Angular

1. Qu'est-ce qu'Angular ?

Il s'agit d'une plateforme de développement construite sur Typescript, on peut le voir comme un « framework ».

C'est-à-dire qu'il contient tout le nécessaire pour réaliser des applications web, il vous offre des bibliothèques et librairies adéquates pour réaliser votre application.

2. Arborescence

A la racine du projet, les plus importants sont :

- Node_modules : contient les différentes bibliothèques d'Angular, vous pouvez les régénérer via un *npm install*
- Src : dossier cœur de notre application, il contient tous les fichiers de fonctionnement de notre application
- Angular.json : fichier qui contient les configurations spécifiques de l'application

Dans le dossier src :

- Index.html : la vue « root » de l'application
- Styles.scss : fichier « root » des styles de l'application
- App : le dossier « cœur » de l'application

Dans le dossier app :

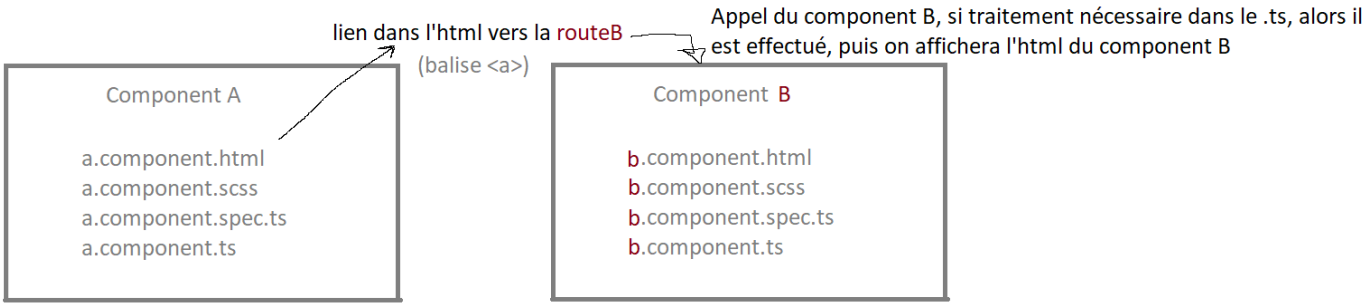
- App.module.ts : le module « root » de l'application, il contient tous les modules de l'application
- App-routing.module.ts : le fichier « root » de l'application pour décrire les différentes routes de l'application
- App.component : le component ou module par défaut d'un projet Angular

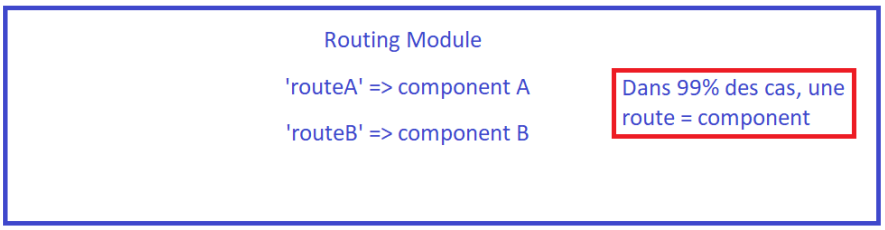
3. Rôle des components

Il s'agit d'un bloc qui compose votre application, un component est composé de 4 fichiers :

- xxx.html : représente la vue de votre component
- xxx.scss : représente la manière dont est stylisée votre vue
- xxx.ts : représente le code de votre vue
- xxx.spec.ts : un fichier pour déclarer des jeux de tests pour votre component

Principe des routes en Angular





4. Syntaxe Angular

(Pensez à faire des contrôles barre espace afin de vous aider !)

(Pensez à faire des ctrl + clic gauche pour la définition d'un élément, attribut ou méthode ou classe)

a. Affichage des variables depuis le component.ts

On affiche une variable entre des balises html, via les accolades.

Et si c'est un objet, on peut choisir l'attribut ou propriété et même fonction à afficher.

(/ !\ toute fois lorsque vous souhaitez afficher un objet, assurer vous que le type soit number, string ou boolean, sinon il vous renverra [Object object] et c'est quand même moins parlant…)

Il est possible d'afficher des variables aussi des attributs html, par exemple :

```
<p class="timer timer-{{ timer }}">{{ timer }}</p>
```

Si une classe vaut timer-(valeur mon attribut timer) alors elle sera appliquée sur la base html.

b. Le *ngFor

C'est une instruction qui permet d'itérer, au même titre qu'un for, sur des tableaux.

Elle se place sur une balise html, et reproduit le contenu de cette balise et de tout ce qu'il y a jusqu'à la balise fermante de celle-ci.

```
<strong *ngFor="let couvert of tiroir">
  {{ couvert.name }}
</strong>
```

Ici, je décide d'itérer sur mon tableau de nom tiroir, qui contient des objets de type Couvert (ayant juste un attribut name de type string), il va pour chacun afficher le nom du couvert dans une balise .

Il est possible de récupérer le numéro d'index en cours d'itération via :

```
<strong *ngFor="let couvert of tiroir; index as i">
```

i : est le nom de la variable correspondant à l'index en cours d'itération de notre boucle.

c. Le *ngIf

C'est une instruction qui permet d'effectuer des tests dans l'html.

Elle se place sur une balise html (elle ne reproduit pas la balise en question !), on peut effectuer des tests comme en typescript, par exemple une valeur, faire un .length sur un tableau etc.

Elle se complète très bien avec le ng-template, dans le else au cas où un traitement n'irai pas.

```
<span *ngIf="tiroir.length > 0; else emptyTiroir"></span>
```

d. Ng-template

Dans la ligne du *ngIf ci-dessus, emptyTiroir représente un ng-template, il se décrit dans la page html actuelle.

```
<ng-template #emptyTiroir>
  <strong>Tiroir vide :(</strong>
</ng-template>
```

Pour que le emptyTiroir fasse bien référence à ce ng-template, on le définit par le #emptyTiroir comme un attribut de balise html classique.

Il affichera alors le contenu du ng-template.

e. Les filtres html en Angular

Ils se représentent avec un| (Alt Gr + -/6) et permet d'appliquer un « filtre » sur la variable, il en existe plusieurs, voir : https://www.w3schools.com/angular/angular_filters.asp

Document des filtres pour formater les dates : https://angular.io/api/common/DatePipe

f. Les events

Un event s'ajoute sur un élément via le nom de l'évènement entre parenthèses, par exemple (click), (mouseover) etc

Par exemple pour le timer :

```
<div>
  <button (click)="startTimer()">O</button>
  <button (click)="stopTimer()">X</button>
  <p>{{ timer }}</p>
</div>
```

On a défini un évènement de click sur le bouton 'O' et sur le bouton 'X'.

L'attribut angular de l'évènement est bindé (bind = lier) à une méthode que l'on passera entre double quote, si cette méthode prendre des paramètres, il est possible de lui en passer.

Ainsi, dans votre component.ts lié à votre component.html, vous devez avoir, selon mon exemple, une méthode startTimer() et une méthode stopTimer.

g. Binding d'attribut html

```
<p [class]="classTimer" [title]="timer">{{ timer }}</p>
```

Entre [] on peut binder les attributs html, c'est-à-dire, qu'ils prennent une valeur obtenue depuis le component.ts, si jamais celle-ci est amenée à changer, alors elle changera en direct.

5. Les components

Une fois votre component créé, via la commande *ng generate component*, Angular vous créer les 4 fichiers nécessaires au fonctionnement d'un component, Angular vous l'ajoute aussi dans le app.module.ts afin qu'il soit utilisable dans toute votre application.

```
C:\Developpement\Cours\Angular\Cours\HB\initial-project (main -> origin)
λ ng generate component mardi
CREATE src/app/mardi/mardi.component.html (20 bytes)
CREATE src/app/mardi/mardi.component.spec.ts (619 bytes)
CREATE src/app/mardi/mardi.component.ts (272 bytes)
CREATE src/app/mardi/mardi.component.scss (0 bytes)
UPDATE src/app/app.module.ts (471 bytes)
```

Pour réutiliser un component, il faut utiliser le nom déclaré dans le 'selector', comme-ci-joint :

```
@Component({
  selector: 'app-mardi-timer',
  templateUrl: './mardi.component.html',
  styleUrls: ['./mardi.component.scss']
})
```

Ici, pour réutiliser notre component 'mardi' dans un autre component, il faudra utiliser une balise html :

<app-mardi-timer></app-mardi-timer>

C'est-à-dire, que si j'inclus mon component mardi dans un component XYZ, l'html du component mardi, apparaîtra en plus dans l'html du component XZY.

Dans le component créé par la commande, plus précisément le fichier .ts, vous remarquerez que la classe implémente l'interface *OnInit*, elle donne accès à la méthode ngOnInit :

```
export class MardiComponent implements OnInit {

  title: string;

  constructor() {
    this.title = 'Exercices du Mardi 20/04/2021';
  }

  ngOnInit(): void {
  }

}
```

Cette méthode ngOnInit fait parti du **cycle de vie des objets Angular (voir Lifecycle Hooks en anglais)** (**seulement dans les components !**), la méthode ngOnInit s'exécute après le constructeur.

Il est recommandé dans le **constructeur** de ne faire que des **initialisations d'attributs**, et de laisser les éventuelles **exécutions de méthodes** au **ngOnInit** (par exemple un appel vers une API).

6. Routing : navigation entre component

(Le cours ici assume que vous ayez bien créer votre projet Angular avec l'option *--routing*)

1. Déclaration d'une route

Pour définir les routes de notre application, il faut le faire dans le fichier *app-routing.modules.ts* qui se situe à la racine du dossier « *app* »

Pour déclarer les routes, il faut les ajouter à la variable constante de nom « *route* » :

```
const routes: Routes = [
  { path: 'mardi', component: MardiComponent },
  { path: 'regions', component: RegionIndexComponent },
  { path: 'departements', component: DepartementIndexComponent },
];
```

Ici, le *path* représente le nom à utiliser dans l'URL pour accéder au *component*, définit via l'attribut component de la *Routes*.

Afin de permettre le « routing » dans l'application, il faut bien s'assurer que le *app.component.html* a bien une balise de nom :

```
<router-outlet></router-outlet>
```

Ce qu'il se passe, c'est que lorsque j'appellerai la route de path « mardi », le *router-outlet* affichera le component *MardiComponent*, il ajoutera la balise *<app-mardi></app-mardi>* dans le code.

2. Utilisation des routes

Une fois nos routes déclarées dans le *app-routing.module.ts*, on peut les utiliser dans notre application via l'annotation Angular « *routerLink* », par exemple :

```
<a routerLink="/mardi" class="btn btn-primary">Exercices Mardi 20/04/2021</a>
```

Ici, on déclare une « *balise html a* », qui au lieu d'avoir un attribut href, a un attribut *routerLink*, dans lequel vous lui donner le nom de la route où aller lorsque l'utilisateur cliquera sur le lien (il faut bien mettre le / en début de nom de route ici).

3. Routes et paramètres

Pour déclarer une route ayant un paramètre, on le fait toujours dans le *app-routing.module.ts* et on nomme notre paramètre précédé de « : », ici **code**, comme ceci :

```
{ path: 'regions/:code', component: RegionShowComponent },
```

Ensuite, dans le component lié à notre route, ici le *RegionShowComponent*, il faut récupérer un objet de type *ActivatedRoute*, que l'on récupèrera par injection de dépendance (Dependance Injection en anglais, ou DI).

Définition de l'injection de dépendance :

Un constructeur prenant en paramètre un objet, revient à faire de l'injection de dépendance, c'est à dire que l'on indique au Framework (ici Angular) qu'il va assurer la création de l'objet en question. Ainsi, on n'a pas à s'en occuper !

```
constructor(private activatedRoute: ActivatedRoute) { }
```

Dans cet exemple, on passe via l'injection de dépendance un objet de type *ActivatedRoute*, qui va donc servir pour la récupération de paramètre à notre component.

Déclarer en paramètre à un constructeur (et uniquement un constructeur !) une visibilité sur celui-ci (public, private ou protected) revient à dire à Typescript que l'on veut un attribut de classe du même nom avec la visibilité que l'on lui a précisé. Dans notre exemple on veut un attribut de nom *activatedRoute* qui est privé et de type *ActivatedRoute*.

La ligne de l'exemple est équivalente à ce code :

```
private activatedRoute: ActivatedRoute;

constructor(activatedRoute: ActivatedRoute) {
  this.activatedRoute = activatedRoute;
}
```

Pour récupérer le paramètre de notre route (ici **code**, voir plus haut) il faut le récupérer via le ngOnInit :

```
ngOnInit(): void {
  this.activatedRoute.params.subscribe((params) => {
    alert(params.code);
  });
}
```

Via l'attribut *activatedRoute* que l'on a récupéré via injection de dépendance, on peut accéder à son attribut params (qui est de type Observable), c'est à dire qu'il attend d'être notifié, il attend quelque chose, ici le paramètre de la route, notre **code**.

On représente le fait qu'il attente, via la méthode *subscribe*, c'est-à-dire que l'on décrit un abonnement à quelque chose, ici le fait de récupérer le paramètre de la route.

Comme pour un évènement Javascript, il faut déclarer une fonction (ici, communément appelée « fonction flèche » : « () => »), comme pour un évènement Javascript, cette fonction a un paramètre de nom params, de type Params, qui permettra de récupérer l'argument de la route pour le component.

Pour se faire, via la variable params, on récupère le paramètre de la route, de notre component, en utilisant le nom que l'on a défini dans l'*app-routing.module.ts*, ici "**code**".

(Attention : l'autocomplétions ne fonctionne pas pour ça, il faut bien faire attention au nom que l'on écrit !)

La « fonction flèche » :

```
this.activatedRoute.params.subscribe(function(params) {

this.activatedRoute.params.subscribe((params) => {
```

Ces deux lignes sont identiques, on déclare bien une fonction qui possède un paramètre de nom « params ».

7. Les services

Un service est là pour renforcer l'idée qu'un component doit être autonome, et n'être « qu'une simple interface graphique avec l'utilisateur », c'est-à-dire que c'est le rôle du service de récupérer les données depuis une API, ou autre.

Souvent, on pourra voir le service comme un « CRUD », le service permet de dispatcher les données dans les différents components où le service sera appelé.

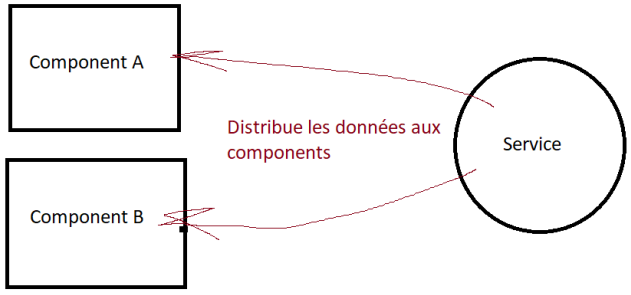
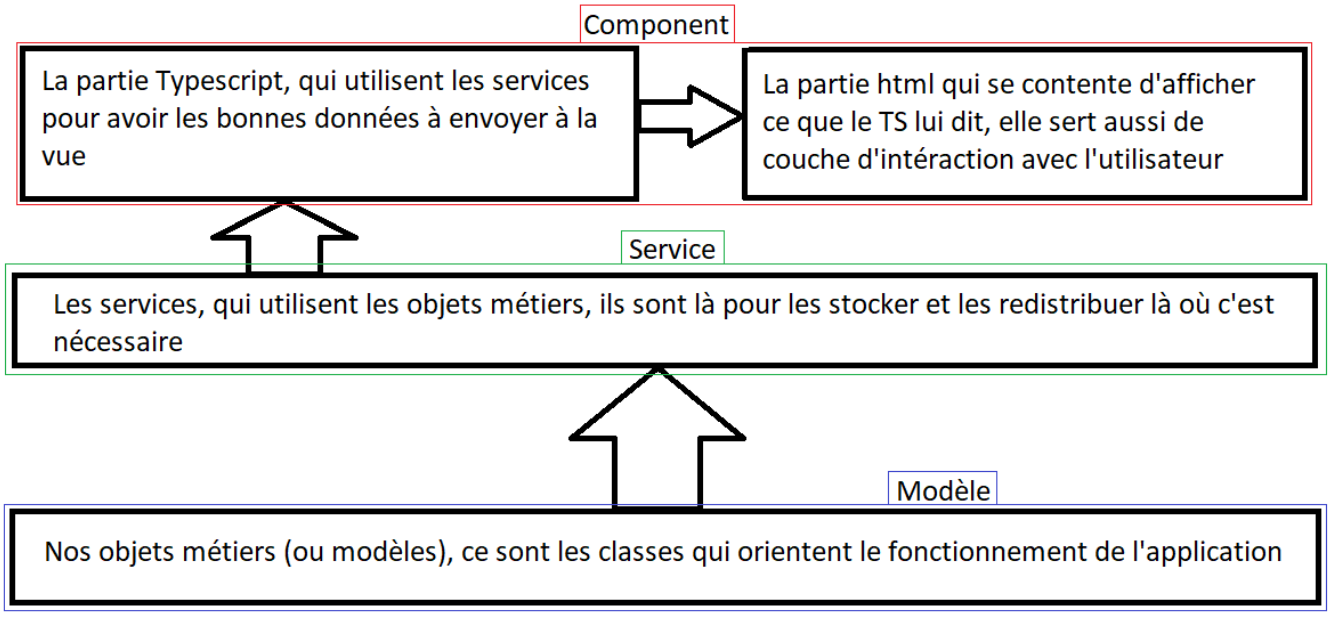


Schéma récapitulatif de l'architecture Angular :



8. Input : communication entre component (partie 1)

Il sert à la communication entre deux components, mère et fille, c'est-à-dire la communication entre un component qui intègre un autre component.

Il s'agit d'un autre moyen de communication entre les components, autre que les routes que l'on a vu jusqu'à maintenant.

1. *Component fille*

On doit déclarer un attribut à la classe fille, auquel on va ajouter une « décoration » de nom *@Input* :

```
@Input()
unNomDAttribut: string;
```

On doit faire implémenter l'interface *OnChanges* au component fille :

```
export class UnNomDeComponent implements OnInit, OnChanges {
```

Implémenter *OnChanges* permet d'ajouter la méthode *ngOnChanges* à notre classe, ce qui lui permet d'avoir une notion « de détection des changements », cela fait parti du **cycle de vie des objets Angular (voir Lifecycle Hooks en anglais)** (**seulement dans les components !**).

```
ngOnChanges(changes: SimpleChanges): void {
  // OnChanges ne sait pas directement le nom de l'attribut qui a changé, il faut le
guider :
  // ici on veut que OnChanges récupère un changement de unNomDAttribut, donc on
reprend le nom de l'attribut : unNomDAttribut
  const oldUnNomDAttribut = changes.unNomDAttribut.previousValue;
  const newUnNomDAttribut = changes.unNomDAttribut.currentValue;
  // Exemple :
  // Au premier passage : oldUnNomDAttribut vaudra undefined
  // Au premier passage : newUnNomDAttribut vaudra la valeur envoyé par le onChanges
via le SimpleChanges en paramètre de la fonction
  if (oldCodeRegion !== newCodeRegion) {
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\\
    // Seulement le contenu du if va changer lors d'un ngOnChanges
    // Le reste sera toujours identique
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\\
  }
}
```

(NB : Lorsque le component mère fera changer la valeur de l'attribut `unNomDAttribut`, la nouvelle valeur sera automatiquement set à l'attribut)

2. *Component mère*

Dans le component mère, on appellera le component fille via les balises html :

```
<app-un-nom-de-component [unNomDAttribut]="unNomDAttributMere"></app-un-nom-de-
component>
```

Le nom de l'attribut avec la décoration @Input dans le component fille

Nom de l'attribut envoyé depuis le component mère