Tutoriel d'utilisation du framework Angular

Logan SAGET

30/01/2026

Table des matières

1 Initialisation d'un projet :

1.1 Commandes utiles à la création d'un projet (dans le dossier voulu) :

```
ng new [nomProjet]
// Ou encore
ng new [nomProjet]
     --style=scss  // Langage du style
     --skip-tests=true

cd nomProjet
ng serve // Lancement du serveur
```

2 Bases d'un projet Angular :

2.1 Création et utilisation d'un composant :

Génération automatique :

```
    ng generate component page-accueil
```

Cette commande génère 3 fichiers :

- fichier HTML
- fichier de style
- fichier de script

Dans le fichier script on peut retrouver la classe qui défini ce composant :

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

@Component({
    selector: 'app-test',
    imports: [],
    templateUrl: './test.html',
    styleUrls: ['./test.scss'],
})

export class Test {

}
```

On pourra bien sûr rajouter des lignes dans cette classe (comme des attributs par exemple).

```
export class Comp1{
  title!: string;
  description!: string;
  createdAt!: Date;
  snaps!: number;
}
```

Afin d'initialiser ces attributs (dès l'appel du composant), on implémente l'interface OnInit :

```
export class Comp1 implements OnInit{
  title!: string;
  description!: string;
  createdAt!: Date;
  snaps!: number;

  ngOnInit(): void {
    this.title = "premierComposant";
    this.description = "premierComposant";
    this.createdAt = new Date();
    this.snaps = 0;
  }
}
```

Une fois initialisée, on peut utiliser ces variables dans le fichier.html du composant :

```
<H2>{{title}}</H2>
<p>{{description}}</p>
```

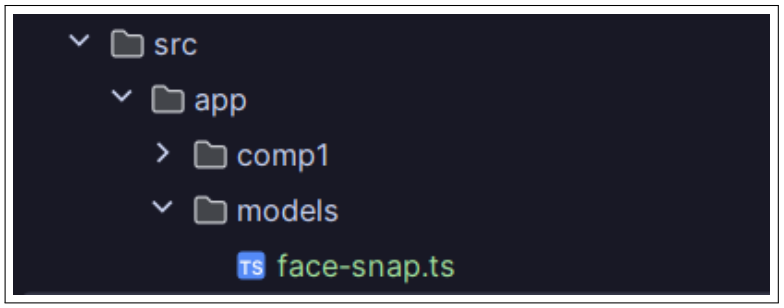
Ce qui donnera :

Pour ajouter une image, on la défini d'abbord comme nouvel attribut, puis on utilise une balise src entre "[]" :

```
<img [src]="url"> <!-- url etant un attribut defini dans
    componant.js -->
```

2.2 Personnalisation des composants :

Dans un premier lieu, on crée un package dans l'application dans lequel on va ajouter des classes :



Dans cette classe, on défini les attributs qui se trouvaient initialement dans le fichier ts du composant.

```
export class FaceSnap {
  constructor(public title: string,
              public description: string,
              public createdAt: Date,
              public snaps: number,
              public url: string) {}
      }
```

Dans le fichier "app.ts", nous allons maintenant déclarer des objets de type FaceSnap :

```
export class App implements OnInit {

  snap1!: FaceSnap;
  snap2!: FaceSnap;
  snap3!: FaceSnap;

  ngOnInit() {
    this.snap1 = new FaceSnap("premierComposant","
       premierComposant",new Date(),0,url);
    this.snap2 = new FaceSnap("2emeComposant","2
       emeComposant",new Date(),10,url);
    this.snap3 = new FaceSnap("3emeComposant","3
       emeComposant",new Date(),100,url);
  }
}
```

Une fois la classe crée, on peut retirer les attributs du composant et simplement crée un objet de type FaceSnap :

```
export class Comp1 implements OnInit{
  @Input() faceSnap!: FaceSnap;
}
```

Enfin, on peut appeler nos composants dans l'html de l'application en initialisant l'attribut crée :

```
<app-comp1 [faceSnap]="snap1"></app-comp1>

<app-comp1 [faceSnap]="snap2"></app-comp1>

<app-comp1 [faceSnap]="snap3"></app-comp1>
```

Avec cette manipulation, on peut personnaliser nos composants.

Il ne faut pas oublier dans le html du composant de remplacer les appels avec faceSnap.attribut :

```
<H2>{{faceSnap.title}}</H2>
<p>{{faceSnap.description}}</p>
<img [src]="faceSnap.url">
```

3 Gestion des événements :

La gestion des événements est beaucou plus simple grâce à angular. Il suffit de créer dans le TypeScript du composant des actions. Par exemple, on veut pouvoir liker un post et retirer le like :

```
export class Comp1 implements OnInit{
  @Input() faceSnap!: FaceSnap;

  snapButton!: string;
  snapOrNot!: boolean;

  ngOnInit(): void {
    this.snapOrNot = false;
    this.snapButton = "snaps"
  }

  onAddSnap() :void{
    if(this.snapOrNot === false){
      this.faceSnap.addSnap();
      this.snapOrNot = true;
      this.snapButton = "UnSnap";
    }else{
      this.faceSnap.delSnap();
      this.snapOrNot = false;
      this.snapButton = "snaps";
    }
  }
}
```

Enfin, on lie cet évènement à un bouton, ici, on parle de l'évènement "click", qu'on va donc écrire entre parenthèse dans la balise du bouton pour le définir :

```
<p>
  <button (click)="onAddSnap()">{{snapButton}}</button>
  {{faceSnap.snaps}}
</p>
```

4 Mise en place d'un service :

4.1 Création du service :

La mise en place d'un service est utile afin de regrouper les méthodes qui seront utiles à plusieurs composants. On les utilise aussi afin de faire appel à des API.

On définit un service de la manière suivante :

```
    @Injectable({
  providedIn: 'root'
})
export class FaceSnapsService {
  private faceSnaps: FaceSnap[] = [
    new FaceSnap("premierComposant", "premierComposant",
       new Date(), 0)
    new FaceSnap("2emeComposant", "2emeComposant", new
       Date(), 10)
    new FaceSnap("3emeComposant", "3emeComposant", new
       Date(), 100)
  ];
}
```

4.2 Utilisation d'un service :

En utilisant la propriété injectable, on peut l'utiliser dans les différentes classes en l'injectant dans les constructeurs :

```

  Export class SingleFaceSnap{
   constructor(private snapsService: FaceSnapsService)
  }
```

Dans notre exemple, notre service possède des méthodes afin de trouver via un id un snap :

```
    @Injectable({
  providedIn: 'root'
})
export class FaceSnapsService {
  private faceSnaps: FaceSnap[] = [
    new FaceSnap("premierComposant", "premierComposant",
       new Date(), 0)
    new FaceSnap("2emeComposant", "2emeComposant", new
       Date(), 10)
    new FaceSnap("3emeComposant", "3emeComposant", new
       Date(), 100)
  ];

    getSnapFaces(): FaceSnap[] {
      return [...this.faceSnaps] // Pour retourner un
         tableau independant meme si il a les memes
         references
  }

  getSnapById(id: string): FaceSnap {
    const trouverSnap = this.faceSnaps.find(FaceSnap =>
       FaceSnap.id === id );
    if (!trouverSnap) {
      throw new Error("No snap found with ID " + id);
    }else{
      return trouverSnap;
    }
  }

  snapFaceById(faceSnapId: string, snapType: SnapType) {
      const trouverSnap = this.getSnapById(faceSnapId);
      trouverSnap.snap(snapType);
    }
}
```

Afin d'utiliser la méthode getSnapById, on va simplement injecté le service dans la classe ou l'on veut l'utiliser, puis l'appeler :

```

  export class SingleFaceSnap implements OnInit{

  constructor(private snapsService: FaceSnapsService) {
  }

  faceSnap!: FaceSnap;

  ngOnInit(): void {
    this.faceSnap = this.snapsService.getSnapById(
       faceSnapId);
```

```
  }

  }
```

5 Navigation et routes :

5.1 Définition des routes :

Dans le fichier routes de l'application, on retrouve un tableau de Routes. On va y insérer toutes les routes de notre application :

```
export const routes: Routes = [
  {path: 'facesnaps', component: FaceSnapList}, // route qui va afficher le contenu de FaceSnapList
  {path: '', component: HubPage}, // route vide, donc le hub
];
```

Ensuite, dans notre app.ts, on va pouvoir importer "RouterOutlet". Grâce à ceci, il sera possible dans le HTML de définir, au lieu d'un composant précis, l'objet qui se trouve à l'emplacement de la route dans l'url :

```
<app-header/>
<router-outlet/> // Le composant trouve a cette route
```

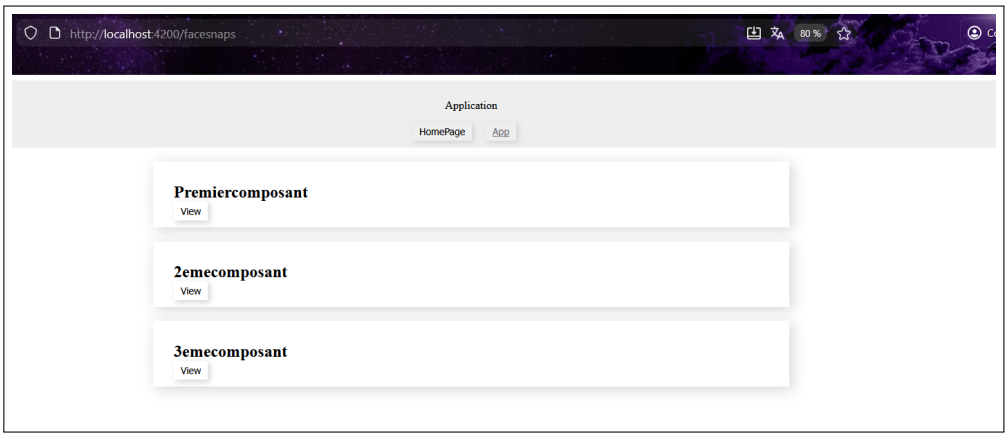
5.2 Naviger entre les composants :

Pour naviguer, il suffit d'utiliser routerLink (qu'on oublie pas d'importer dans le .ts de la où on l'utilise). Par exemple, avec un composant header :

```
<header>
  <p>Application</p>
  <nav>
    <button routerLink="" routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}">HomePage</button>
    <button routerLink="facesnaps" routerLinkActive="active">App</button>
  </nav>
</header>
```

Ici, routerLinkActive sert à donné la classe active à l'élément dont la route est celle selectionnée.

Maintenant, on peut naviguer dans la pages sans problème entre le hub et l'app :



5.3 Accéder à un élément en particulier :

Il est possible dans les routes de définir une variable qui changera :

```
1
2 export const routes: Routes = [
3   // La route avec l'id du snap afficher
4   {path: 'facesnaps/:id', component: SingleFaceSnap},
5   {path: 'facesnaps', component: FaceSnapList},
6   {path: '', component: HubPage},
7 ];
```

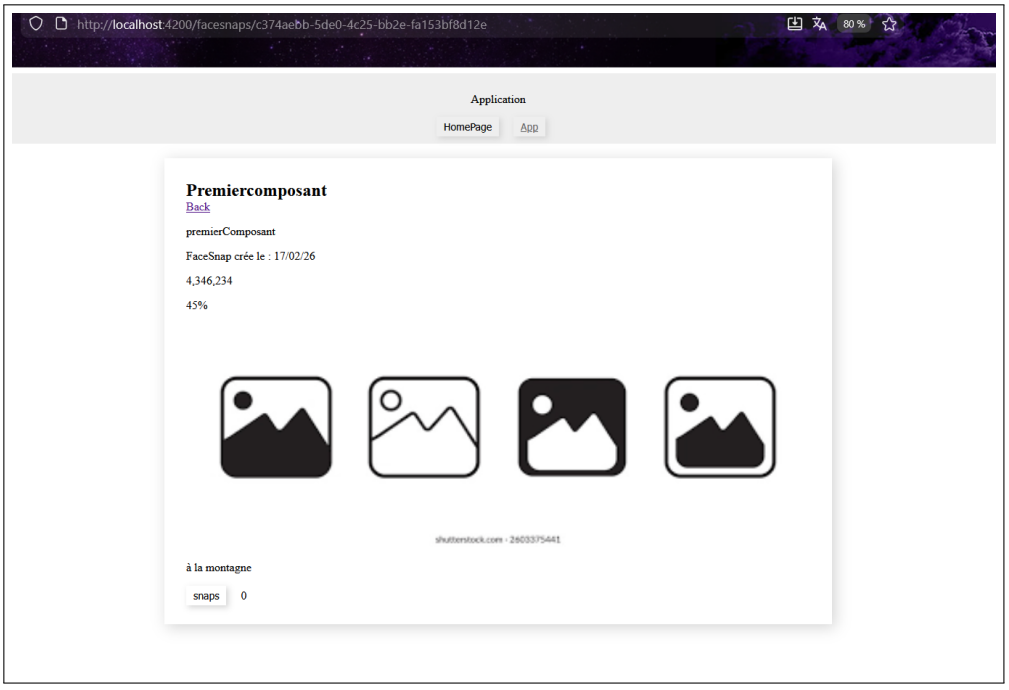
Grâce aux méthodes pour récupérer un snap via un id, on peut dans une classe qui reprends nos snap (copié collé de Comp1) faire :

```
1
2 constructor(private snapsService: FaceSnapsService,
3   private route: ActivatedRoute) {
4   }
5
6   faceSnap!: FaceSnap;
7
8   ngOnInit(): void {
9     const faceSnapId = this.route.snapshot.params['id'];
        // Recup le face snap qui correspond a l'id dans
        l'url
10    this.faceSnap = this.snapsService.getSnapById(
        faceSnapId); // On va recup le faceSnap qui
        correspond a cette id pour l'afficher
11  }
```

Il ne manque plus qu'un moyen de mettre l'id du face snap dans l'url, j'utilise un bouton lié par un événement à cette méthode dans Comp1 :

```
1
2  constructor(private router: Router){}
3
4    @Input() faceSnap!: FaceSnap; // Accepte une valeur qui
         vient du parent
5
6    protected onview() {
7     this.router.navigateByUrl('facesnaps/'+this.faceSnap.
         id); // Methode pour inclure l'id choisi dans l'
         url
8    }
```

On peut maintenant choisir quel composant regarder en cliquant sur le bouton view :



6 Entrées des utilisateurs et dynamisme :

6.1 Entrée simple :

On veut par exemple pouvoir changer dynamiquement le nom d'un composant, pour cela, on utilise ngModel sur l'attribut que l'on veut changer :

```
<label for="choixTitre">
    Titre choisi :
    <input id="choix" type="text" [(ngModel)]="faceSnap.
       title">
  </label>
```

Cela peut également permettre d'ajouter des tris :

```
  <input type = "number" placeholder="Filtrer␣par␣snaps"
     #filter>
  <button (click)="filtrerRes(Number.parseInt(filter.
     value))">Filtrer</button>
  <br>
  <input type = "text" placeholder="Filtrer␣par␣nom" #
     filterNom>
  <button (click)="filtrerResNom($event,␣filterNom.value)
     ">Filtrer</button>
```

Ces tris ont un "filter", un élément qui représente le contenu des input qu'on va pouvoir passer en paramètre de nos fonctions de tris :

```

filtrerRes(nb: number) {
    if(!nb){
      this.filteredfacesSnap = this.facesSnap;
    }else{
      this.filteredfacesSnap = this.faceSnapsService.
         getSnapFaces().filter(elt => elt.snaps > nb);
    }
  }


  filtrerResNom(event: Event, value: string) {
    event.stopPropagation();
    if(!value || value === '') {
      this.filteredfacesSnap = this.facesSnap;
    }else{
      this.filteredfacesSnap = this.faceSnapsService.
         getSnapFaces().filter(elt => elt.title.
         toLowerCase().includes(value.toLowerCase()));
    }
  }
```

6.2 Entrée via un forms :

!!! Tout d'abbord il faut faire attention, un form refresh la page, si on l'utilise pour un tri par exemple, le tri ne se fera que pendant 1 secondes puis sera rafraichi.

J'ai personnelement utilisé les forms pour l'authentification d'un utilisateur. On commence par définir le form dans le ts du composant (ici, hub-page.ts) :

```
1
2  export class HubPage {
3    profileForm = new FormGroup({
4        name: new FormControl('', Validators.required),
5        email: new FormControl('', [Validators.required,
             Validators.email]),
6      })
7  }
```

Les validators servent à valider les données. Il en existe de plusieurs types. On peut ensuite définir un form avec le même nom que l'attribut :

```
1
2  <form [formGroup]="profileForm" (ngSubmit)="handleSubmit
       ()">
3      <input type="text" formControlName="name" />
4      <input type="email" formControlName="email" />
5      <button type="submit" [disabled]="!profileForm.valid"
           >Continuer vers Snapface</button>
6    </form>
```

Ce form va utiliser ngSubmit, qui va envoyer les informations à la méthode entre guillemet. Dans cette méthode, on pourra accéder aux éléments du form avec nomForm.value.objetVoulu :

```
1
2  public handleSubmit(): void {
3      const email = this.profileForm.value.email;
4      this.userService.getUsers().subscribe(users => { //
           Je subscribe pour voir les changements
5        const present = users.some(u => u.email === email);
             // On check juste la presence
6
7        if (!present) {
8          throw new Error("L'utilisateur n'est pas dans la
             base");
9        } else {
10         this.onContinue();
11       }
12     });
13   }
```

7 Objets observables :

Par convention, on les écrits : objet$ = of(values)

Un observable commence à publier des valeurs uniquement si quelque chose est abonné à lui (méthode subscribe()). Exemple concret :

```

const numbers$ = of(1, 2, 3); // simple observable that
    emits three values
```

```

numbers$.subscribe(
  // Fontion qu'on execute lors de la lecture
  value => console.log('Observable emitted the next value
      : ' + value)
);
```

Pour utilser le constructeur d'observables il faut 4 choses :

Premièrement, une fonction qu'utiliera le constructeur d'observable :

```

// This function runs when subscribe() is called
function sequenceSubscriber(observer: Observer<number>) {
  // synchronously deliver 1, 2, and 3, then completes
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
}
```

Ensuite, on construit l'objet observable :

```
const sequence = new Observable(sequenceSubscriber);
```

Enfin, l'objet subscription et la méthode qui va dérouler utiliser l'objet observable :

```

const subscription = numbers$.subscribe({
  next: value => console.log('Observable emitted the next
      value: ' + value),
  error: err => console.error('Observable emitted an
      error: ' + err),
  complete: () => console.log('Observable emitted the
      complete notification') // on est automatiquement
      unsubscribe apres
})
```

Exemple pour insérer le nom dynamique d'une page :

```
ngOnInit() {
    this.afficheBvn = "";
    this.obsBienvenu$ = new Observable<string>(observer
       => {
      const message = "HomePage";
      for(let i = 0; i < message.length; i++) {
        setTimeout(() => {
          observer.next(message[i]);
        },(i+1)*100)
      }
      setTimeout(() =>{
        observer.complete();
      },(message.length +1)*500)
    })

    const affichage = {
      next: (value: string) => {
        this.afficheBvn += value;
        this.cd.detectChanges();
      }
    }

    const subscriber = this.obsBienvenu$.subscribe(
       affichage);

  }
```

8 Commandes utiles :

8.1 Commande "Input"

On peut utilser input de la manière suivante :

```

class User {
  occupation = input<string>();
}
```

Puis dans le html du composant :

```

<app-user occupation="Angular␣Developer"></app-user>
```

8.2 Commande d'affichage en différé :

Pour afficher des éléments en différé, on dispose de plusieurs commandes :

```

@defer {
        <article-comments />
      } @placeholder (minimum 1s) {
        <p>Placeholder for comments</p>
      } @loading (minimum 1s; after 500ms) {
        <p>Loading comments...</p>
      } @error {
        <p>Failed to load comments</p>
      }
```

8.3 Pipes (formatage) :

L'utilisation d'un pipe se fait en important la bonne libraire dans le .ts, puis, dans le .HTML, on peut l'utiliser de la manière suivante :

```
<h2>{{ faceSnap.title | titlecase }}</h2>
```

Il existe plusieurs pipes tels que :

- titlecase (Majuscule A Chaque Mot)
- percentPipe (Transforme un nombre en pourcentage)
- UpperCasePipe/LowerCasePipe (Full maj ou full min)
- DecimalPipe (Transforme un nombre décimal)
- DatePipe (Formate une date selon le format choisi)

Par exemple :

```
<p>FaceSnap cree le : {{faceSnap.createdAt |date : 'dd/MM
    /yy'}}</p>
<p>{{ 4346234.36 | number:'1.0-0'}}</p>
<p>{{ 0.4536 | percent}}</p>
```

...

Il est aussi possible de crée son propre pipe, on crée un nouveau fichier pipe.ts de la manière suivante :

```

import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'Pipetest',
})
export class StarPipe implements PipeTransform {
  transform(value: string): string {
    return `(etoile) ${value} (etoile)`;
  }
}
```

Puis de l'utiliser :

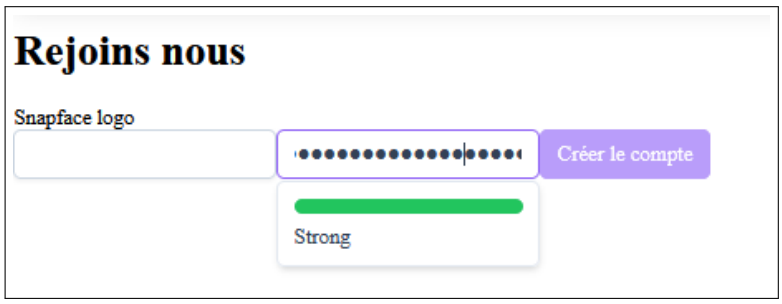
```
<h2>{{ faceSnap.title | titlecase | Pipetest}}</h2>
```

Ce qui donnera :

9 Librairie PrimeNG :

9.1 Qu'est-ce que primeNG :

PrimeNG est une bibliothèque qui permet de limiter le css dans les classes tout en ayant un rendu propre est travaillé à l'aide de preset déjà fais. Voici ce qu'on peut optenir d'une page sans AUCUN css :



9.2 Mise en place de PrimeNG :

Dans un premier temps, il faut installer sur la machine à l'aide de npm les prérequis de primeNG :

```
1 npm install primeng @primeuix/themes
```

Ensuite, on peut commencer la configuration. Le premier fichier à modifier sera le fichier config de l'application. Il faudra y introduire PrimeNg, un thème et des options :

```
1
2  import {providePrimeNG} from 'primeng/config';
3  import Aura from '@primeuix/themes/aura';
4  import Nora from '@primeuix/themes/nora';
5  import Lara from '@primeuix/themes/lara';
6  import Material from '@primeuix/themes/material';
7
8  export const appConfig: ApplicationConfig = {
9
10   providers: [
11     providePrimeNG({
12       theme:{
13         preset: myPreset,
14         options: {
15           prefix: 'p',
16           darkModeSelector: '.my-app-dark',
17           cssLayer: false
18         }
19       },
20       ripple: true,
21       csp:{
22         nonce: '...'
23       },
24     })
25   ]
26 };
```

On peut ensuite définir son propre preset en le faisant dériver d'un éxistant, c'est la meilleure manière de personnaliser son application. Pour se faire, on crée un nouveau fichier myPreset :

```
  const MyPreset = definePreset(Aura, {
  semantic: {
    colorScheme: {
      light: {
        surface: { // Ecritures ...
          50: '{slate.50}',
          100: '{slate.100}',
          200: '{slate.200}',
          300: '{slate.300}',
          400: '{slate.400}',
          500: '{slate.500}',
          600: '{slate.600}',
          700: '{slate.700}',
          800: '{slate.800}',
          900: '{slate.900}',
          950: '{slate.950}'},
        primary:{ // Boutons, input...
          color: '{violet.500}',
          inverseColor: '#ffffff',
          hoverColor: '{violet.700}',
          activeColor: '{violet.800}'},
        formField:{
          hoverBorderColor: '{primary.color}',
        }
      },
      dark: {
      surface: {
        50: '{zinc.50}',
        100: '{zinc.100}',
        200: '{zinc.200}',
        300: '{zinc.300}',
        400: '{zinc.400}',
        500: '{zinc.500}',
        600: '{zinc.600}',
        700: '{zinc.700}',
        800: '{zinc.800}',
        900: '{zinc.900}',
        950: '{zinc.950}'},
        primary:{
          color: '{slate.300}',
          inverseColor: '#ffffff',
          hoverColor: '{slate.500}',
          activeColor: '{slate.600}'}
      },
      formField:{
        hoverBorderColor: '{primary.color}'}
    },}});
export default MyPreset;
```

Une fois se fichier crée, il suffit de remplacer le preset dans le fichier config :

import myPreset from './MyPreset';

```
export const appConfig: ApplicationConfig = {

  providers: [
    provideBrowserGlobalErrorListeners(),
    provideRouter(routes),
    provideAnimationsAsync(),
    providePrimeNG({
      theme:{
        preset: myPreset,
        options: {
          prefix: 'p',
          darkModeSelector: '.my-app-dark',
          cssLayer: false
        }
      },
    }
}
```

Une fois le preset mis en place, il s'applique sur les balises prime ng. Puisque j'ai défini le préfixe comme étant p, on peut retrouver des balises tels que <p-button>.

9.3 Différentes utilisations :

9.3.1 DarkMode

Il est assez simple d'ajouter par exemple un mode sombre à notre application, j'ai défini dans mon préset les couleurs avec l'option dark, dans mon fichier config, j'ai ajouter un selector pour le dark mode. Une fois réalisé, il suffit de 2 choses : Un moyen d'activé le dark mode (ici un élément switch de primeNG :)

```
  <p-toggleswitch [(ngModel)]="modeSombre" (click)="
     toggleDarkMode()" [dt]="switchLight"/>
```

Ainsi que la méthode toggleDarkMode qui suit :

```
  protected toggleDarkMode() {
    const element = document.querySelector('html');
    if(element !== null){
      element.classList.toggle('my-app-dark');
    }
  }
```

9.3.2 Authentification

Avec PrimeNg, il est assez simple de modéliser un mot de passe ainsi que de vérifier son niveau de fiabilité. On crée simplement un mdp à l'aide d'une balise a laquelle on va ajouter des options :

```
<p-password promptLabel="Choose␣a␣password" [feedback]="
    false" [toggleMask]="true" formControlName="pwd"></p-
    password>
```

Ici, un mot de passe pour se connecter, il n'y a pas de feedback et on peut voir le mdp à l'aide d'un oeil. Mais pour créer un compte ça sera plutot :

```
<p-password promptLabel="Choose␣a␣password"
    formControlName="pwd"/>
```

Ce qui donnera :

9.3.3 Autocomplete des input :

Pour l'autocomplete, il suffit de définir la liste des items qu'on recherche et d'initialiser une méthode de recherche. Elle se présente sous cette forme :

```
search(event: any) {
  const query = event.query.toLowerCase(); // recup le contenue

  this.items = this.facesSnap
    .map(face => face.title) // Remplace tous les face par leur titre
    .filter(title => title.toLowerCase().includes(query)); // filtre
}
```

Il suffit maintenant dans le html d'utiliser une balise <p-autocomplete/>

```
<p-autoComplete [(ngModel)]="filtrerNom" [dropdown]="true" [suggestions]="items" (completeMethod)="search($event)"  inputId="labelF" #filterName/>
```

9.3.4 Icones primeNG :

Pour commencer, il faut installer les packages nécessaires :

```
npm install primeicons
```

Dans le fichier angular.json, dans notre projet dans le tableau des "styles", si ce n'est pas déja fais, on ajoute :

```
"node_modules/primeicons/primeicons.css",
```

Ensuite, il sera possible de les utiliser. Pour ce faire, il suffit soit de définir un icone dans un objet :

```
<p-button
    [icon]="VueTable ? 'pi pi-list' : 'pi pi-info-circle'"
    (click)="VueTable = !VueTable"
    [rounded]="true"
    [text]="true">
</p-button>
```

Ou alors de définir une zone d'icone (en général devant un input) :

```
<p-iconfield>
   <p-inputicon class="pi␣pi-lock"></p-inputicon>
 <p-password promptLabel="Choose␣a␣password" [feedback
    ]="false" [toggleMask]="true" formControlName="
    pwd"></p-password>
 </p-iconfield>
```

9.3.5 Boutons split :

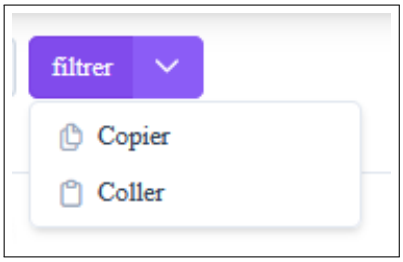
Pour déclarer un split bouton, on utilise une balise prévu pour. Cette balise se comporte comme un bouton normal, mais on peut ya joter une liste d'item (qui seront les autres actions possible) :

```
<p-splitbutton label= "filtrer" (click)="
   filtrerResNom($event,␣filterName.value)" [model]=
   "items2">Filtrer</p-splitbutton>
```

Cette liste de type MenuItem[] sera remplie de cette mannière :

```
this.items2=[
    {
      label : "Copier",
      icon: "pi␣pi-copy",
      command: () => {
        this.copierNom();
      }
    },
    {
      label: "Coller",
      icon: "pi␣pi-clipboard",
      command: ()=>{
        this.collerNom();
      }
    }
  ]
```

Ce bouton se présente sous cette forme :



9.3.6 Copy Paste :

C'est une option assez simple à mettre en place et assez utile. J'ai écrit deux méthodes que j'ai assigné à mon splitBouton, les voici :

```
copierNom(){
    if(!this.filtrerNom)
      return;
    else{
      navigator.clipboard.writeText(this.filtrerNom); // Ecrit dans le presse papier
    }
  }

  collerNom() {
    navigator.clipboard.readText().then(
      text => {
        this.filtrerNom = text;   // met a jour le ngModel aussi
      });
  }
```

9.4 Tableaux :

Les tableaux sont une partie importante de l'affichage, leur mise en place n'est pas complexe grâce à primeNG. Dans un premier temps, on définit un tableau dans notre code HTML :

```
<p-table [value]="filteredfacesSnap" dataKey="id"> //
    Liste sur lequel repose le tableau a preciser
    <ng-template pTemplate="header">
      <tr>
        <th>Titre</th>
        <th>Image</th>
        <th>Prix</th>
        <th>Date</th>
        <th>Rating</th>
      </tr>
    </ng-template>
    <ng-template pTemplate="body" let-product>
      <tr>
        <td>{{ product.title }}</td>
        <td><img [src]="product.url" class="table-img"></
            td>
        <td>{{ product.prix | dolarPipe }}</td>
        <td>{{ product.createdAt | date: 'dd/MM/yy' }}</
            td>
        <td><p-rating [(ngModel)]="product.rating" [
            readonly]="true"></p-rating></td>
        <td><p-button icon="pi␣pi-search" (click)="onview
            (product.id)"></p-button></td>
      </tr>
    </ng-template>
  </p-table>
```

Ce tableau est conçu de la manière la plus simple possible. On peut remarquer que j'y ai ajouté une balise <p-rating> qui va retranscrire un nombre en rating (ex : 4 = 4 étoiles pleines). J'ai également ajoué un bouton pour inspecter le produit

9.4.1 Mise en ordre automatiques des colonnes :

Il est possible de faire un tri croissant ou décroissant sur chaque colonne. Il suffit d'ajouter quelques arguments : Dans l'entête du tableau on ajoute un tri de base au lancement de l'application :

```
<p-table [value]="filteredfacesSnap" sortField="title">
```

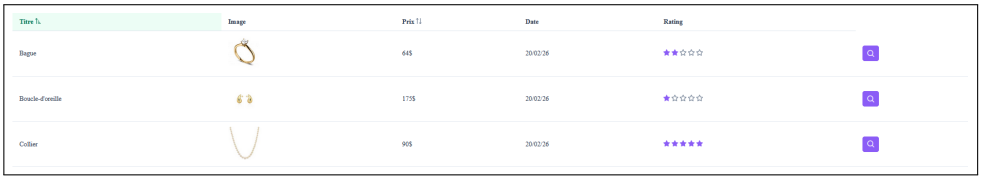
Ensuite, on peut déterminer les colonnes sur lesquels on pourra éffectuer un tris :

```
<ng-template pTemplate="header">
     <tr>
       <th pSortableColumn="title"> // Sort sur la
          colonne du titre

         Titre
         <p-sortIcon field="title"/> // Icone
       </th>
       <th>Image</th>
       <th pSortableColumn="prix"> // Sort sur la
          colonne du titre

         Prix
         <p-sortIcon field="prix"/> // Icone
         </th>
       <th>Date</th>
       <th>Rating</th>
     </tr>
   </ng-template>
```

Notre tableau est maintenant propre et proffessionel, voici le rendu de ces quelques lignes :



9.5 Paginator :

Il existe plusieurs types de paginator, ceux sur les tableaux et les autres. Je vais donc diviser cette partie en 2 catégories

9.5.1 Paginator dans les tableaux :

Dans un tableau, il est très simple d'ajouter un paginator. On ajoute simplement dans l'entête :

```
 // Avec rows le nombre de ligne sur chaque page
<p-table [value]="filteredfacesSnap" dataKey="id" [
   paginator]="true" [rows]="3" sortField="title">
```

9.5.2 Paginator dans un cas général :

Lorsque l'on implémente un paginator en général, c'est un petit peu plus complexe. Dans un premier temps, on le défini :

```
<p-paginator (onPageChange)="onPageChange($event)" [first
    ]="0" [rows]="3" [totalRecords]="filteredfacesSnap.
    length"></p-paginator>
```

On devra ensuite dans le ts écrire :

- La méthode appelé par onPageChange
- L'attribut first
- L'attribut rows
- Une méthode qui découpe notre liste avec le nombre d'item sur chaque page

On déclare :

```
first: number = 0;
rows: number = 3;
```

Pour la méthode onPageChange :

```
onPageChange(event: any){
    this.first = event.first;
    this.rows = event.rows;
  }
```

Et enfin la méthode de découpe :

```

 getPaginatedSnaps() {
    return this.filteredfacesSnap.slice(this.first, this.
        first + this.rows);
  }
```

Il ne reste plus qu'a afficher nos éléments selon cette liste découpée :

```
@for(f of getPaginatedSnaps(); track f.id){
    <app-comp1 [faceSnap]="f" />

  }
  <p-paginator (onPageChange)="onPageChange($event)" [
     first]="0" [rows]="3" [totalRecords]="
     filteredfacesSnap.length"></p-paginator>
```

On peut maintenant naviguer dans notre page comme dans un vrai site de E-boutique :