-image qui est définit dans le .scss avec un enfant de type balise image (déclaré > img{width = x%, height = y%} dans le scss. Se lit pour tt balises image contenues dans un container-image alors on appliques ces règles). Attention ce chevron (*chevron selector*) ne fonctionne que pour un enfant direct (pas pour un enfant d'un enfant), au niveau du html.

Remarque : *sur [class.une_classe] = condition.*

Signifie que l'on applique la classe css si la condition est vrai.

Remarque : *Sur le partage de classe entre component.*

Même avec l'export il n'est pas possible de directement appeler une classe d'un component dans un autre component. Par contre il est possible de faire des classes communes à tt les components et donc utilisables par tous.

Remarque : *Sur la directive [hide]="condition".*

Permet de masquer la div dans lequel il est contenue si la condition à l'intérieur est réalisée.

Mot clef : **Sur les selector**.

Les sélecteurs permettent de définir les éléments sur lesquels appliquer un ensemble de règles css. Il s'agit du nom de balise html associé au component. On le déclare dans dans la balise @Component au dessus d'une classe dans le .ts. A creuser : notion de QuerySelector.

Remarque : *Sur le fichier app-touting.module.ts.*

On donne à la variables **Route** un objet json (d'ou les accolades). Pour chaques routes on doit définir le `path :'chemin_du_path', component:nom_du_component`.

Remarque : *Sur la balise* ***<router-outlet></router-outlet>***.

Indique à partir de quel endroit on intègre un component par rapport à la `Route` que l'on saisie dans l'url. C'est un indicateur qui indique que l'on veut afficher des components à partir d'une route. Il est nécessaire pour utiliser les `Route`. Avant l'appel à <router-outlet></router-outlet> on peut déclarer une balise header avec une balise Nom_du_bouton (qui déclare un lien sur la route routerLink qui doit être déclarer dans le ficheir app-routing.modules.ts). Voir : balise **a** (*anchor*). On aurait aussi pu faire [routerLink]="nom_d_une_variable".

Remarque : *<footer></footer>.*

déclare un footer (footnote) sur une page.

Note :

Dans un constructeur on peut mettre une visibilité aux arguments ! Si l'argument n'existe pas encore comme attribut il sera crée à la volé par le constructeur.

Remarque : *console.log(une string).*

Permet de faire remonter des messages dans la console (dans le naviguateur) lors de l'éxécution de.

Remarque : *Sur la notion de filtre et de pipe.*

le | permet de déclarer des **pipes**, qui permettent d'appliquer des filtres particulier pour de la mise en forme.

Note :

En générale un componenet est lié à une route.

Remarque : *Sur l'appel d'un component.*
On peut appeler un component directement par sa route bien via son sélecteur.
Exemple : `<component_name></component_name>`.

Remarque : *Sur la balise title.*
La balise `<title>MonTitre</title>` permet de changer le nom afficher dans le naviguateur au niveau de la page. Il faut en définir une par component. Cela ne fonctionne pas. Il faut mieux passer par un service (ici setTitle) à injecter dans chaque component dans le OnInit.

Remarque : *Sur la notion de service.*
Il s'agit d'une classe réutilisable au besoin. pour générer un service :
`ng g s` (pour `ng generate service`). Un dossier service à la racine de `src` est une bonne idée. Un service doit possérer le décorateur **@Injectable** (voir note plus bas)

Note :
/** puis entrez permet de générer de la doc par défaut au dessus d'une méthode (avec la liste des paramètres d'entrées et de sortie @param et @return).

Note :
le décorateur `@Injectable` permet de dire à Angular qu'il peut instancier lui même ce service au besoin.

Remarque : *Sur la différence entre == et ===.*
On passe en générale par le triple égal pour vérifier la notion de strict différence (peut jouer si l'on compare deux type différent comme un `string` et un `number`).

2 Sur la connection à une API

Remarque : *Imports nécessaires.*
dans le `app.modules.cs` : **HttpClientModule**.

Note :
Il est préférable de dédier service à la gestion des requêtes http.

Remarque : *Sur le type **Observable**.*
Un Observable est un type qui représente une donnée qui arrive mais on ne sait pas quand exactement. C'est une variable asynchrone.

Remarque : *Sur les **interfaces** en TypeScript.*
Elle peuvent servir comme les interface classiques ou bien comme un support permettant de tapper directement du json ! (dans un objet non instanciable). Pour faire de l'affichage on peut donc se contenter d'une interface au lieu de faire une classe pleine et entière (constructeur, getter, setter, ...).

Note :
lorsque l'on récupère les données au format json depuis l'api, il faut que les noms renvoyées par l'api *collent* avec ceux de nos objets/ou interfaces les récupérant.

Note :

Dans le component qui appel le service on s'abonne ensuite à l'objet correspondant à la requete. C'est possible car http client renvoie forcément un observable. Il faut passer une fonction fléchée à subscribe (un peu comme un lambda)

Remarque : *Sur les fonction fléchées en TypeScript.*

On peut utiliser des lambda à la c# au format `var => var = qqchose{})`.

Note :

Comme on ne peut pas instancier une interface. Pour l'initialiser à une valeur valide (mm si vide ou null) on peut lui rajouter `|undefinded` après le type. Voir : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined

Remarque : *Sur le raisonnement en colonne de bootstrap.*

Bootstrap peut fonctionner en colonnes (col) allant de 1 à 12. col-4 correspond à 1 tier de la largeur totale ($\frac{4}{12}$), col-3 à un quart, col-6 à la moitié,... On peut aussi utiliser offset qui utilise le même système numéraire que col. Tt sa pour gérer des tailles ou espacements entre éléments. En plus : my-3 correspond à une marge dans le même système le long de l'axe y. mt marge en haut. Le total des colonnes sur une seul ligne doit au final faire 12.

Remarque : *Un peu plus de Bootstrap.*

Bootstrap a des alias. Exemple col-3 peut être remplacé par col-md-3, ou col-sm-2 (voir remarque au dessus). A ce moment la le chiffre est un maximum et l'alias md, sm (y'en a d'autre) gère la partie responsive de la page (l'ajustement auto en fonction de la taille total de la fenêtre.) Voir : https://getbootstrap.com/docs/4.0/layout/grid/ pour plus de détails. On doit par contre enchainer ces indicateurs pour gérer tt les types d'écrans (de vue dans le navigateur) possibles.

Remarque : *sur le* ***sort***.

sur une liste (déclaré comme `variable[]`) on peut faire `variable.sort(fonction_decomparaison)`. Voir dans le code car vraiment contre intuitif. Voir aussi : https://www.tutorialspoint.com/typescript/typescript_array_sort.htm

Remarque : *Sur le passage de paramètres à des components.*

Lorsque l'on écrit le path dans la `approuting.cs` on peut faire `path = mon_path+'/:name'` pour pouvoir ensuite injecter un nom déterminé à la place du paramètre `name` donné dans le path. Dans le component on peut injecter dans le constructeur un objet de type **ActivatedRoute** qui est l'objet qui va permettre de récupérer le nom (via un `subscribe`sur la réception des paramètres).

Note :

Pour faire des test via `consol.log` il faut le faire dans les `subscribe`. Cela vient du fait qu'en dehors du `subscribe` la donnée n'est pas récupérée tant qu'elle n'est pas explicitement demandée (le `console.log` en lui même ne suffit pas pour être considéré comme une demande).

Note :

Si il y a un tiret - dans le nom d'une variable dans l'api, il faut *protéger* ce nom dans la déclaration de variable dans notre interface (au sens container json) avec des guillemets simples ".

Remarque : *Sur le* ***routerLink***.

On est pas obligé d'être dans un environnement <a/> pour utiliser cette propriété. (voir la manière dont les pokemons sont atteint via leurs listes).

Remarque : *Sur* ***subscribe***.

Le `subscribe` sert un peu comme un gestionnaire de requête async. Sa à l'air différent du subscribe du pattern *Observer*. → **A creuser**. Le subscribe à un observable permet de traiter cette donnée par la suite (en l'affectant à une variable typescript définie en attribut par exemple.)

Remarque : *Sur le ?*.

Permet de déclare un objet (une instance de classe) comme probablement indéfinie. C'est un tricks JavaScript/TypeScript.

Remarque : *Pour transférer un objet d'un component à un autre.*

Il faut décorer l'attribut que l'on veut récupérer avec l'attribut **@input** (et ne pas oublier d'`import` le input dans la liste de package au début du fichier .ts).

Remarque : *Sur les sélecteurs et le transfert d'objet.*

On peut avec sa appeler un objet OA provenant d'un component CB, dans le component CA pour ensuite appeler le component CA dans le component CB → voir le yahtzee/hero-detail. C'est un exemple alambiqué mais en gros on peut transférer les objet et appeler les components dans d'autres components.

Remarque : *Sur les routes.*

Cela permet de naviguer entre component. Attention quand on déclare un `Route` dans le approuting.ts il ne faut pas mettre de / devant la déclaration du `path`.

Note :

Attention lorsque l'on utilise `routerLink` il ne faut pas oublier le premier / de la route.

Remarque : *Sur keyrefUndefined.*

TYpe à associer à une variable qui représente une variable nulle.

Note :

Quand on fait ma_variable : IPokemon|Undefined on déclare une variable de type IPokemon ou Undefined (nulle). Sa à l'air légèrement différent des pipes qui permettent les filtre mais pas tt à fait non plus → Subtilité à creuser.

3 Les formulaires

Note :

Les formulaires se déclare dans une balise <form><form/>.

Remarque : *2 types de formulaires.*

Par template et par code (template-driven et code-driven).

Remarque : *Sur le template driven.*

Il faut commencer par importer le module `FormsModule` et créer un objet vide correspondant au formulaire. Puis on va remplir des balises <input />.

Remarque : *Sur la syntaxe [(ngModel)]="user.attribut".*

Indique un binding dans les deux sens ! Les modifications de l'un s'applique à l'autre. NgModel

est nécessaire pour se baser sur un objet existant.

Note :
Pour valider le formulaire il est nécessaire d'avoir un buttun de type ="submit".

Remarque : *Pour que ce type de formulaire fonctionne.*
Il faut que les champs [name]="un truc" (de la prop html du NgModel) et les attributs [(ngModel)]="user.attribut" (de l'objet user) aient le même nom.

Note :
Les validations sont commune aux deux types de formulaire.

Remarque : *Sur la* ***validation****.*
on passe par les attributs de validation html. Le fait de nommer une balise <input #nom="ngModel"/> permet ensuite de gérer par la suite des comportements différents selon les actions utilisateurs. Pour cela on passe par des propriétés Angular (nom_donné_à_la_balise.propriété_Angular). On affecte à la balise <input/> différentes classes css en fonction de conditions à donner déterminé via ces propriétés angular.

Note :
la classe [class.is-invalid] est une classe de bootstrap.

Note :
pattern = "regex" permet de donner un regex. Pour gérer facielemtn les regex et leur interprétation voir : https://regex101.com/.

Remarque : *Pour afficher un message informatif sur une validation non passée.*
On utilise *ngIf="nom_de_linput.error ?.nom_de_lattribut_de_validation_html" pour faire apparaitre une div dédiée à sa.

Remarque : *Pour faire un champ à choix multiple.*
Il faut passer par une balise <select class="form-select">X<select/> à la place du <input /> avec une balise <option> <options/> (au niveau du X) à l'intérieur pour déclarer les différentes options possibles. Dans les options on peut faire un [ngValue]="valeur" ou valeur permet de faire un bind avec un objet. Voir : https://angular.io/api/forms/SelectControlValueAccessor pour un peu plus de détails.

Remarque : *Sur les formulaires par le Code.*
Il faut import ReactiveFormModule (un formulaire par le code est dit *reactive form*). On doit ensuite créer dans un component un objet de type FormGroup. Avec ce type de formulaire les modifications ne sont pas faite en direct, c'est la différence fondamentale. A partir de cette objet FormGroup on va ensuite créer les champs et les liaisons associées à partir d'un FormControl par champ du formulaire.

Note :
Tant que la validation n'est pas faite : l'objet associé au formulaire (ici User) ne sera pas modifié.

Remarque : *Sur les getters.*
Ils sont l'équivalent du #name + vérification du coté des formulaires par template.

Remarque : *Dans le html.*
[formControlName] est l'équivalent du [(ngModel)] des formulaire par template.

Remarque : *Sur `attribut != qq_chose`.*
le ! permet de déclarer une variable comme n'étant jamais null même si pas encore initialisé (assertion). voir : https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html et https://www.typescriptlang.org/docs/handbook/2/everyday-types.html.

Note :
Si la classe est le formulaire correspondent on peut pour l'assignation faire directement this.User = this.FormGroup.value au lieux de retapper la correspondance attribut par attribut (ou prop par prop).

4 Sur l'intégration de Angular en .NET

Dans visual studio on peut créer un projet en standalone ou bien en intégré (serveur commun au front Angular et au back .Net).
Dans un prjet complet on a ensuite un program.cs qui s'occupe du npm install et de lancer le serveur.

Remarque : *Pour debugger.*
On peut passer par le naviguateur → inspecter → debogger → Webpack → le fichier de notre code qui nous interesse. Voir également le lien suivant https://dvoituron.com/fr/2018/08/22/debugger-angular-vi pour faire du debug directement dans VS Code.

Remarque : *Le truc à comprendre.*
Le lazy loading des modules ! Lorsque l'on déclare une route, angular lit les fichiers js et cherche les liaisons. En remonatnt d'une liaison à l'autre il construit ensuite un énorme fichier js. Au lieu de faire sa on peut faire du lazy loading, c'est à dire importer un module spécifique pour un path donné si ce module est utilisé. Au lieu de mettre path component on peut passer par children : [{ loadChildren : () => import... },...] (dans le app-routing.module.ts). Dans ce cas on aura un fichier js spécifique par children loadé au lieu d'un énorme fichier commun à tt l'appli. → très important à gérer.