Mon premier projet Angular

Framework front-end

Reprenons ensemble le projet Soif200 et voyons comment l'intégrer sur Angular

Table des matières

Commencer :

Premièrement, voici la maquette : https://res03.sites.3wa.io/assets/files/inte/j17/Maquette-exercice-3.png

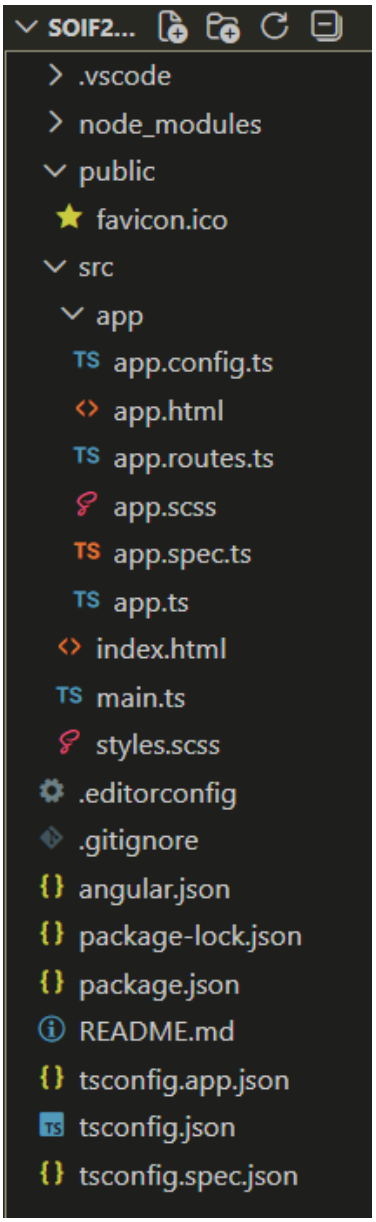
Nous allons nous placer dans la répertoire **wsldev/projects** sur **Ubuntu**, puis nous allons utiliser le CLI Angular pour générer la base notre projet :

```
ng new soif200
```

On choisit le moteur SASS, pas de Server-Side Rendering, une appli « zoneless », **PAS** D'IA, puis, une fois notre dossier créé, déplaçons nous dedans avec

```
cd soif200
```

Nous y trouvons cette arborescence :



Afin d'être tranquille, on peut commencer par ajouter cette propriété dans la clé « compilerOptions » du fichier **tsconfig .json** :

```
"strictPropertyInitialization": false,
```

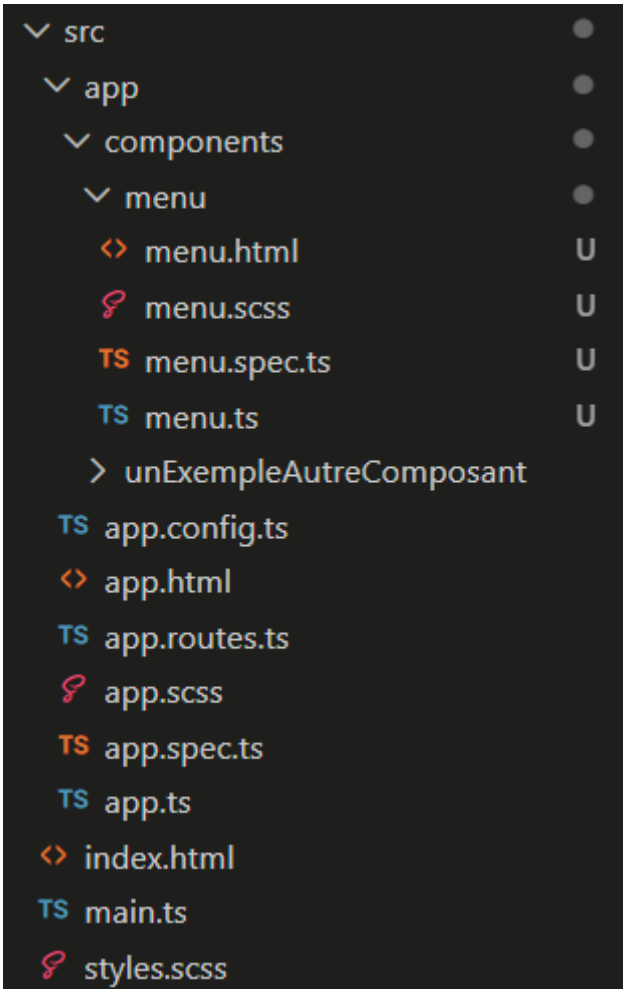
Cette propriété va nous permette de créer des variables ou des propriétés dans nos classes sans devoir leur assigner de valeurs immédiatement.

Créer un composant

Nous allons commencer par créer le premier **component** de notre page : le menu. Pour cela, nous allons demander au CLI Angular de nous générer le composant en question :

```
ng generate component menu
```

Angular CLI va donc créer un dossier **menu** dans **src/app** contenant 4 fichier, un html, un scss, un ts, et un fichier de spec pour les tests. Afin de garder de la clarté dans notre code, nous allons créer un dossier components dans **src/app/**, puis glisser les futurs components et le menu dedans. On se retrouve donc avec cette arborescence :



Nous allons maintenant **inclure** ce **component** dans le composant racine, c'est-à-dire le composant **App**. Pour cela, rendons-nous dans le fichier app.ts, et ajoutons-y les instructions suivantes :

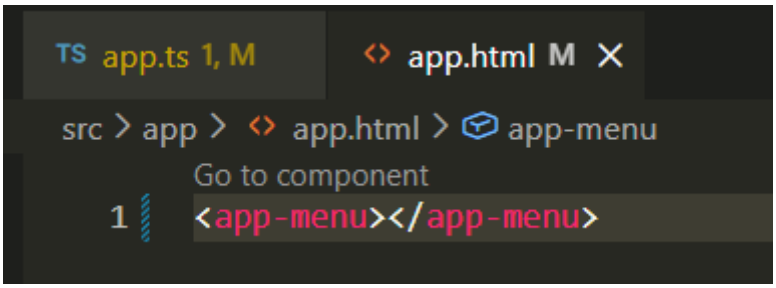
```
import { Menu } from './components/menu/menu';
```

Et enfin, dans la clé « imports » du décorateur **@Component()**, nous allons ajouter la classe **Menu** que l'on vient d'importer.

```
@Component({
  selector: 'app-root',
  imports: [Menu],
  templateUrl: './app.html',
  styleUrl: './app.scss'
})
```

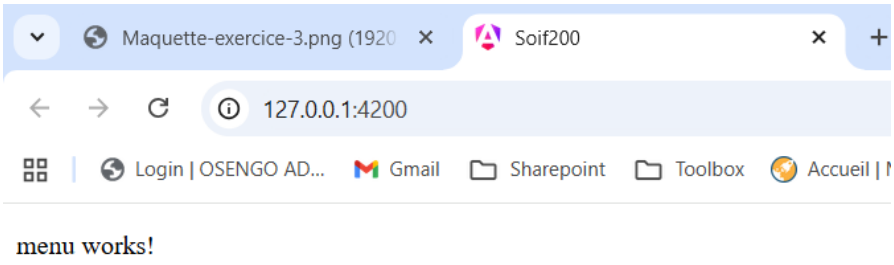
Maintenant que le composant racine a toutes les données nécessaires pour utiliser le component **Menu**, il ne nous reste plus qu'à lui dire où l'implémenter dans son **template**.

Pour cela nous allons vider le contenu généré dans le fichier app.html, et y mettre à la place la balise dans laquelle va se placer notre **component Menu**. Si vous avez oublié le nom de cette balise, retournez voir le fichier **menu.ts**.



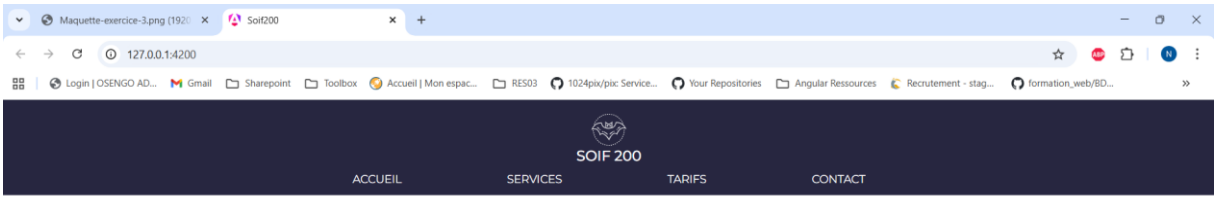
Vous pouvez maintenant exécuter la commande **ng serve** dans le terminal, puis accéder à http://localhost:4200

Vous devriez voir ceci :



Le contenu du fichier **src/app/components/menu/menu.html** s'affiche bien à l'emplacement qu'on lui a donné dans le composant racine. Nous avons alors simplement à modifier ce fichier **menu.html** afin d'y inclure le vrai HTML que nous voulons, à savoir le menu de Soif200. Pour le moment, les données ne sont pas dynamiques, c'est normal, nous verrons cela une fois qu'on aura tout intégré.

Résultat attendu :



Recommencez depuis « Créez un composant » avec le header, les services, les tarifs, le module de contact et le footer.

Dynamiser son contenu

Une fois qu'on a tout notre site découpé en composant, avec le CSS qui convient, nous allons vouloir dynamiser son contenu plutôt que de tout écrire à la main dans chaque fichier.

Pour cela, nous avons deux options :

- Lire les données dans un fichier json stocké localement
- Lire les données renvoyées par une API externe.

Commençons par le plus simple :

1. Lire les données depuis un fichier json

Nous allons d'abord placer le fichier en question dans un répertoire **/assets**, au même niveau que **/app**

Puis, dans **app.ts** nous allons importer ces données comme ceci :

```
import * as jsonData from './assets/data.json';
```

Cet objet **jsonData** que nous venons de créer va contenir toutes les propriétés que nous avons dans le fichier, puisqu'il est une **traduction** de ce JSON en **objet**.

Nous allons donc pouvoir simplement assigner le contenu de cet objet à une ou plusieurs variables, par exemple, si l'on veut récupérer les liens de menu, on va alors faire comme ceci, dans les propriétés de la classe App :

```
  dataMenu: any = jsonData.menu;
```

Comment passer ces liens de menu dans le composant menu si on les récupère dans le composant racine ? La première chose importante à comprendre est qu'il n'est pas obligatoire de passer par le composant racine pour récupérer ces données, on peut très bien les récupérer directement dans le composant menu. Toutefois, ici, on fait le choix de **centraliser** la récupération des données car le fichier JSON contient les données de plusieurs composants.

Dans le composant **menu.ts**, nous allons préparer une propriété **liens** qui sera créer avec le décorateur **@Input()**. Attention à ne pas oublier d'importer **Input** depuis @angular/core :

```
  @Input() liens:any = [];
```

Pour remplir cette variable avec les données que le composant racine est allé chercher dans le fichier JSON, nous n'avons plus qu'à utiliser l'attribute binding dans le template :

```
<app-menu [liens]="dataMenu"></app-menu>
```

*Il faut lire cette ligne comme ceci : la propriété **liens** du composant **menu** va prendre la valeur de la propriété **dataMenu** du composant **racine***

2. Lire les données depuis une API

Cette méthode sous-entend l'implémentation d'une API au préalable. Dans ces exemples, j'ai développé une API PHP qui me renvoie toutes les données sous la forme d'une chaîne de caractère JSON.

Nous allons créer ce que l'on appelle un **Service**, c'est-à-dire une brique de code fonctionnelle dont le but est de répondre à un besoin du **développeur** et non pas de l'utilisateur. Pour cela on peut faire

```
ng generate service data-service
```

Cela va nous créer un fichier DataService dans lequel nous allons pouvoir développer notre outil de récupération des données sur l'API :

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

@Injectable({
  providedIn: 'root',
})
export class DataService {

}
```

Nous allons avoir besoin d'importer la classe HttpClient pour pouvoir lui demander d'exécuter des requêtes http.

Pour cela nous allons ajouter cet import dans le fichier **app.config.ts**

```
import { provideHttpClient } from '@angular/common/http';
```

Puis utiliser cette fonction dans le tableau des imports du même fichier, comme ceci :

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideZonelessChangeDetection(),

    provideRouter(routes),
    provideHttpClient()
  ]
};
```

Maintenant que la classe HttpClient est importée et **Injectable**, il faut que l'on puisse l'utiliser dans le fichier **app.ts**

Pour cela nous allons simplement ajouter cette ligne dans **app.ts** :

```
import { HttpClient } from '@angular/common/http';
```

Puis déclarer une propriété « monClientHttp » de type HttpClient dans notre classe DataService :

```
  constructor(private monClientHttp: HttpClient){}
```

On déclare la propriété directement dans la signature du constructeur, ça nous permet de gagner un peu de place et c'est plus rapide aussi ☺

Maintenant que notre service est prêt à fonctionner, soyons un peu plus précis sur ce que l'on veut qu'il fasse et créons une méthode **getJsonData()** :

```
  getJsonData(){
    return this.monClientHttp.get('http://exercice-api.loc/soif200.php', {
      headers: {'Authorization' : 'VBnAzKpOLlf5DZSNpNuXJmvg4' },
      responseType: 'text'
    });
  }
```

Cette méthode utilise la propriété monClientHttp de type **HttpClient**, sur laquelle elle appelle la méthode **.get()** en lui passant l'url, les headers, ainsi que la directive « responseType : 'text' »

Et enfin, cette méthode utilise un **return** pour renvoyer le résultat de cette requête http

Retournons du côté de **app.ts** et voyons comment intégrer ce service dans notre logique. Premièrement on va l'importer :

```
// Import de mon service qui va aller chercher les données sur mon API
import { DataService } from './data-service';
```

Puis nous allons déclarer une propriété **data** de type DataService dans le constructeur directement, comme plus haut :

```
constructor(private data:DataService) {}
```

Ensuite, nous allons importer un autre élément du moteur d'Angular, le **ChangeDetectorRef**. Il faut donc l'importer :

```
// Imports dans le moteur d'angular
import { Component, ChangeDetectorRef } from '@angular/core';
```

Puis on va ajouter une propriété **cdr** de type ChangeDetectorRef dans notre constructeur :

```
  constructor(private data: DataService, private cdr: ChangeDetectorRef) {}
```

Dans ce constructeur, nous allons appeler la méthode **getJsonData()** que nous avons défini dans **DataService**, et nous allons appeler la méthode **subscribe()** sur sa valeur de retour : une réponse http

que l'on considère **« Observable »**. Un peu comme un **eventListener**, un **Observable** va réagir lorsqu'il a quelque chose de nouveau à afficher.

On parse alors les données JSON de la variable « response » pour obtenir un tableau, et on assigne chacune de ces clés aux propriétés de ma classe.

Enfin, on appelle le **ChangeDetectorRef** pour qu'il détecte les changements sur les données et donc rafraîchisse les composants auxquels on passe ces données avec **this.cdr.detectChanges()**

```
// On appelle la méthode getJsonData() de mon service puis on "souscrit" à sa réponse
puisqu'elle est de type Observable (c'est une réponse HTTP)
    this.data.getJsonData().subscribe({
      next: (response) => {
        let result = JSON.parse(response);
        // Assignation des valeurs directement avec la clé du tableau JSON
        // Plutôt qu'un push pour qu'Angular détecte bien le changement.
        this.dataMenu = result.menu;
        this.dataServices = result.services;
        this.dataTarifs = result.tarifs;

        // On déclenche le ChangeDetectorRef à la main pour qu'il relise les données et
détecte les changements.
        this.cdr.detectChanges();
      },
      error: (err) => console.error('Erreur API :', err)
    });
```

Classe complète :

```
export class App {
  title: string = 'soif200';

  dataMenu: any[] = [];
  dataServices: any[] = [];
  dataTarifs: any[] = [];

  constructor(private data: DataService, private cdr: ChangeDetectorRef) {
    this.data.getJsonData().subscribe({
      next: (response) => {
        let result = JSON.parse(response);
        // Assignation des valeurs directement avec la clé du tableau JSON
        // Plutôt qu'un push pour qu'Angular détecte bien le changement.
        this.dataMenu = result.menu;
        this.dataServices = result.services;
        this.dataTarifs = result.tarifs;

        // On déclenche le ChangeDetectorRef à la main pour qu'il relise les
données et détecte les changements.
        this.cdr.detectChanges();
      },
```

```
      // Si on a eu une erreur lors de la requête HTTP, on log l'erreur en
console
      error: (err) => {
        switch(err.status) {
          case 401:
            console.log('Erreur 401 : Token manquant');
            break;
          case 403:
            console.log('Erreur 403 : Accès interdit');
            break;
        }
      }
    });
  }
}
```

3. Et maintenant ?

Nous avons deux façons différentes de passer les données dont nos composants avaient besoin, maintenant nous pouvons les intégrer dans le template du composant lui-même en utilisant soit l'**attribute binding**, soit la **string interpolation**.

Dans notre cas précis, chaque élément récupéré dans le fichier JSON est un **tableau**, nous allons donc utiliser une syntaxe de **boucle**, la boucle **@for**.

La boucle **@for** permet d'itérer sur un tableau et de **suivre** les itérations du tableau. Voyez plutôt :

```
<nav>
    <div class="container text-center">
        <img src="vampire-bat.svg" class="mx-auto"/>
        <h1 class="text-white">SOIF 200</h1>
        <ul class="navbar mb-0 mx-auto d-flex justify-content-between list-unstyled">
<!-- Boucle @for dans laquelle on déclare la variable "lien" en bouclant sur le tableau "liens" -->
<!-- Je rappelle que le tableau "liens" est une proriété de la classe Menu décrite dans menu.ts -->
<!—Puis on ajoute l'instruction track suivi de l'objet et d'une propriété (n'importe laquelle) -->
<!-- cela permet à Angular de garder une trace de chaque élément même si son contenu change, il sera
identifié -->
            @for(lien of liens; track lien.lien) {
                <li class="nav-item" >
                                                                <!-- Attribute Binding ici -->
                    <a class="nav-link text-white text-uppercase" [href]="lien.lien">
                        <!-- String interpolation ici -->
                        {{ lien.titre }}
                    </a>
                </li>
            }
        </ul>
    </div>
</nav>
```

Vous pouvez recommencer ces étapes de dynamisation pour la partie Services et aussi la partie Tarifs. Votre projet devrait alors toucher à sa fin avec un composant Racine dont le template ressemble à ceci :

```
<!-- Chaque composant appelé se voit passer les attributs nécéssaires par le
composant app-root en utilisant l'attribute binding -->
<app-menu [liens]="dataMenu"></app-menu>
<app-header></app-header>
<app-services [services]="dataServices"></app-services>
<app-tarifs [tarifs]="dataTarifs"></app-tarifs>
<app-contact></app-contact>
<app-footer [liens]="dataMenu"></app-footer>
```

À vous de jouer ☺

Un mot sur l'API

Si on doit gérer l'API de notre côté, il se peut que l'on ait besoin de gérer un certain nombre de cas de figures notamment concernant les **CORS POLICY**, c'est-à-dire la Politique de partage de ressources depuis de multiples origines. (Cross Origin Resources Sharing).

Concrètement, cela signifie que l'on doit **autoriser** notre site Angular à interroger l'API. Pour cela, nous allons préciser dans les **headers** de l'API ce qu'il est possible ou non de recevoir. Voyez plutôt :

```
// Envoyer les headers CORS en tête de réponse
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Headers: Authorization");
```

Ici on autorise toutes les origines à venir questionner notre API, et ensuite en deuxième ligne on autorise également la clé « AUTHORIZATION » à être interceptée dans les headers de la requête pour finir dans **$_SERVER**.

Ensuite, nous devons éliminer un problème de taille : les requêtes **preflight**. Il s'agit d'une sorte de pré-requête HTTP que le navigateur va générer avec la méthode **OPTIONS** afin de vérifier s'il est sécurisé d'appeler cette URL. Sauf que lorsque l'API reçoit cette pré-requête, elle ne trouve pas le header Authorization, et du coup elle peut nous renvoyer vers une erreur 403 indésirable.

Pour cela, nous allons simplement nous assurer de toujours renvoyer un code http 200 en cas de requête **preflight**, comme ceci :

```
// Répondre aux requêtes preflight (OPTIONS)
if (isset($_SERVER['REQUEST_METHOD']) && $_SERVER['REQUEST_METHOD'] ===
'OPTIONS') {
    http_response_code(200);
    exit();
}
```

Nous pouvons maintenant nous concentrer sur la gestion des erreurs concernant cet appel à l'API. Nous allons en gérer deux : la 401 et la 403. Pour cela nous allons tester respectivement la présence puis la validité du token passé dans **$_SERVER['HTTP_AUTHORIZATION']**, comme ceci :

```
$token = $_SERVER['HTTP_AUTHORIZATION'];

if (!$token) {
    header('HTTP/1.1 401 Unauthorized');
    echo json_encode(['error' => 'Token manquant']);
    exit();
}

if ($token !== "VBnAzKpOLlf5DZSNpNuXJmvg4") {
    header('HTTP/1.1 403 Forbidden');
    echo json_encode(['error' => 'Token invalide']);
    exit();
}
```

Notez qu'à chaque fois qu'on traite un cas, on conclut notre instruction par **exit()**, ce qui nous permet d'interrompre l'exécution du code.

Enfin, une fois que tous les cas que l'on souhaite sont traités (La requête de préflight, le token manquant, le token erroné…) on peut se concentrer sur ce que l'on veut vraiment faire avec notre API, c'est-à-dire renvoyer les données en que l'on souhaitait à la base :

```php
// Connexion à la base de données
$database = new mysqli("192.168.56.56", "homestead", "secret", "soif200");
mysqli_set_charset($database, "utf8mb4");

$menus = $database->execute_query("SELECT * FROM menu")->fetch_all(MYSQLI_ASSOC);
$services = $database->execute_query("SELECT * FROM service")->fetch_all(MYSQLI_ASSOC);
$tarifs = $database->execute_query("SELECT * FROM tarif")->fetch_all(MYSQLI_ASSOC);

$arrayFinal = array(
    "menu" => $menus,
    "services" => $services,
    "tarifs" => $tarifs
);

echo json_encode($arrayFinal);
```

Nous n'avons maintenant plus qu'à repasser côté Angular pour intercepter ces données et en faire ce que bon nous semble ☺