

The background is a light gray gradient. It is decorated with numerous realistic water droplets of various sizes, some with highlights and shadows, scattered across the surface. In the upper center, there is a faint, circular logo or watermark that appears to contain a stylized 'A' or similar symbol.

# ANGULAR

# QU'EST-CE QU'ANGULAR



Angular est un Framework Typescript, front-end et open-source réalisé par Google, il permet de faire des applications Web, mobile ou de bureau.

Qu'est-ce qu'un Framework ? Pourquoi utiliser un Framework ?

Framework signifie : « cadre de travail ». Il s'agit d'un ensemble de briques permettant de structurer et de faciliter le développement.

Ainsi, avec un Framework on a :

- Cadre de travail identique
- Un gain de productivité
- Utilisation de briques réutilisables
- Réponds aux normes de sécurités

# ***HISTORIQUE D'ANGULAR***



- Version 1.0.0 : Angular JS, sorti en 2009
- La première vraie version d'Angular est la 2.0, apparue en 2014
- Différentes versions sont sorties au fil des années, Google le maintient très régulièrement. Il faut compter à peu près une par an
- La version actuelle est la 14.0.0

Google fait en sorte que la rétrocompatibilité soit là !

# ***INSTALLATION D'ANGULAR***



Dans un terminal lancer la commande : `npm i -g @angular/cli`

Afin de vérifier l'installation, toujours dans un terminal,  
lancer la commande : `ng version`

« ng » est ce qui permet d'appeler les commandes « Angular »

(PS : Angular 14 utilise la version 16 de node)

# CREATION D'UN PROJET ANGULAR

Toujours dans un terminal, afin de créer un projet lancer la commande : `ng new NOM_PROJET`  
Vous devriez voir un prompt qui vous pose des questions :

```
? Would you like to add Angular routing? (y/N)
```

 Faire « y »

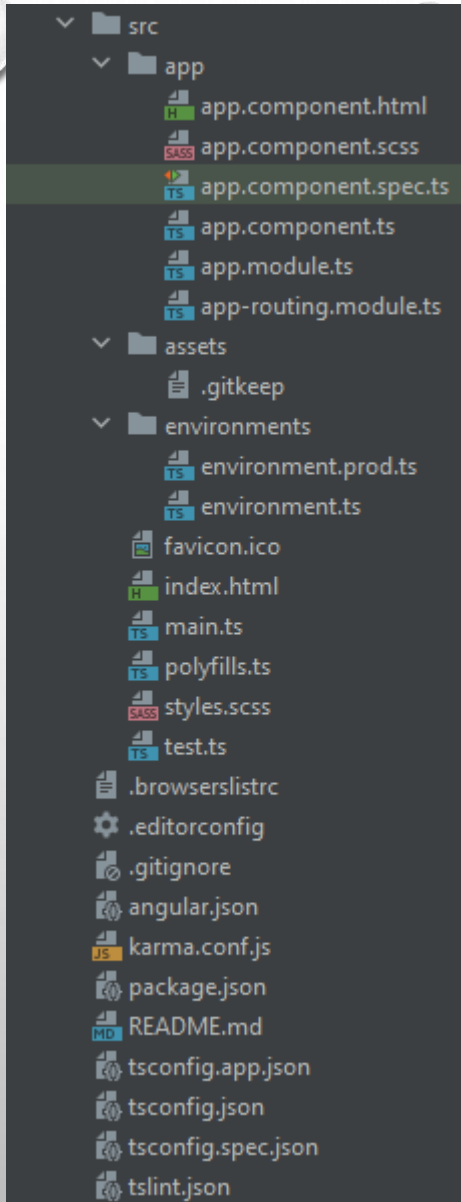
```
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
  SCSS  [ https://sass-lang.com/documentation/syntax#scss ]
  Sass  [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less  [ http://lesscss.org ]
```

Sélectionner « SCSS »

Maintenant, Angular va maintenant chercher les dépendances requises pour votre projet !



# ABO RESCENCE ANGULAR

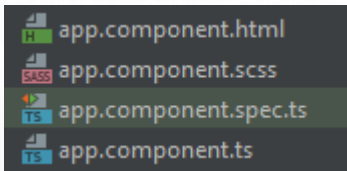


Ouvrez le dossier créé dans votre IDE, vous devriez voir cette arborescence.

À la racine du projet vous trouverez :

- **tsconfig.json** : celui nécessaire pour faire fonctionner Typescript
- **package.json** : le fichier où sera installé les autres dépendances du projet, comme Bootstrap
- Dans le dossier src :
  - **index.html** : le fichier principal de l'affichage du projet
  - **styles.scss** : le fichier SCSS principal du projet, il est commun à toutes les vues
- Dans le dossier src/app :
  - **app-routing.module.ts** : fichier de routing de l'application
  - **app.module.ts** : fichier qui organise votre projet et gère les imports de votre projet
  - **app.component.xxx** : un component (composant) Angular est composé de plusieurs fichiers, un html, un scss, un .ts et un spec.ts.

# COMPONENT ANGULAR



On a vu qu'un component Angular est composé de plusieurs fichiers, un html, un scss, un .ts et un spec.ts.

Les components en Angular sont les éléments core d'une application, c'est là où nous écrivons notre code Typescript et créerons nos vues html.

Explication des différents fichiers :

- .html : c'est ici que nous écrivons le code html, on l'appelle la « **vue** »
- .scss : c'est ici que nous écrivons le code scss, on stylisera la vue
- .ts : c'est ici que nous écrivons le code typescript relatif à la vue, on parle du « **controller** »
- .spec.ts : fichier permettant d'effectuer des tests du component

# COMMUNICATION INTER-COMPONENT

La logique Angular permet une communication interne simplifiée d'un component entre le **xxx.component.ts** et le **xxx.component.html**

Ce qui permet d'afficher « facilement » des données depuis le « **.ts** » dans l'« **.html** »

Vous ne pouvez afficher que des attributs ou des valeurs de retours de méthodes **public** depuis .ts

Exemple d'affichage dans l'html (en supposant que « **title** » soit un attribut public dans le .ts :

```
<h1>{{ title }}</h1>
```

```
<h1>{{ getTitle() }}</h1>
```





# JS EVENTS



En Angular, l'ajout d'évènement Javascript (**click, focus, over etc**), est simplifié, tout se faire directement sur l'HTMLElement :

```
<button (click)="close()">Close</button>
```

Ici, lorsque l'on clique sur le bouton « Close » on va appeler la fonction « **close()** », qui est présente dans le fichier .ts

# DIRECTIVE ANGULAR

## 1 / 2

En Angular, il est possible de faire des « if » dans l'html, afin d'afficher ou non des attributs, ou masquer complètement de l'html.

```
<div *ngIf="!isValidEmail">
```

Ici, on affichera le contenu de la <div>, si « **isValidEmail** » est false, sinon elle sera masquée.

On peut aussi indiquer à Angular d'afficher un autre code, ici si « **idFaction** » n'existe pas ou est false on affiche le bloc « nold »

```
<div class="container" *ngIf="idFaction; else noId">
```

En Angular, on peut déclarer un template alternatif, nommé « ng-template », et représenté par le **#nold** :

```
<ng-template #noId>  
  <p>Aucun ID trouvé</p>  
</ng-template>
```



# ***DIRECTIVE ANGULAR***

## ***2/2***

Il est possible d'effectuer des boucles, afin de parcourir les éléments d'un tableau par exemple :

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

Ici, on va parcourir le tableau « tiroir », on nomme chaque itération de la boucle « couvert » et on affiche la valeur dans la balise <strong>

Il est important de noter : la boucle « \*ngFor » va dupliquer la balise sur laquelle elle est mise.

Ce qui signifie qu'ici on va afficher le contenu des couverts dans une balise <strong>



# BINDING HTML

Il est possible d'appliquer une propriété html, selon une condition, ici on applique la propriété « hidden » si « upIsClicked » est true

```
[hidden]="upIsClicked"
```

Cela est valable pour les autres : « disabled », ou pour appliquer une classe CSS/SCSS en fonction :

```
[class.noHitPoint]="starship.hitPoint <= 0"  
[class.hitPoint]="starship.hitPoint > 0"
```

On applique la classe « noHitPoint » si les points de vie de « starShip » sont inférieur à 0

```
[style.background-color]="starship.faction.color"
```



# CREATION DE COMPONENT

## 1 / 2

On a vu qu'un component était représenté par 4 fichiers, mais quel est vraiment le rôle d'un component ?

Un component accentue le fait de découper notre code par fonctionnalité, nos components doivent être capable d'être autonome, c'est-à-dire que je dois être capable d'inclure mon component dans un autre, et qu'il fonctionne.

Afin de créer un component il faut lancer la commande suivante dans un terminal (à la racine de votre projet) : **ng generate component NOM\_COMP** (ou « **ng g c NOM\_COMP** »)





# CREATION DE COMPONENT

## 2/2

```
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)
```

Vous devez voir apparaître quelque chose comme ça, Angular vous génère tous les fichiers relatifs à votre component.

Autre point intéressant, votre component est aussi ajouté dans le « **app.module.ts** » :

C'est-à-dire qu'il est bien reconnu comme un component de votre application, et peut donc être utilisé !

```
@NgModule({
  declarations: [
    AppComponent,
    MardiComponent,
```

# LIFECYCLE HOOKS

## -- Lifecycle Hook Log --

```
#1 name is not known at construction
#2 OnChanges: name initialized to "Windstorm"
#3 OnInit
#4 DoCheck
#5 AfterContentInit
#6 AfterContentChecked
#7 AfterViewInit
#8 AfterViewChecked
#9 DoCheck
#10 AfterContentChecked
#11 AfterViewChecked
#12 DoCheck
#13 AfterContentChecked
#14 AfterViewChecked
#15 OnDestroy
```

Angular assure la création des composants, que ça soit le controller (.ts) ou la vue (.html).

Mais il permet aussi de « dynamiser » ses changements, c'est-à-dire, de pouvoir interagir à chaque instant du **cycle de vie du component**, on appelle ça les « lifecycle hooks ».

Il en existe de nombreux (voir ci-joint), on se sert principalement du **onInit** et du **onChanges**, mais il est intéressant de connaître les autres.

<https://angular.io/guide/lifecycle-hooks>

# COMMUNICATION ENTRE COMPONENT

Nous savons maintenant comment créer nos composants, mais comment les appeler entre eux et ainsi relier tous les composants de notre application ?

Dans un component, il y a une déclaration **@Component** au dessus de la classe :

3 attributs existent :

- selector : il représente le nom de la balise html du component
- templateUrl: le nom du fichier html du component
- styleUrls: le nom du fichier scss propre au component

Du coup, il est possible d'intégrer un component dans un autre en se servant du « **selector** », et en l'utilisant comme une balise html :

```
<exos-mardi></exos-mardi>
```

```
@Component({  
  selector: 'exos-mardi',  
  templateUrl: './mardi.component.html',  
  styleUrls: ['./mardi.component.scss']  
})  
export class MardiComponent implements OnInit {  
  |
```

# COMMUNICATION ENTRE COMPONENT : ROUTING 1/2

Un autre moyen de communiquer entre les composants, est d'utiliser ce que l'on appelle des « **routes** ». Vous vous souvenez en html lorsque vous faisiez des href ? On affichait dans l'url index.html, et bien c'est le même principe, sauf que l'on ne verra pas le nom du fichier appelé dans l'url, et c'est plus propre pour l'utilisateur.

Pour cela il faut modifier le fichier **app-routing.module.ts** (à la racine du dossier « **src** ») :

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

Dans le tableau de « Routes », il faut ajouter une ligne entre accolades, avec les attributs :

- **path** : nom de la route, c'est ce qui sera affiché dans l'url
- **component** : le nom du component à appeler lorsque l'on appelle cette route



# COMMUNICATION ENTRE COMPONENT : ROUTING 2/2

Nous avons déclaré une route pour le component, mais ce n'est pas pratique de l'appeler uniquement dans l'URL du navigateur, Il existe un moyen très simple en Angular de faire des liens entre les composants afin qu'ils communiquent entre eux : **la balise <a>**. Cependant, nous n'allons pas déclarer l'attribut href, Angular nous en offre un nouveau ! Il s'agit du « **routerLink** », en reprenant le nom donné dans le « **path** » du app-routing.module.ts :

```
<a routerLink="/mardi" class="btn btn-primary">Exercices du Mardi</a>
```

(PS : notez que cette fois il y a un « / » devant le nom « mardi », alors que dans le app-routing, non)

Dorénavant lorsque nous cliquerons sur le bouton « Exercices du mardi », l'HTML du component Mardi apparaîtra à l'écran (ou pas...), il nous reste une dernière chose à configurer pour qu'Angular comprenne qu'il doit utiliser le routing, **ajoutons cette balise dans le app.component.html**, ainsi l'html des composants appelé par routing remplacera cette balise dans le navigateur

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



# INJECTION DE DEPENDANCE

En Angular et dans beaucoup de framework (comme Symfony), il existe l'injection de dépendance ou « Dependency Injection » en anglais (DI).

Mais qu'est-ce que c'est ?

L'injection de dépendance est auto-gérée par Angular, elle permet aux composants de créer automatiquement les objets dont ils ont besoin pour fonctionner. Attention, cela ne fonctionne pas pour tous les objets, mais des objets Angular spécifique ou il faut indiquer à Angular comment le faire.

Comment cela fonctionne t'il ?

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

Il faut passer en paramètre au constructeur l'objet souhaité.

Mais que permet de faire le « private » devant le paramètre ?

Il permet de créer automatiquement un attribut dans la classe et de l'assigner.

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

# COMMUNICATION ENTRE COMPONENT : ROUTING AVEC PARAMETRE 1 / 2

Il est aussi possible de passer des informations d'un component à un autre, et ainsi éviter de perdre des informations, le « routing » permet de le faire ! Mais comment ?

```
{ path: 'year-finder/:age', component: YearFinderComponent },
```

Toujours dans le « **app-routing.module.ts** », vous pouvez déclarer une route, et dans l'attribut « path » lui ajouter un « / » suivi de « : », cela indique à Angular que le component appelé par cette route attend un paramètre. Après les deux points on indique le nom de ce paramètre, ici c'est « **age** ». Il faut aussi prévoir un changement dans l'HTML :

```
<a routerLink="/year-finder/{{age}}" class="btn btn-primary">Year Finder</a>
```

Ici, « age » est un attribut créé depuis le « .ts », il suffit d'ajouter un « / » après le nom de notre route et de passer notre paramètre.

# COMMUNICATION ENTRE COMPONENT : ROUTING AVEC PARAMETRE 2/2

Ci-joint, le code du component « **YearFinderComponent** », il a besoin d'un paramètre qui vient depuis le routing, il faut maintenant le traiter.

Pour cela il faut :

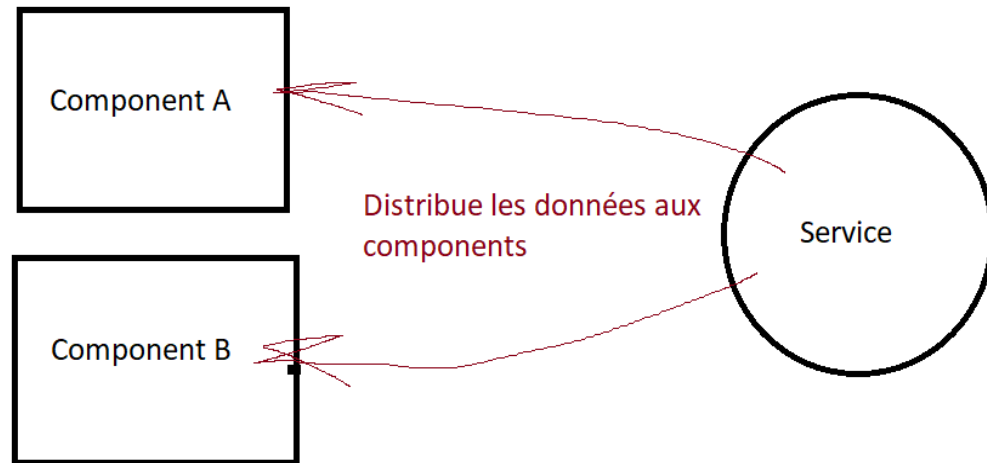
- Utiliser l'injection de dépendance avec un objet de type « **ActivatedRoute** »
- Via le « **OnInit** » on peut implémenter la méthode « **ngOnInit** ». Ainsi à l'intérieur de celle-ci on récupère le paramètre de la route.
- **Le traitement ici sera toujours le même**
- On appelle l'attribut « **params** » de l'objet « **activatedRoute** » et on appelle ensuite la méthode « **subscribe** » sur celui-ci.
- « **params** » permet d'accéder aux paramètres de la route, il suffit de reprendre le nom du paramètre de la route et de le récupérer dans notre attribut.

```
age!: number;

constructor(private activatedRoute: ActivatedRoute) { }

ngOnInit(): void {
  this.activatedRoute.params.subscribe( next: params => {
    this.age = params['age'];
  });
}
```

# LES SERVICES



Un service est là pour renforcer qu'un component doit-être autonome, il est là pour « rendre un service » aux composants de l'application. Souvent, il est utilisé pour distribuer des données aux composants, ou simplement comme un utilitaire permettant d'éviter une répétition de fonction.

Afin de créer un service, je vous recommande de vous mettre dans le dossier « service », puis de lancer la commande dans le terminal : « **ng g s NOM\_SERVICE** »



# LES APIS

« Application Programming Interface », est une interface logicielle qui permet de connecter un logiciel ou un service à un autre logiciel, afin de communiquer entre les deux

On se sert souvent des API pour renvoyer du JSON (Javascript Object Notation), qui est un format de données textuelles.

Le JSON est composé d'un ensemble clé/valeur (key/value) :

Ici « count » est la clé, et 1154 la valeur.

« results » contient un tableau

**Il est important de noter qu'un objet  
Typescript/Javascript n'est autre que du Json**

```
{
  "count": 1154,
  "next": "https://pokeapi.co/api/v2/pokemon/?offset=20&limit=20",
  "previous": null,
  "results": [
    {
      "name": "bulbasaur",
      "url": "https://pokeapi.co/api/v2/pokemon/1/"
    },
    {
      "name": "ivysaur",
      "url": "https://pokeapi.co/api/v2/pokemon/2/"
    }
  ]
}
```



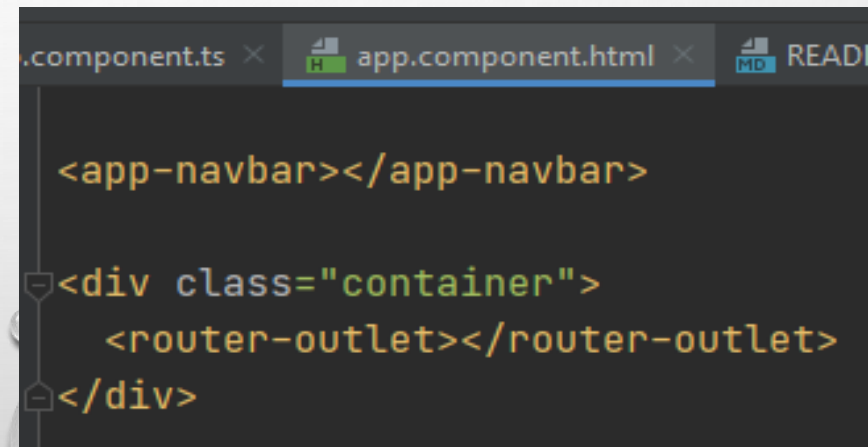
# COMMUNICATION ENTRE COMPONENT : INPUT 1 / 2

Nous avons vu que l'on peut communiquer entre les composants via le routing, avec ou sans paramètre, mais il est aussi possible de le faire avec la balise HTML du composant.

Cependant, lorsque nous sommes dans ce cas de figure, comment peut-on faire si l'on aimerait que le composant que l'on appelle ait un paramètre ? La solution est via la décoration **@Input**.

**On l'appelle la communication entre composant mère et fille. Où le composant fille est le composant dont la balise html est appelée à l'intérieur d'un autre composant.**

Par exemple, ici, le composant mère est le « app » et le composant fille est « navbar ».

A screenshot of a code editor with three tabs: 'component.ts', 'app.component.html', and 'README'. The 'app.component.html' tab is active, showing the following HTML code:

```
<app-navbar></app-navbar>

<div class="container">
  <router-outlet></router-outlet>
</div>
```

# COMMUNICATION ENTRE COMPONENT : INPUT 2/2

Pour faire fonctionner l'input il faut :

- Déclarer un attribut dans le **component** fille avec la décoration **@Input**

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

- Dans le component mère il faut assigner l'attribut déclaré en **@Input** directement sur la balise HTML

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

Cela aura pour impact de dire à l'attribut en **@Input**, qu'il prenne la valeur d'un attribut de la mère !  
Il est possible de passer des objets par ce biais.

# COMMUNICATION ENTRE COMPONENT : OUTPUT 1 / 2

Nous avons vu que l'on peut communiquer entre les composants depuis un composant « mère » vers un composant « fille », mais il est aussi possible, de faire l'inverse : **communication de la fille vers la mère**.

Pour cela il faut, **dans le composant fille** :

- Il faut créer un **attribut** de type **EventEmitter<T>**, où **<T>** représente le type de l'attribut que l'on veut renvoyer vers la mère. L'attribut sera toujours de type **EventEmitter<T>**.

```
@Output()  
shootingStarship: EventEmitter<IStarshipShooter>;
```

- Lorsque nous avons décidé du moment où informer le composant mère, il faut appeler la fonction « **emit** » sur l'objet **EventEmitter**

```
this.shootingStarship.emit(eventShooter);
```

# COMMUNICATION ENTRE COMPONENT : OUTPUT 2/2

Dans l'HTML du component mère il faut :

- Cette fois, l'attribut remonté depuis le component fille est appelé entre parenthèse, on passe ensuite une fonction pour traiter la remontée depuis le component fille avec en paramètre un **\$event**, ça sera toujours le cas

```
<app-card-starship (shootingStarship)="setLastStarshipShooter($event)"></app-card-starship>
```

- Angular comprends que c'est un évènement qui remonte d'un component fille, et sait le traiter

```
setLastStarshipShooter(iStarshipShooter: IStarshipShooter): void {
```

Dans le typescript du component mère, on type le paramètre du même type que celui déclaré dans l'**EventEmitter** du component fille :

# FORMULAIRE TEMPLATE-DRIVEN

Lors d'un formulaire par le template, tout est géré dans le .html de votre component.  
Le formulaire va se baser, via le **binding**, sur un objet et le modifiera en temps réel, ainsi dans le .ts de votre component vous devrez simplement **instancier un objet vide** :

```
user: User = new User();
```

Il faut aussi dans le **app.module.ts** importer le **FormsModule**



# FORMULAIRE

## TEMPLATE-DRIVEN 1 / 4

Lors d'un formulaire par le template, tout est géré dans le .html de votre component.

Le formulaire va se baser, via le **binding**, sur un objet et le modifiera en temps réel, ainsi dans le .ts de votre component vous devrez simplement **instancier un objet vide** :

```
user: User = new User();
```

Et avoir une méthode qui sera appelée lors de la soumission du formulaire.

Il faut aussi dans le **app.module.ts** importer le **FormsModule**

# FORMULAIRE

## TEMPLATE-DRIVEN 2/4

Dans l'html on se base toujours sur la balise html `<form>`, cependant vous n'avez pas à définir « **action** », on va laisser Angular gérer la soumission du formulaire via un `(ngSubmit)` qui aura la méthode où aller :

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

**#formUser** est le nom du formulaire, et il faut lui dire que c'est un « **ngForm** ».

On déclare les champs du formulaire comme en HTML, avec l'attribut `type` et on lui ajoute les directives Angular (**ngModel**) - l'attribut `html name` est primordial, car Angular l'utilise pour le `[(ngModel)]` :

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

# FORMULAIRE

## TEMPLATE-DRIVEN 3/4

**#nicknameUser** est l'alias que l'on donne à l'input, mais à quoi cela sert ?

Cela permet de pouvoir effectuer différents tests sur le champ, notamment s'il est valide ou non, pour cela on a les propriétés Angular :

- **Touched** : l'utilisateur a au moins cliqué une fois sur le champ
- **Valid** : le champ est valide, il remplit les validations html définies
- **Dirty** : l'utilisateur a modifié le champ
- **Invalid** : le champ est invalide, il ne remplit pas les validations html

Angular se base sur les validations HTML :

- required
- minLength
- etc

```
<input name="user[name]"
      class="form-control"
      type="text"
      [(ngModel)]="user.name"
      #userName="ngModel"
      [class.is-invalid]="(userName.dirty || userName.touched) && userName.invalid"
      [class.is-valid]="userName.valid"
      placeholder="Nom"
      required
      minlength="3"
>
```

# FORMULAIRE

## TEMPLATE-DRIVEN 4/4

Pour les validations on peut directement appeler celles que l'on a défini sur le champ (ici c'était **required** et **minLength**) et ainsi vérifier si le contenu de notre input correspond bien avec ce que nous attendions.

```
<div *ngIf="(userName.dirty || userName.touched) && userName.invalid" class="btn-danger ps-2 py-1">
  <div *ngIf="userName.errors?.required">
    Le nom est obligatoire
  </div>
  <div *ngIf="userName.errors?.minlength">
    Le nom doit faire au moins 3 caractères
  </div>
</div>
```

# FORMULAIRE

## CODE-DRIVEN 1/5

Dans le **app.module.ts**, il faut ajouter un module pour que les formulaires « code-driven » fonctionnent :

```
ReactiveFormsModule
```

Dans le **typescript du component**, on va déclarer un attribut de type **FormGroup** :

```
userFormGroup!: FormGroup;
```



# FORMULAIRE

## CODE-DRIVEN 2/5

Toujours dans le typescript :

- Lors de l'instanciation du **FormGroup**, on peut lui passer plusieurs paramètres, qui correspondent aux champs du formulaire.
- Un champ est un **FormControl**
- « **\_nickname** » est le nom du champ, cette fois on n'a pas à faire l'alias dans l'html
- On voit « **formState** », il s'agit de l'équivalent du [(ngModel)] dans le formulaire par le template, il est à renseigner pour définir l'état initial du formulaire.
- Le dernier paramètre « **validatorOrOpts** », comprend les différentes validations HTML à appliquer sur le champ

```
ngOnInit(): void {  
  this.userFormGroup = new FormGroup(  
    controls: {  
      _nickname: new FormControl(  
        formState: '', validatorOrOpts: [  
          Validators.required,  
          Validators.minLength(minLength: 5),  
        ]  
      ),  
    },  
  );  
}
```

# FORMULAIRE

## CODE-DRIVEN 3/5

Toujours dans le typescript :

- Il faut faire les getters pour les champs du formulaire, afin de pouvoir les réutiliser dans l'html (pour la vérification d'erreurs, entre autres)

```
get nickname(): AbstractControl {  
    return <AbstractControl>this.userFormGroup.get('_nickname');  
}  
  
get email(): AbstractControl {  
    return <AbstractControl>this.userFormGroup.get('_email');  
}
```

Dans l'html du component :

- On déclare cette fois un **[formGroup]**, auquel on lui donne l'attribut du formulaire que l'on a créé dans le typescript

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

# FORMULAIRE

## CODE-DRIVEN 4/5

Dans l'html du component :

- Cette fois la liaison entre de l'input de l'html et le code se fait via un attribut **[formControlName]** auquel on lui passe le nom du formControl que l'on a déclaré dans le code.
- Grâce au getter que l'on a fait dans le typescript, on peut appliquer facilement notre classe css « **is-invalid** » ou « **is-valid** », comme on l'a fait dans le formulaire par le template

```
<input type="text"  
      class="form-control"  
      formControlName="name"  
      placeholder="Nom"  
>
```

# FORMULAIRE

## CODE-DRIVEN 5/5

Dans l'html du component :

- Il n'y a pas de **ngSubmit**, l'action de soumission du formulaire se fait via un (**click**) avec le nom de la fonction à appeler.

Dans le typescrit du component :

- A la différence du formulaire par le template, le formulaire par le code ne modifie pas notre objet automatiquement. Il faut lui indiquer de quelle manière le faire !
- Les objets de type `formControl` ou `formGroup` ont accès à l'attribut « **value** » qui permet de récupérer les valeurs des champs, et ici de les set aux propriétés de l'objet.

```
<button class="btn btn-primary"
        [disabled]="userFormGroup.invalid"
        (click)="onSubmit()"
>
    Soumettre
</button>
```

```
onSubmit(): void {
    this.user.nickname = this.nickname.value;
    this.user.email = this.email.value;
}
```

# FORMULAIRE UTILITY

Dans un formulaire classique (html) on peut avoir un `<select>`, qui contient des `<option>`, il est aussi possible d'en définir avec Angular !

Cependant Angular nous propose une directive supplémentaire qui permet de récupérer directement un objet : `[ngValue]`

**Il s'agit de l'équivalent de l'attribut html : value, sauf qu'il ne prend que des number, string, boolean, autrement dit des types de base.**

```
<select [formControlName]='race' class='form-select'>
  <option *ngFor='let race of raceService.races' [ngValue]='race'>
    {{race.name}}
  </option>
</select>
```