Requête HTTP

Un site conventionnel interagit avec le serveur à chaque chargement de page, ce qui peut lui permettre de faire des tas de vérification, demander des données à la BDD ou autre, mais le problème avec une SPA c'est que les chargements de page, on en a aucun.

Il va donc falloir faire des requêtes http (comme avec fetch en js vanilla) pour obtenir nos informations.

On va importer à la racine de notre projet dans "**app.module.ts**", le module suivant :

```
import { HttpClientModule } from '@angular/common/http';
// Puis dans les imports de @NgModule au dessus de nos propres modules :
HttpClientModule,
```

Créer une fausse API

Jusqu'ici on a récupéré nos informations via notre fichier "**RecetteListe.ts**" mais dans un cas normal, nos informations devraient venir d'une BDD afin que ce soit des informations qui puissent être gardé en mémoire.

On ne va pas apprendre à créer une BDD ici, on va plutôt utiliser une extension Angular qui va nous permettre de simuler cette BDD via une API, installons donc :

```
npm install angular-in-memory-web-api --save-dev
# "--save-dev" indique que cette dépendance ne doit pas se retrouver en production
```

Puis on va créer un nouveau service à la racine de notre projet :

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

Dans celui ci on importera et implémentera l'interface suivante :

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
/* ... */
export class MemoryDataService implements InMemoryDbService
{
// Puis on va créer une méthode :
    createDb()
    {
        const recettes = RECETTES;
        return {recettes};
        // On indique ici que notre fausse BDD sera notre liste de recette
    }
}
```

> ATTENTION, le nom donné dans notre return aura son importance, ici "**recettes**".

On pourra supprimer le constructor et les possibles imports inutile; Enfin retournons dans notre fichier "**app.module.ts**" pour ajouter notre service :

```
// juste en dessous de HttpClientModule et on n'oublie pas les imports
HttpClientInMemoryWebApiModule.forRoot(MemoryDataService, {dataEncapsulation:
false}),
```

L'option est là pour indiquer à notre fausse api de ne pas encapsuler nos réponses dans un objet data.

READ modification

Pour la suite on va pouvoir travailler avec notre fichier "**recette.service.ts**" dans lequel on va venir modifier "**getRecetteList()**". Jusqu'ici on retournais un objet qu'on avait dans nos fichiers, c'était une requête *synchrone*, c'était instantané, sauf que maintenant on veut simuler une requête à un serveur. Cela peut prendre un peu de temps, on va avoir besoin d'une requête *asynchrone*.

Pour cela la première étape va être d'importer "**HttpClient**" depuis Angular :

```
import { HttpClient } from '@angular/common/http';
/* ... */
constructor(private http: HttpClient){}
```

Puis on va modifier ce que retourne notre méthode :

```
import { Observable } from 'rxjs';
getRecetteList():Observable<Recette[]>{}
```

Un "**Observable**" c'est un objet qui va changer dans le temps et que l'on va pouvoir observer. Il va nous permettre de savoir quand est ce qu'on a récupéré les données qu'on attend.

Les "**Observables**" sont des objets venant de la bibliothèque "**rxjs**". Une bibliothèque spécialisé dans les requêtes asynchrones.

Pour récupérer cet "Observable", on va changer notre return :

```
return this.http.get<Recette[]>("api/recettes").pipe()
```

"**.get**" prend en paramètre l'url de notre api, comme pour un fetch. et lance une requête *HTTP*.

Notre fausse API, aura pour adresse, "**api/**" suivi du nom donné dans le return de notre fausse BDD. ici "**recettes**".

"**.get**" nous retournera un **Observable**.

- "**<Recette[]>**" indique que notre **Observable** devra nous rendre un tableau de Recette.
- "**.pipe**" nous permet d'indiquer des choses à faire en plus du traitement de la requête;

On va donc remplir notre "**.pipe**" avec deux fonctions :

```
tap(response=>console.table(response)),
catchError(err=>{ console.log(err); return of([]) })
```

- "**tap**" peut être imaginer comme le console log de "**rxjs**" il est là pour faire de petites actions sans conséquence.
- "**catchError**" se déclenchera en cas d'erreur, ici il fera un log et retournera "**of**"
- "**of**" crée un observable qui retourne ce qui est donné en paramètre.

> **ATTENTION** : "**tab**", "**catchError**" et "**of**" doivent être importé dans "rxjs"

```
    import { catchError, Observable, of, tap } from 'rxjs';
```

On va maintenant gérer le cas de "**getRecetteById**" auquel on va appliquer le même traitement :

```
getRecetteById(receteId: number):Observable<Recette|undefined>
{
    return this.http.get<Recette>(`api/recettes/${receteId}`).pipe(
        tap(response=>console.table(response)),
        catchError(err=>{
            console.log(err);
            return of(undefined)
        })
    );
}
```

On se retrouve avec la même structure si ce n'est que les types de retour ont changés, que l'url a aussi changé et le paramètre de **of()** aussi.

Avant de passer à la fin on supprimera ce qui est inutile :

```
import { RECETTES } from './RecetteList';
```

De plus on se retrouve avec du code qui se répète, et on n'aime pas cela, alors améliorons ça avec deux nouvelles méthodes en private.

```

```

```
private log(response: any)
{
    console.table(response)
}
private handleError(error: Error, response: any)
{
    console.error(error);
    return of(response) ;
}
```

Enfin on va en changer nos fonctions **tap** et **catchError** dans les deux méthodes où elles sont utilisé :

```
tap(this.log),
catchError(err=>this.handleError(err, []))
// [] ou undefined selon la fonction
```

Cela nous permettra de ne pas avoir à faire des modifications à plusieurs endroits si on souhaite modifier quelque chose.

Maintenant nos composants ont des erreurs car ils ont été développé pour accueillir des données *synchrones* (des objets "Recette" ou des tableaux, mais reçoivent des observable qui sont *asynchrones*)

Commençons avec "**liste-recette.component.ts**", on trouvera une erreur à la ligne :

```
this.recetteList = this.recetteService.getRecetteList();
// Supprimons là et remplaçons là par :
this.recetteService.getRecetteList()
```

On verra qu'une nouvelle méthode est disponible, la méthode `.subscribe()`. Elle va nous permettre de nous abonner à notre observable pour en attendre la réponse. Utilisons là et ajoutons :

```
this.recetteService.getRecetteList().subscribe(liste => this.recetteList = liste)
```

On lui indique, abonne toi à l'observable retourné par `getRecetteList()` puis quand tu as reçu une réponse, donne là à `this.recetteList`. C'est une fonction callback (ici fléché) qui est exécuté lorsque l'abonnement reçoit une réponse.

On va aller faire de même dans "**detail-recette.component.ts**" :

```
this.recetteService.getRecetteById(recetteId).subscribe(recette=> this.recette =
recette);
```

Et puis la même ligne dans "**edit-recette.component.ts**".

Ce que l'on est en train de faire ensemble est appelé un "**CRUD**". Le "**CRUD**" est un des travaux les plus communs du développeur back, il demande beaucoup de vérification et un travail précis, il en va de la sécurité du site. Mais ici on est en Front, on va se contenter d'envoyer des informations à notre "**API**" et elle fera tout le travail. On verra pour faire notre propre "**CRUD**" et notre propre véritable "**API**" dans les cours de back.

Le terme "**CRUD**" est un acronyme dont on a déjà fait la lettre **R** pour "**Read**", c'est à dire les données d'une BDD. Faisons le tour des trois autres lettres ensemble:

Maintenant un problème vicieux s'est faufilé dans notre application. Si on tente d'éditer une de nos recettes, on verra que nos changements ne sont pas sauvegardé.

Update

Pour cela on va venir dans notre "**recette.service.ts**" et ajouter la méthode suivante :

```
updateRecette(recette: Recette): Observable<null>{}
```

Notre méthode prend une recette en argument et retourne un Observable de "**null**"; Jusque là on ne faisait que récupérer des données venant de notre fausse BDD. C'est plutôt facile, mais quand on souhaite en envoyer, cela se complique un peu. On l'avait vu avec "**fetch**" mais on va devoir donner des informations dans le "**header**" de notre requête.

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

Le "**HttpHeaders**" fait partie de la dépendance "**http**" de Angular. Dans notre méthode précédemment créé :

```
const options = {
    headers: new HttpHeaders(
        {
            'Content-Type': 'application/json'
        })
    }
```

On prépare une constante "**options**" à laquelle on donne un objet avec la propriété "**headers**", celle ci contient une nouvelle instance de "**HttpHeaders**" à laquelle on donne les informations de notre "**header**" sous la forme d'un objet.

Ici on dit que les informations que l'on va envoyer seront sous la forme de "**JSON**";

```
// à la suite de notre méthode.
return this.http.put<null>('api/recettes', recette, options).pipe()
```

Ce coup ci on utilisera la méthode "**put**" qui nous permet d'indiquer que ce qui suis est une modification de la BDD. On donnera en paramètre toujours l'url de notre API, suivi de notre objet modifié puis de nos "**options**" contenant le "**Header**". Puis on remplira le pipe de la même manière que précédemment dans "**getRecetteById**";

Pour en finir avec l'update, il nous faut nous rendre dans "**recette-form.component.ts**" dans la méthode "**onSubmit()**" : On va ajouter dans notre condition :

```
this.recetteService.updateRecette(this.recette).subscribe()
```

On fait appel à notre updateRecette à laquelle on donne notre recette mis à jour, puis on s'y abonne et quand on a eu la réponse :

```
// En fonction callback de notre subscribe
()=>this.router.navigate(["/recette", this.recette?.id])
```

On a déplacé notre redirection ici. On attend bien d'avoir la réponse pour changer de page.

DELETE

On va maintenant permettre la suppression d'une de nos recettes. Pour cela on va commencer par notre "**recette.service.ts**" et ajouter comme à l'habitude une nouvelle méthode :

```
deleteRecetteById(recetteId: number): Observable<null>{}
```

Et elle va ressembler trait pour trait à notre "**getRecetteById**" à un détail près, on va utiliser la méthode "**.delete()**" à la place de "**.get()**" et on ne précisera pas la valeur de retour de "**.delete()**". Notre API vérifiera elle même si la requête envoyé vient d'une méthode "**delete**" ou "**get**".

Ensuite il ne nous reste plus qu'à ouvrir notre "**detail-recette.component.ts**" et son "**.html**" pour ajouter un bouton et le lié à une nouvelle méthode.

```
<!-- HTML -->
<button *ngIf="recette" (click)="deleteRecette()">Supprimer</button>
```

```
// TS
deleteRecette(): void
{
    if(!this.recette)return;
    this.recetteService.deleteRecetteById(this.recette.id).subscribe(()=>
this.goToRecetteList())
}
```

La première action que fera notre fonction deleteRecette est de s'arrêter si on a pas de recette sélectionnée. Puis on appellera notre service et une fois qu'on a eu la réponse de notre delete, alors on pourra faire appelle à une méthode défini auparavant `goToRecetteList()` qui nous permet de retourner à notre liste.

Maintenant qu'on a notre "**Read**", notre "**Update**", notre "**Delete**", la seule lettre du "**CRUD**" qui nous manque, c'est le **C** de "**Create**".

Create

Encore une fois nous voici dans notre "**recette.service.ts**" à faire une nouvelle méthode.

```
addRecette(recette: Recette): Observable<Recette>{}
```

Et pour son contenu, ça ne va pas être compliqué, on va copié coller notre méthode "**updateRecette()**" et changer simplement `.put()` par `.post<Recette>()`. et voilà notre "**Create**" est terminé... Si seulement, il nous reste à gérer le formulaire de création.

Créons un nouveau composant :

```
ng generate component recette/add-recette
```

Évidement on l'a créé dans notre module Recette. On va y ajouter un petit html :

```
<h2>Ajouter une nouvelle Recette</h2>
<app-recette-form [recette]="recette"></app-recette-form>
```

Et côté "**add-recette.component.ts**" :

```
recette?: Recette;
ngOnInit(): void
{
    // On défini une nouvelle recette dans une méthode adapté
    this.recette = this.newRecette();
}
newRecette(): Recette
{
    return {
        id: 0,
        name: "",
        type: "Plat",
        image: "",
        description: "",
        duration: 0,
        steps: [],
```

```
        ingredients: [],
        createdAt: new Date()
    }
}
```

Puis on va définir notre route dans notre "**recette.module.ts**":

```
{path: "recette/add", component: AddRecetteComponent}
```

Nous avons un nouveau problème, actuellement notre formulaire, au submit lance "**updateRecette**", Et bien nous allons régler ça grâce à nos routes, allons dans "**recette-form.component.ts**" puis ajoutons une propriété :

```
isAddForm: boolean= false;
// Puis au début du "ngOnInit" :
this.isAddForm = this.router.url.includes("add");
// On vérifie si notre url contient le mot "add"
```

Puis au niveau de mon "**onSubmit**" je vais modifier pour obtenir ceci:

```
if(this.isAddForm){
    this.receetteService.addRecette(this.recette).subscribe(
        (recette)=>this.router.navigate(["/recette", recette.id])
    )
}else{
    this.receetteService.updateRecette(this.recette).subscribe(
        ()=>this.router.navigate(["/recette", this.recette?.id])
    )
}
```

On remarquera que je ne défini pas l'id, tout simplement car si cet identifiant est undefined la BDD va s'occuper automatiquement de lui attribuer un nouvel identifiant.

On ne gère pas le cas des images, mais on va tout de suite inclure dans "**recette-form.component.html**" :

```
<!-- Recette Name -->
<div *ngIf="isAddForm">
    <label for="photo">Photo :</label>
    <input type="text" name="photo" id="photo" required pattern=".+\.
(jpg|jpeg|png)$" [(ngModel)]="recette.image" #photo="ngModel">
    <div class="error" [hidden]="photo.valid || photo.pristine">
        La photo de la recette est requise (jpg ou png)
    </div>
</div>
```

Et on ne l'affiche que si on ajoute une nouvelle recette et pas en cas d'édition. On se contentera d'un input de type text car avec notre fausse API il ne nous serait pas possible d'ajouter un upload de fichier, on est donc bloqué à utilisé les images déjà présente dans cet exemple.