

TD Front-End

Introduction

Comme dans la vraie vie, nous allons avoir affaire à un client (appelons le Fabien) qui sait à peu près ce qu'il veut. L'avantage pour nous c'est que cela va nous permettre de pouvoir faire de nombreux changements et ainsi apprendre les bases pour survivre dans Angular 😟

Il vous a normalement transmis un répertoire Github contenant le code de son précédent développeur qui est malheureusement tombé malade 😢 Vous devriez normalement être en sa possession et pouvoir le faire tourner sur vos machines !

Oui, le TD fait 10 pages mais ne vous inquiétez pas tout va bien se passer 🚀

Exercice 0 : Prenez vos marques !

Petit exercice d'introduction des familles pour vous permettre de faire copain-copain avec le code existant. Vous avez de la chance, le développeur précédent semble avoir bien organisé son code 😊

- 1) Allez dans le fichier **src/mocks/tickets.mock.ts** et modifiez le titre d'un des tickets. Observez maintenant les changements dans votre navigateur.
- 2) Comprendons ce qui vient de se passer :
 - un mock est un fichier qui contient des (fausses) données
 - le mock des tickets est importé dans le service **src/services/ticket/ticket.service.ts** et sert ensuite d'initialisation au flux d'un observable (BehaviorSubject)
 - le composant **TicketList** (**src/app/tickets/ticket-list/ticket-list.component.ts**) s'abonne au flux de l'observable et stocke les données transmises par le flux dans une variable **ticketList**
 - dans l'html associé au composant **TicketList** (**src/app/tickets/ticket-list/ticket-list.component.html**), nous trouvons un ngFor qui permet d'itérer sur la liste des tickets (la fameuse variable **ticketList**) et ainsi d'afficher chaque ticket avec le composant **app-ticket**
 - Le composant **app-ticket** (**src/app/tickets/ticket/ticket.component.ts**) affiche alors les informations du ticket qui lui sont données en paramètre.
- 3) Modifiez du HTML dans le fichier **src/app/tickets/ticket/ticket.component.html** : mettez un "Titre :" devant le **{{ticket.title}}** et constatez le changement
- 4) Modifiez du CSS dans le fichier **src/app/tickets/ticket/ticket.component.scss** en ajoutant "background: red;" dans le block du ".ticket"
- 5) Modifiez le model **Ticket** en remplaçant l'attribut **author** par **student** dans **src/models/ticket.ts**. Malheur ! Votre code ne compile plus ! Il faut mettre à jour **src/app/tickets/ticket/ticket.component.html**, **src/app/tickets/ticket-form/ticket-form.component.ts** (fonction **addTicket**) et **src/mocks/tickets.mock.ts** pour pouvoir de nouveau voir le nom de l'étudiant.

Note : Pensez à ouvrir votre developer console sur votre browser (F12 sur chrome) pour voir vos erreurs de compilation.

Vous devriez maintenant un peu mieux comprendre comment le tout s'articule et fait pour fonctionner ensemble. En cas de besoin n'hésitez pas à refaire ce bref tutoriel pour bien assimiler toutes les notions présentées ici (mock, service, observable, composant (typescript, html, css), model).

Exercice 1 : Qu'est ce que c'est quoi la filière ?

Notre cher Fabien avoue ne pas se rappeler des différentes filières associées à chaque ticket : SI, GE, GB, ... Et comme chaque filière a ses spécificités, il serait plus pratique pour lui de pouvoir le spécifier dans les tickets afin de pouvoir mieux préparer ses rendez-vous.

Étapes :

- Modifiez le model Ticket **src/models/ticket.ts** pour y ajouter l'attribut string **major**
- Modifiez le mock pour ajouter ce nouveau champ aux 2 tickets
- Modifiez le HTML du TicketComponent pour afficher la filière **src/app/tickets/ticket/ticket.component.html**

Exercice 2 : Les icônes, c'est bien

Afin d'apercevoir d'un coup d'oeil rapide la filière associée à chaque ticket, Fabien nous demande d'ajouter des icônes propres à chaque filière. Il vous fait confiance sur le choix des icônes, ne le décevez pas 😊

Étapes :

- Dans **src/app/tickets/ticket/ticket.component.html**, ajoutez une icon spécifique en fonction de la spécialité : [liste des icons](#). Vous avez déjà des exemples d'utilisation des icons dans ce fichier.
- Utiliser la directive **ngIf** pour afficher l'icon en fonction de la spécialité.

Exercice 3 : Un formulaire qui marche, c'est mieux !

Fabien nous a envoyé un mail dont voici le sujet : "À part regarder des tickets, je peux faire quoi ?". Vous l'aurez compris, il commence à s'impatienter et le formulaire qui ne marche pas l'a particulièrement énervé 😡

Étapes :

- dans le fichier **src/app/tickets/ticket-form/ticket-form.component.html**, ajoutez un nouvel input text pour pouvoir donner le nom de la filière (major)

- **src/app/tickets/ticket-form/ticket-form.component.ts**, ajoutez le nouvel attribut **major** dans l'objet **ticketForm** définit dans le constructeur comme pour les attributs **title** et **description** (si besoin: [documentation form](#)).
Note: La dernière ligne dans **ticket.component.html** est commentée. **Décommentez là** pour voir votre objet **ticketForm** changé dynamiquement lorsque vous remplissez le formulaire !
- Dans **src/services/ticket/ticket.service.ts** remplissez la méthode **addTicket** en 2 lignes :
 - Ajout du nouveau ticket dans la liste ([doc array](#))
 - Mise à jour de l'observable ([Update observable with new value](#))

Exercice 4 : Trop de tickets, tue le ticket...

Voilà qui est nettement mieux, mais à force d'avoir des rendez-vous, le dashboard de Fabien est rempli de tickets ! Il vous demande donc d'ajouter en urgence une fonctionnalité lui permettant de supprimer des tickets car il n'arrive plus à vivre dans ces conditions.

Étapes :

- Dans **TicketComponent** :
 - **src/app/tickets/ticket/ticket.component.html** : Ajoutez un bouton de suppression
 - **src/app/tickets/ticket/ticket.component.ts** :
 - Ajoutez un **@Output** pour notifier le composant parent de la suppression (voir **Note** juste en dessous). L'output a comme type **EventEmitter<Ticket>**.
 - Créez une fonction **deleteTicket()** qui va émettre le ticket courant.
 - **src/app/tickets/ticket/ticket.component.html**: Ajoutez la fonction **deleteTicket()** à votre bouton de suppression pour qu'elle soit appelée lors du **(click)**. Vous avez un exemple dans ce composant avec la fonction **selectTicket()**.
- Dans **TicketService** :
 - **src/services/ticket/ticket.service.ts** : Créez une fonction **deleteTicket(ticket: Ticket)** pour supprimer le ticket de la liste et mettre à jour l'observable.
- Dans **TicketListComponent** :
 - **src/app/tickets/ticket-list/ticket-list.component.ts** : Créez une fonction **deleteTicket(ticket: Ticket)** dans la classe.
 - **src/app/tickets/ticket-list/ticket-list.component.html** : Modifiez l'appel du composant **<app-ticket>** afin de rajouter votre nouvel Output (comme ce qui existe déjà pour **ticketHasBeenSelected**) et votre nouvelle fonction (**deleteTicket(\$event)**) qui sera appelée lorsqu'un événement est émis.
 - **src/app/tickets/ticket-list/ticket-list.component.ts** : Appelez la fonction **deleteTicket()** du service **TicketService** pour supprimer un ticket.

Note :

Vous avez un exemple d'output appelé **ticketHasBeenSelected** dans **TicketComponent** sur lequel **TicketListComponent** réagit (regardez dans la console de votre browser) + [documentation pour les @Output](#)

Exercice 5 : `input[type=text]` → `select`

Fabien nous fait savoir qu'il ne trouve pas cela très adapté d'entrer du texte pour renseigner le champ "filière". Il aimeraient que cela soit plus standard et passe par la sélection parmi une liste.

Étapes :

- Changez le champ **major** du model Ticket pour que celui-ci devienne un enum : <https://www.typescriptlang.org/docs/handbook/enums.html>
- Dans `src/app/tickets/ticket-form/ticket-form.component.html`, modifiez le champ d'input text par un `select`.

Le développeur malade vous envoie un de ses `select` d'une ancienne application pour vous aider :

Dans sa classe `RecipeFormComponent` :

```
public INGREDIENT_LIST: string[] = ['Pasta', 'Nutella'];
```

Dans l'html du composant `RecipeFormComponent`:

```
Ingredient:  
<select name="ingredient-field" id="ingredient" [formControlName]="'ingredient'">  
  <!-- We set the formControlName to the string 'ingredient'-->  
  <option *ngFor="let currentIngredient of INGREDIENT_LIST" [value]="currentIngredient">  
    <!-- We set the value of the option to the object currentIngredient created from the ngFor-->  
    {{ currentIngredient }}  
  </option>  
</select>
```

Exercice 6 : j'ai perdu tous mes tickets

Il semblerait que Fabien se soit mal exprimé lors de l'exercice 4 (où alors que nous ne lui avons pas assez posé de question pour valider son besoin) 😞 Apparemment, il aurait voulu archiver ses tickets, et non les supprimer ! #oops

Heureusement, il ne demande pas de dédommagement pour le temps et les tickets perdus, nous pouvons nous estimer heureux 😊

Étapes :

- Ajoutez un boolean **archived** dans le model Ticket
- Mettez à jour le fichier de mock pour ajouter un ticket archivé
- transformez la fonctionnalité de suppression en archivage dans les fichiers nécessaires (à vous de trouver cette fois)
- `src/app/tickets/ticket-list/ticket-list.component.ts`:
 - Ajoutez un attribut `displayTicketArchived: boolean` dans la classe du composant
- `src/app/tickets/ticket-list/ticket-list.component.html`:

- Ajoutez un bouton sur l'interface permettant de filtrer les tickets en fonction de s'ils sont archivés ou pas. Faites un simple bouton. Il vous suffit de mettre à jour votre nouvel attribut **displayTicketArchived** lorsque l'utilisateur clique sur votre bouton.



- Il faut maintenant mettre à jour le contenu de votre ***ngfor** en rajoutant un ***ngIf** afin de filtrer les tickets archivés. Si **displayTicketArchived** est true, vous affichez uniquement les tickets archivés.

Exercice 7 : Qu'est ce que c'est qui l'étudiant ?

Tout comme pour la filière dans l'exercice 5, il est peu pratique pour Fabien d'écrire en texte le nom de l'étudiant (vous savez le champ que vous avez renommé **student** lors de l'exercice 0) sachant qu'il s'agit d'une liste prédéfinie dans son système informatique. Vous allez donc devoir lui permettre à l'aide d'un select, de choisir un étudiant parmi une liste. En revanche, cette fois, il ne s'agira pas d'une simple string à sélectionner 😊

Étapes :

- Créeation d'un modèle Student pour les étudiants avec
 - Un id
 - Un nom
 - Un prenom
- Créez un mock contenant une liste de Student
- Créez un Student Service
- Mettez à jour l'attribut **student** au type Student dans le modèle Ticket
- Mettez à jour le mock des tickets pour remplir comme il faut le champ **student**,
- Dans TicketComponent, affichez le nom de l'étudiant en respectant le format.
- Mettez à jour le ticketForm (dans **src/app/tickets/ticket-form/ticket-form.component.ts**) en rajoutant le champ **studentID**. Vous stockerez uniquement l'ID du student.
- Dans TicketFormComponent, ajoutez un select pour choisir le nom de l'étudiant en respectant le format d'affichage ci-dessous.
- Mettez à jour la fonction **addTicket()** pour récupérer le bon **Student** à partir du **studentID** sélectionné. Une fois l'objet récupéré, vous pouvez initialiser votre **ticketToCreate** avant l'envoi au service.

Format d'affichage des étudiants : \${id} - \${nom} \${prénom}

Exemples :

- 1 - Delmotte Jean-Yves
- 2 - Pourtier Rémi

Exercice 8 : Je veux la vraie liste des utilisateurs

L'ancien développeur est de retour dans la course et a pu coder une partie du back-end concernant la gestion des utilisateurs. Fabien veut désormais supprimer les mocks des utilisateurs et que vous fassiez appel à l'URL suivante pour charger la liste initiale des utilisateurs.

URL de la liste des utilisateurs : <https://api.myjson.com/bins/ck44c>

Pensez à rajouter l'identifiant des utilisateurs dans votre modèle, ça peut être utile par la suite 😊

Étapes :

- Ajoutez HttpClientModule dans la liste des **imports:[]** dans **@NgModule** dans **src/app/app.module.ts** ([exemple ici](#))
- Dans StudentService :
 - Créez un attribut pour stocker l'url à appeler
 - Injectez HttpClient dans le constructeur du service ([exemple ici](#))
 - Créez une fonction getStudent() qui fait l'appel à l'url et récupère la liste des utilisateurs ([exemple ici avec /** GET heroes from the server */](#). **Attention** dans l'exemple ils retournent l'observable, mais nous ne voulons pas faire ça)
 - On utilise le **subscribe** sur le **this.http.get()** pour récupérer la réponse reçue et ainsi mettre à jour le la liste des utilisateurs ainsi que le behavior subject. (vous avez un exemple de subscribe dans TicketListComponent)

Note : Si vous voulez mettre plusieurs lignes dans votre subscribe, vous devez rajouter des accolades :

```
this.ticketService.tickets$.subscribe( next: (tickets) => this.ticketList = tickets);
this.ticketService.tickets$.subscribe( next: (tickets) => {
  this.ticketList = tickets;
  console.log(tickets);
});
```

Vous pouvez aussi créer une fonction et l'appeler pour éviter d'avoir des pavés illisibles dans vos subscribes 😊

Fin du TD

Wahou ! Enfin arrivé au bout du TD.. 1... Comme dirait la voix : "C'est tout... Pour le moment 😊". Fabien va pouvoir enfin faire mumuse avec votre super projet et vous faire de nouveaux retours d'ici peu !

Suite TD1

Exercice 9 : Je veux voir mes utilisateurs

Maintenant que nous avons un système de ticket viable, Fabien aimerait que vous vous attaquiez au second point de son besoin à savoir : la gestion de utilisateurs. Pour cela nous allons déjà commencer par afficher la liste sur une page dédiée aux utilisateurs. Vous allez donc devoir setup un système de routing et recréer un certain nombre de fichier en vous inspirant de ce qui a déjà été fait pour les tickets.

Étapes pour le routing :

- Création du module AppRoutingModule
 - Ajout des routes /students et /tickets
 - Ajout du RouterModule.forRoot()

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { TicketListComponent } from './tickets/ticket-list/ticket-list.component';
import { UserListComponent } from './users/user-list/user-list.component';

const routes: Routes = [
  {path: 'tickets', component: TicketListComponent},
  {path: 'users', component: UserListComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

- Ajouter AppRoutingModule dans AppModule dans la liste des imports

```
imports: [
  BrowserModule,
  ReactiveFormsModule, // Import all dependencies
  AppRoutingModule
],
```

- Remplacer l'appel aux composants dans app.component.html par le router-outlet

```
<app-header></app-header>
<div class="content">
  <router-outlet></router-outlet>
</div>
```

- Accéder à vos pages en rajoutant /students ou /tickets

Useful links :

- Toutes les étapes décrites ici: [Tutorial routing !](#)
- [Display a list](#)
- [Inputs: Pass data from parent to child](#)
- [Services](#)

Exercice 10 : Je veux pouvoir créer un utilisateur

Étapes :

- Création d'un composant StudentFormComponent en charge de l'ajout d'un utilisateur
- Mise à jour du service permettant d'ajouter un nouvel utilisateur dans la liste
- Mise à jour de la liste des utilisateurs affichés après ajout.

Exercice 11 : Je veux supprimer un utilisateur

Étapes :

- Ajout d'un bouton pour supprimer l'utilisateur dans le composant StudentComponent et envoie d'un événement au composant père
- StudentListComponent reçoit l'événement et appelle le service pour supprimer l'utilisateur
- Mise à jour du service pour supprimer un utilisateur

Exercice 12 : Je veux prendre des notes sur les utilisateurs

Avec les nombreux rendez-vous que Fabien a, il ne se souvient plus forcément de qui est qui. Les photos lui sont d'une grande aide mais ce qu'il aimerait par dessus tout c'est pouvoir prendre des notes ! Vous allez donc devoir implémenter le UI pattern master/detail afin de pouvoir se rendre sur la page de chaque utilisateur et avoir un **textarea** afin que Fabien puisse mettre ses notes.

Étapes :

- Mise à jour du modèle d'utilisateur pour stocker les notes
- Mise à jour du routing pour naviguer vers la page de détail de l'utilisateur avec la route suivante qui comporte un paramètre dans son url : students/:id avec id = id de l'utilisateur (ex: students/11)

- Création d'un composant StudentDetailComponent qui affiche les détails de l'utilisateur ainsi que le champ **textarea** pour prendre des notes.
- Mise à jour du service d'utilisateur pour mettre à jour les notes pour le bon utilisateur.

Useful links :

- [Steps to add a parameter in the url !](#)
- [Extract id from the route !](#)

Exercices Optionnels :

Exercice 13 : Je veux des beaux utilisateurs

Fabien semble plutôt content de son nouveau système permettant de lister les utilisateurs. Cependant il m'a transmis ceci : "Ils ne sont pas à mon image, ils ne sont pas beaux". Il a fait faire une maquette à un designer professionnel dont voici le résultat :

Vous devez faire au mieux pour vous rapprochez de cette maquette 😊

Nom	Prénom	Email	
	Jean-Yves	Delmotte	jean-yves@delmotte.com
	Rémi	Pourtier	remi@pourtier.com

Quelques indices pour vous aider :

<https://developer.mozilla.org/fr/docs/Web/CSS/box-shadow>
<https://developer.mozilla.org/fr/docs/Web/CSS/border-radius>
<https://developer.mozilla.org/fr/docs/Web/CSS/font-weight>
<https://developer.mozilla.org/fr/docs/Web/CSS/font-style>

Exercice 14 : la liste des utilisateurs devient trop longue...

Cela fait maintenant plusieurs jours que Fabien utilise votre superbe application mais il a cependant un dernier retour à faire. En effet, sa liste d'utilisateurs ne cesse de s'allonger et il devient pénible pour lui d'utiliser le <select> de l'exercice précédent ! Il aimerait avoir un champ de recherche filtrant les résultats afin de sélectionner plus rapidement le **bon** utilisateur.

Vous trouverez ci-dessous des suggestions de notre cher ami que nous avons reçu en pièces jointes de son mail.

Pick one

- One
- Two
- Three

Pick one

- o
- One
- Two

Étapes :

- Mettre à jour le TicketComponent avec un **input** pour filtrer parmi les utilisateurs..
- Pour le filtre : le texte entré dans l'input doit être contenu soit dans le nom, soit dans le prénom.
- Ne pas afficher plus de 5 utilisateurs (sinon pour la lettre "e" on se retrouve avec le même problème et il faut inciter Fabien à entrer plus qu'une lettre..)
- Une fois l'utilisateur sélectionné, l'input doit disparaître et une croix de suppression doit permettre de supprimer l'utilisateur pour revenir à l'état initial