Utilisation du service

Modification de `cocktail-container.component.ts`

Nous allons modifier le composant conteneur, aussi appelé composant intelligent, qui contient la logique et qui la passe aux composants d'affichage, appelés aussi composant bêtes (on parle d'architecture `smart / dumb components`).

Nous y plaçons le code suivant, que nous allons expliquer :

```
import { Component, OnDestroy, OnInit } from "@angular/core";
import { Subscription } from "rxjs";
import { Cocktail } from
"../shared/interfaces/cocktail.interface";
import { CocktailService } from
"../shared/services/cocktail.service";

@Component({
  selector: "app-cocktail-container",
  templateUrl: "./cocktail-container.component.html",
  styleUrls: ["./cocktail-container.component.scss"]
})
export class CocktailContainerComponent implements OnInit,
OnDestroy {
  public selectedCocktail: Cocktail;
  public cocktails: Cocktail[];
  public subscription: Subscription = new Subscription();

  constructor(private cocktailService: CocktailService) {}

  ngOnInit() {
```

```
    this.subscription.add(
      this.cocktailService.cocktails$.subscribe((cocktails:
Cocktail[]) => {
        this.cocktails = cocktails;
      })
    );
    this.subscription.add(
      this.cocktailService.selectedCocktail$.subscribe(
        (selectedCocktail: Cocktail) => {
          this.selectedCocktail = selectedCocktail;
        }
      )
    );
  }

  public selectCocktail(index: number) {
    this.cocktailService.selectCocktail(index);
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}
```

Nous créons une instance de `Subscription` pour contenir les deux `subscriptions` que nous avons dans ce composant.

```
public subscription: Subscription = new Subscription();
```

Il est important de conserver les références aux `subscriptions` afin de se désinscrire (ou `unsubscribe()`) de celles-ci lorsque le composant est gratuit. Si nous n'effectuons pas ces désinscription, cela peut provoquer des

fuites de mémoire dans l'application.

Une fois que nous avons une `Subscription`, nous pouvons utiliser la méthode `add()` pour ajouter autant de `subscriptions` que nécessaire.

Pour rappel, chaque utilisation de la méthode `subscribe()` retourne une `Subscription`.

```
ngOnInit() {
    this.subscription.add(
      this.cocktailService.cocktails$.subscribe((cocktails:
Cocktail[]) => {
        this.cocktails = cocktails;
      })
    );
    this.subscription.add(
      this.cocktailService.selectedCocktail$.subscribe(
        (selectedCocktail: Cocktail) => {
          this.selectedCocktail = selectedCocktail;
        }
      )
    );
  }
```

Ici nous avons une `subscription` pour recevoir la liste de cocktails et ses mises à jour éventuelle (nous en aurons lorsque nous ajouterons ou éditerons des cocktails).

Nous avons une seconde `subscription` pour recevoir le cocktail sélectionné.

Nous modifions la méthode `selectCocktail()` :

```
public selectCocktail(index: number) {
  this.cocktailService.selectCocktail(index);
}
```

Celle-ci va simplement appeler une méthode sur notre service, car nous voulons que toute la logique de modification de nos cocktails soit à un seul endroit (pour rappel, ce pattern s'appelle `single source of truth`).

Nous n'oublions pas de désinscrire toutes nos `subscriptions` lorsque le composant est détruit dans le `hook ngOnDestroy()` :

```
ngOnDestroy() {
  this.subscription.unsubscribe();
}
```

Modification de `cocktail.service.ts` :

Dans notre service, nous ajoutons un `$` à la fin de chaque `Observable`.

Il s'agit d'une convention `rxjs` qui permet de distinguer en un coup d'oeil si une variable contient un flux (`Observable`, `Subject` etc) ou non :

Pour rappel, les `BehaviorSubject` sont une forme particulière de `Subject` qui retiennent en mémoire la dernière valeur émise.

Les `Subjects` sont eux-mêmes des `Observables` qui permettent un multicast à plusieurs `Observers`.

```
import { Injectable } from "@angular/core";
import { BehaviorSubject } from "rxjs";
import { Cocktail } from "../interfaces/cocktail.interface";

@Injectable({ providedIn: "root" })
export class CocktailService {
  public cocktails$: BehaviorSubject<Cocktail[]> = new
BehaviorSubject([
    // les cocktails ...
  ]);

  public selectedCocktail$: BehaviorSubject<Cocktail> = new
BehaviorSubject(
    this.cocktails$.value[0]
  );

  public selectCocktail(index: number): void {
```

```
    this.selectedCocktail$.next(this.cocktails$.value[index]);
  }


  constructor() {}
}
```

Nous créons également une méthode qui va modifier la valeur du cocktail sélectionné grâce à la méthode `next()`.

La méthode `next()` permet de transmettre une nouvelle valeur aux `Observers` qui ont `subscribe()` à un `Subject`.

Code exécutable

Voici donc où nous en sommes :

https://stackblitz.com/edit/angular-c9-l3