HUMAN booster
VOTRE SOLUTION COMPETENCE

Angular

Table des matières

1. Qu'est-ce qu'Angular ?

Il s'agit d'une plateforme de développement construite sur Typescript, c'est un « framework ».
C'est-à-dire qu'il contient tout le nécessaire pour réaliser des applications web, il vous offre nativement tout le nécessaire pour réaliser une application complète.
On dit d'Angular qu'il est « Single-Page Web » et aussi « Progressive Web App », c'est-à-dire qu'il faut voir Angular comme une application ou plusieurs applications communiquant entre elles, plutôt qu'un site web contenant plusieurs pages.

2. Arborescence

A la racine du projet, les plus importants sont :

- **Node_modules** : contient les différentes bibliothèques d'Angular, définit par le fichier « **package.json »,** vous pouvez les régénérer via un *npm install*
- **Src** : dossier cœur de notre application, il contient tous les fichiers de fonctionnement de notre application
- **Package.json** : fichier qui contient les dépendance 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.t**s : un fichier pour déclarer des jeux de tests pour votre component

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 (la variable à afficher doit être **public** !)
Et si c'est un objet, on peut choisir l'attribut ou propriété et même la 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 dans 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 balise html.

b. Le *ngFor

C'est une instruction qui permet d'itérer, au même titre qu'un for, sur des tableaux, dans notre vue.
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 un ng-template (voir partie suivante), dans le else au cas où un traitement n'irai pas.

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

On peut aussi faire un « if » de manière implicite via un « ? » :

```
{{oneCase.piece?.nom}}
```

Si la propriété *piece* existe, alors on affichera son nom, sinon on n'affichera rien.

d. ng-template

Dans la ligne du *ngIf ci-dessus, emptyTiroir représente un **ng-template**, autrement dit un template Angular, sa particularité est que par défaut il n'est pas affiché.
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| (pipe, 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
Imaginons un timer qui se lance et qui se stoppe :

```
<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 doit 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 de définit pour que cela fonctionne.

g. Binding d'attribut html

Entre les crochets - [] - 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.

C'est aussi un excellent moyen de dynamiser notre vue, par exemple il est possible d'appliquer une classe CSS ou de désactiver un bouton directement avec un test dans l'HTML !

Exemple #1 : je veux masquer une balise p, tant que le nombre de round de ma partie de yatzee n'est pas arrivé à 13.

```
<p [hidden]="yatzee.totalRound !== 13"><strong>Fin de partie</strong></p>
```

Exemple #2 : j'applique à la classe css la valeur de l'attribut « classTimer » sur ma balise p, si la valeur de classTimer est amenée à changer, alors les changements seront appliquée sen temps réel. Pour l'attribut title, il évoluera en fonction de l'évolution de l'attribut timer.

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

Exemple #3 : je souhaite appliquer une classe css à ma balise, en fonction d'une condition précise

```
<p class="card-text"
   [class.noHitPoint]="starship.hitPoint <= 0"
   [class.hitPoint]="starship.hitPoint > 0"
>
```

Il existe aussi une directive angular « **ngStyle** », qui permet de modifier directement le style d'une balise via des attributs du .ts. La différence par rapport à la balise « **style** », est que ngStyle s'écrit comme un objet, par exemple :

```
<div class="row"
     [ngStyle]="{
     'background-image':
        'url(' + planet.pathImage + ')',
        'height': '960px',
        'width': '100%',
        'background-size': 'cover'
     }"
>
```

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>

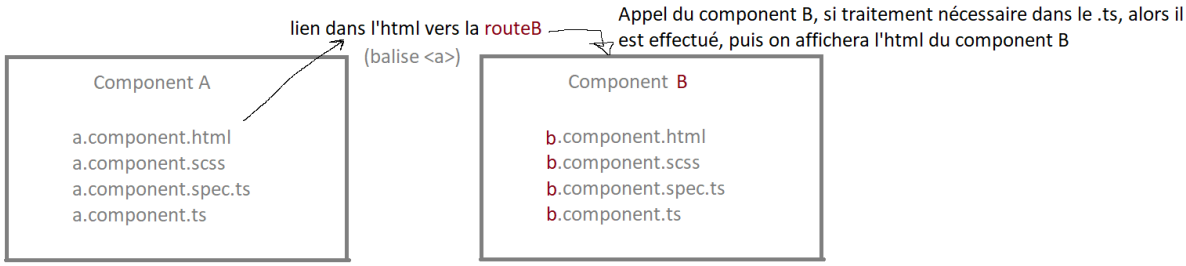
C'est-à-dire, que si j'inclus mon component mardi dans un component XYZ, l'html du component mardi, apparaîtra en plus dans l'html du component XZY.

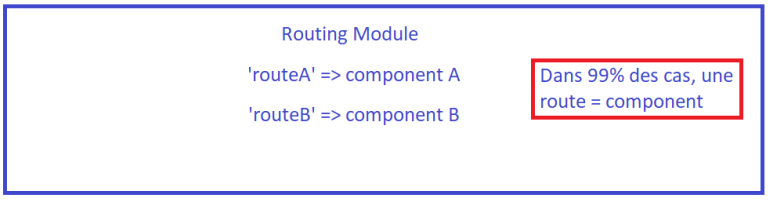
Dans le component créé par la commande, plus précisément le fichier .ts, vous remarquerez que la classe implémente l'interface *OnInit*, elle donne accès à la méthode ngOnInit :

```
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





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.

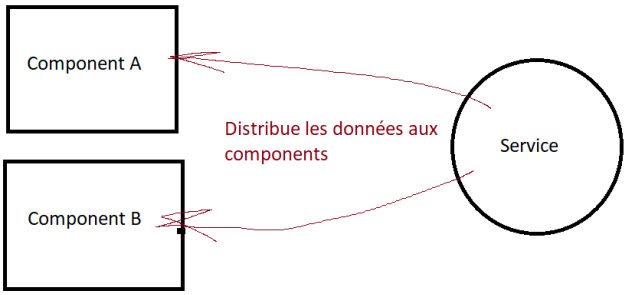
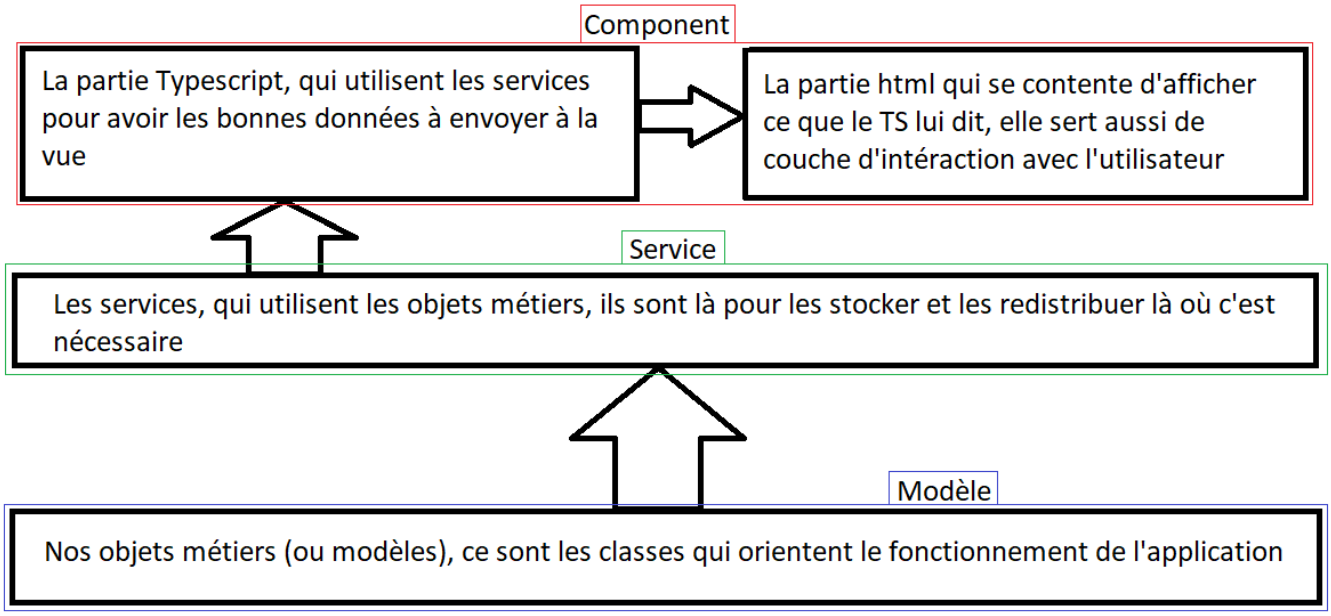


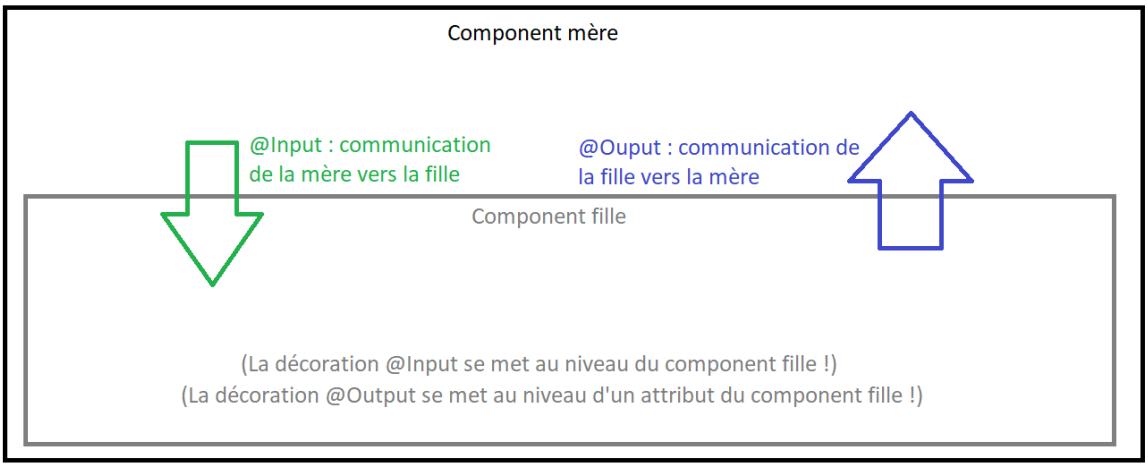
Schéma récapitulatif de l'architecture Angular :



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

9. Les Output : communication entre component (2ème partie)

Cette fois-ci, c'est la fille qui cherche à communiquer avec la mère, afin de lui indiquer qu'elle doit se mettre à jour.
La décoration @Ouput() se place au niveau d'un attribut du component fille.

<u>Component fille :</u>

Cet attribut doit être de type EventEmitter<**TypeDeVotreChoix**>, il est variable en fonction de notre besoin.
Il faut bien penser à instancier cet objet dans votre constucteur, au même titre qu'un autre objet via un new EventEmitter< **TypeDeVotreChoix** >()

Lorsque vous avez « déterminer » sur quel action votre component fille doit mettre à jour le component mère, il faut utiliser la fonction « emit » depuis votre attribut de type EventEmitter, elle prendra en paramètre un objet de **TypeDeVotreChoix**.

<u>Component mère :</u>

Il faut déclarer dans l'html, au niveau de l'intégration du component fille, quel attribut et la manière dont on souhaite le récupérer :

```
<app-card-starship [starship]="playerStarship"
                   [newHitPoint]="playerNewHitPoint"
                   (shootingStarship)="setLastStarshipShooter($event)"
></app-card-starship>
```

Ici, je souhaite récupérer la valeur envoyée par shootingStarship (l'EventEmiitter de ma fille) via la méthode setLastStarshipShooter, le **$event** doit toujours être renseigné et toujours de cette manière, il aura directement la valeur émise par le shootingStarship du component fille, et son type (type définit par l'EventEmitter du component fille)

10. Formulaire

a. Template driven

Lors d'un formulaire par le template, tout est dirigé par le template, néanmoins il vous faut au minimum un objet instancié « vide » du type souhaité dans votre .ts.
Si vous utilisez un objet avec ses attributs remplis, le formulaire sera initialisé dessus.

Pour utiliser un formulaire « template driven » on utilise la directive Angular **ngForm** sur un alias pour nommer notre formulaire, le tout sur une balise form :

```
<form (ngSubmit)="onSubmit()" #formWeapon="ngForm">
```

Ici je créé un **ngForm** qui se nommera **formWeapon**, et lorsqu'il sera soumis, via **ngSubmit**, il appellera la fonction **onSubmit**.

Tout comme un formulaire html classique, on utilise un bouton de type submit pour confirmer la soumission du formulaire :

```
<button class="btn btn-primary"
        type="submit"
>
  Soumettre
</button>
```

A l'intérieur d'un formulaire **ngForm**, on peut utiliser la directive **[(ngModel)]** qui permet de binder l'input ou n'importe quel champ de notre formulaire à un attribut ou une propriété de notre objet (celui qui a été créé dans le .ts)

Exemple :

```
<input [(ngModel)]="weapon.name" type="text">
```

Ici je définis un input de type text qui sera « binder » à la propriété name de la weapon.

Il est aussi possible de nommer notre input (ou n'importe quel autre champ) de notre formulaire afin de faire les vérifications dessus :

```
<input type="text"
       #nameWeapon="ngModel"
       name="name"
       [(ngModel)]="weapon.name"
       required
>
```

L'alias donné à l'input est représenté par le #, ici mon input se nomme **nameWeapon**, il faut bien penser à lui donner la valeur de **ngModel,** même si la directive **[(ngModel)]** est nécessaire pour « binder » notre formulaire à un objet, il est nécessaire de lui préciser un attribut « name ».

Nommer un champ d'un formulaire est très important car cela permet de pouvoir effectuer des vérifications sur le formulaire :

```
<input type="text"
       #nameWeapon="ngModel"
       name="name"
       [(ngModel)]="weapon.name"
       [class.is-invalid]="nameWeapon.touched && !nameWeapon.valid"
       class="form-control"
       placeholder="Nom"
       required
>
```

Ici, j'applique la class css « is-invalid » si la condition est remplie.
Angular propose plusieurs propriétés pour nous aider lors de nos vérifications de données :

- **Touched** : si l'utilisateur a au moins cliqué une fois sur le champ
- **Valid** : si le champ est valide, c'est-à-dire qu'il rempli les conditions de validation, ici le champ est juste « required » (ngForm se base sur les validations HTML)
- **Dirty** : si l'utilisateur a modifié le champ
- **Invalid** : si le champ est invalide, c'est-à-dire qu'il ne rempli pas les conditions de validation.

Ainsi il est possible d'afficher le bon message d'erreur en fonction de ce que fait l'utilisateur :

```
<div *ngIf="nameWeapon.invalid && (nameWeapon.dirty || nameWeapon.touched)"
     class="alert-danger">
  <div *ngIf="nameWeapon.errors.required">
    Name is required.
  </div>
</div>
```

Ici, je souhaite afficher un message d'erreur si le champ name n'a pas été rempli, mais que l'utilisateur a au moins sélectionné une fois le champ ou qu'il l'a modifié.

Afin d'offrir une parfaite expérience utilisateur et aussi de s'assurer que l'utilisateur ne fait pas n'importe quoi, il est recommandé de bloquer le bouton de soumission tant que le formulaire n'est pas valide :

```
<button class="btn btn-primary"
        type="submit"
        [disabled]="!formWeapon.valid">
  Soumettre
</button>
```

b. Code driven

Les formulaires par le code ne se basent pas sur un **ngForm**, mais sur un objet de type **FormGroup**, qu'il faut déclarer dans le code, il faut aussi un objet du type que l'on veut créer instancier « vide » (comme pour les formulaires « template driven »).

L'instanciation du FormGroup se fait de cette manière :

```
this.factionFormGroup = new FormGroup(
  {
    name: new FormControl(
      this.faction.name, [
        Validators.required,
      ]
    ),
    pathImage: new FormControl(
      this.faction.pathImage, [
        Validators.required,
      ]
    )
  }
);
```

Un **FormGroup** est composé de plusieurs **FormControl**, un **FormControl** représente un champ de notre formulaire.
name : correspond au nom de l'input, il faudra lui faire une propriété pour l'utiliser dans l'html.
this.faction.name représente la valeur initiale de notre formulaire (utile pour la modification)
Le deuxième paramètre représente les validations requises pour le champ (ici Validators.required)

Exemple de propriété pour récupérer un champ de notre formulaire dans l'html :

```
get name(): AbstractControl {
  return this.factionFormGroup.get('name');
}
```

A contrario du formulaire par le template, notre objet n'est pas modifié en direct lorsque l'on rempli le formulaire, cela doit se faire au moment de l'action effectuée par la soumission avec un :

```
this.faction = this.factionFormGroup.value;
```

Où **factionFormGroup** est notre formulaire instancié plus haut (voir image précédente)

Dans l'html, on n'a pas besoin de déclarer un **ngSubmit** ni un alias pour **ngForm**, on utilise juste un attribut « **formGroup** » :

```
<form [formGroup]="factionFormGroup">
```

On affecte à l'attribut **formGroup** à notre objet de type **FormGroup** de notre .ts (ici **factionFormGroup**).

Tout comme un formulaire par le template, on peut utiliser les propriétés dirty, touched, valid ou invalid, cette fois via la propriété du champ ou l'attribut correspondant à notre formulaire (ici `factionFormGroup`)

Afin de faire la liaison entre un champ de notre formulaire, depuis notre objet **FormGroup** (côté code) et côté formulaire, il faut remplir un attribut de type **formControlName** :

```
<input [class.is-invalid]="name.invalid && name.touched"
       placeholder="Nom"
       class="form-control"
       type="text"
       name="name"
       formControlName="name"
>
```

Le nom du formControlName doit être le même que l'attribut html « name ».

c. Select sur une entité

Il arrive que des objets aient un attribut du type d'un autre objet, souvent on utilise un select pour proposer toutes les possibilités à l'utilisateur et restreindre son choix à une seule possibilité.

```
<select class="form-select"
        name="faction"
        formControlName="faction"
>
  <option *ngFor="let faction of factionService.getArrayFaction()"
          [ngValue]="faction"
  >
    {{faction.name}}
  </option>
</select>
```

(L'exemple ici se base sur un formulaire par le code, notez que l'attribut **formControlName** est sur le select)

On créé les options du select à partir des objets récupérés depuis un tableau, ici on utilise **[ngValue]** pour lui donner un objet et le set à notre objet final.
(Ici il s'agissait d'un **Starship** qui a un attribut de type **Faction**, le bind se fait directement)

d. FormArray

Un **FormArray** permet de stocker plusieurs **FormControl**, autrement dit plusieurs formulaires.
L'idée est de pouvoir dupliquer un formulaire pour ajouter plusieurs objets lorsque c'est nécessaire (Exemple : un **Starship** peut avoir plusieurs **Weapon**, autrement dit il a un tableau de **Weapon**, grâce au **FormArray** on peut lui ajouter plusieurs **Weapon**).

Exemple :
On créé dans le **FormControl** un **FormArray**, initialisé avec un seul formulaire (impose donc le fait qu'un vaisseau ai au moins une seule arme, car le formulaire est obligatoire).

```
weapons: new FormArray([
  new FormControl('', Validators.required),
]),
```

Ensuite, il faut lui définir une propriété pour accéder au champ des **weapons** dans l'HTML.

```
<div formArrayName="weapons">
  <div *ngFor="let _ of weapons.controls; index as indexFormControl">
    <h4>Weapon #{{indexFormControl + 1}}</h4>
```

Au lieu de définir un **formControlName** on définit un **formArrayName** avec le nom de la valeur dans le .ts.

On boucle sur le tableau de **weapons** (nom de la propriété dans le .ts) sur l'attribut **controls** de celui-ci, qui correspond aux formulaires présents. Ce qui nous intéresse ici c'est surtout l'index pour « nommer » nos formulaires.

Ici, on voulait que chaque formulaire permette d'ajouter une **Weapon** au **Starship**, on va pouvoir les sélectionner une à une via un select.

```
<select class="form-select"
        [formControlName]="indexFormControl"
>
  <option *ngFor="let weapon of weaponService.getArrayWeapon()"
          [ngValue]="weapon"
  >
    {{weapon.name}}
  </option>
</select>
```

Vous noterez ici que le formControlName est bien sur le select et non sur l'option ! Le **formControlName** prend la valeur de l'index de notre ngFor.
afin de « binder » nos **Weapons** ajoutées au **Starship** sur le select il faut utiliser le **[ngValue]** qui permet de gérer des objets, alors que l'attribut html value, ne permet de gérer que des types primitifs.
Pour gérer le dynamisme de notre formulaire, il faut ajouter des boutons + et -, afin d'ajouter ou de supprimer un formulaire.

```
<button type="button"
        (click)="addWeaponField()"
        class="btn btn-success"
>
  +
</button>
<button type="button"
        (click)="deleteWeaponField(indexFormControl)"
        class="btn btn-danger"
>
  -
</button>
```

Le bouton + fait un push dans notre **FormArray** :

```
addWeaponField(): void {
  this.weapons.push(new FormControl('', Validators.required));
}
```

Et le bouton – retire le **formControlName** courant, en gardant un minimum de 1 élément :

```
deleteWeaponField(index: number): void {
  if (this.weapons.length !== 1) {
```

```
    this.weapons.removeAt(index);
  }
}
```

11. Requêtes vers une API

Afin de réaliser des requêtes http vers une API (ou autre), il faut ajouter dans le **AppModule.ts** le module **HttpClientModule**, dans les imports :

```
imports: [
  HttpClientModule,
],
```

Il est recommandé de faire un service qui s'occupera de faire les requêtes http vers les API, ce Service utilisera, via injection de dépendance, un objet de type **HttpClient**.

Afin de récupérer nos données depuis une API, on va utiliser une méthode, qui va nous renvoyer des données sous forme **d'Observable**<TypeDeNotreChoix>, via la méthode **get** de **HttpClient** (Le type précisé avant la parenthèse correspond au type attendu réceptionné par l'API, ici **<Array<Region>>.**

Exemple :

```
getRegions(): Observable<Array<Region>> {
  return this.httpClient.get<Array<Region>>(GeoApiHttpService.urlRegion);
}
```

Ici, on souhaite récupérer un tableau de **Region**, ma fonction va donc renvoyer un Observable de tableau de **Region**.

Un Observable veut donc dire que l'on va s'attendre à recevoir un tableau de **Region** à un moment donné, cependant on ne sait pas quand exactement, entre autres à cause des latences serveurs ou autre. Il faut donc les traiter par la suite avec un subscribe (comme pour les routes).

Ainsi, dans les services où on en a besoin, on va « s'abonner » à l'**Observable** récupéré depuis notre **HttpService** :

```
this.geoApiHttpService.getRegions().subscribe((regions) => {
  this.arrayAbstractGeoApi = regions;
});
```

La variable **regions** est du type récupéré par **l'Observable<>**, autrement dit ici de type **Array<Region>.**