Angular TP-04

Améliorez la structure du code avec les Services

Créez maintenant un sous-dossier services dans app, et créez-y un nouveau fichier appelé student.service.ts :

export class StudentService {

}

Vous y intégrerez bientôt des données et des fonctions, mais pour l'instant, vous allez injecter ce service dans AppModule avec un import et en l'ajoutant à l'array providers.

Pour l'intégrer dans un component, on le déclare comme argument dans son constructeur. Importez-le dans AppComponent et passer le service en argument dans son constructeur :

constructor(private studentService: studentService)

Utilisez le service student

Le premier élément qu'il serait logique de déporter dans le service serait l'array students. Copiez-le depuis AppComponent, collez-le dans studentService et, de nouveau dans AppComponent, déclarez students simplement comme un array de type any :

```
export class studentService {
  students = [
    {
      name: 'Louis',
      status: 'absent'
    },
    {
      name: 'Henri',
      status: 'present'
    },
    {
      name: 'Philippe’,
      status: 'absent'
    }
  ];
}
```

Il faut maintenant que AppComponent puisse récupérer les informations stockées dans studentService. Pour cela, vous allez implémenter la méthode ngOnInit().

Importez la méthode depuis @angular/core,

import { Component, OnInit } from '@angular/core';

puis ajoutez-la après le constructeur :

```
ngOnInit() {
        Ajouter l'affectation ici !
}
```

puis affectez l'array students du service à celui du contexte.

Testez !

Ajouter des méthodes au service

Pour manipuler les données, nous allons créer deux nouvelles méthodes :

switchOnAll() et switchOffAll() :

```
switchOnAll() {
   for(let student of this.students) {
     student.status = 'present';
   }
}

switchOffAll() {
   for(let student of this.students) {
     student.status = 'absent';
   }
}
```

Puis ajoutez un deuxième bouton dans le template de AppComponent :

```
<button class="btn btn-success"
            [disabled]="!isAuth"
            (click)="allPresent()">Tous Présents</button>
<button class="btn btn-danger"
            [disabled]="!isAuth"
            (click)="allAbsent()">Tous Absent</button>
```

Enfin, il ne vous reste plus qu'à capturer les événements click dans AppComponent pour ensuite déclencher les méthodes dans studentService.

Pour allPresent(), rajoutez après l'alert, l'appel dans le contexte de switchOnAll().

Ensuite, pour allAbsent(), vous allez d'abord afficher un message de confirmation :

```
allAbsent() {
   if(confirm('Etes-vous sûr qu\'ils sont tous absents ?')) {
     this.studentService.switchOffAll();
   } else {
```

```
    return null;
  }
}
```
Testez !

Faites communiquer vos components

studentService fournit les données sur les étudiants à AppComponent. Mais il n'y a actuellement aucune communication entre les components enfants et leur parent.

Récupérer l'index de la directive *ngFor dans le Html de studentComponent :

*ngFor="let student of students; let i = index"

Cette commande rend disponible l'index de l'objet student dans l'array students.

Ajoutez un membre index de type number au component en tant que @Input() :

Puis liez-y l'index i depuis le template Html

Dans studentService, vous allez maintenant créer les méthodes permettant de changer le status d'un seul étudiant :

```
switchOnOne(i: number) {
  this.students[i].status = 'present';
}

switchOffOne(i: number) {
  this.students[i].status = 'absent';
}
```

Ensuite, dans studentComponent, vous allez d'abord intégrer le service studentService, en l'important en haut du fichier et en le passant dans le constructeur.

constructor(private studentService: studentService) { }

Puis vous allez préparer la méthode qui, en fonction du statut actuel de l'étudiant le changera en present ou absent :

```
onSwitch() {
  if(this.studentStatus === 'present') {
    this.studentService.switchOffOne(this.index);
  } else if(this.studentStatus === 'absent') {
    this.studentService.switchOnOne(this.index);
  }
}
```

Puis dans le template, vous allez créer 2 boutons qui déclencheront cette méthode.

```
<input type="text" class="form-control" [(ngModel)]="studentName">
<button class="btn btn-sm btn-success"
        *ngIf="studentStatus === 'absent'"
        (click)="onSwitch()">Présent</button>
<button class="btn btn-sm btn-danger"
```

```
        *ngIf="studentStatus === 'present'"
        (click)="onSwitch()">Absent</button>
</li>
```

Vous pouvez supprimer la conditionnelle rouge, car elle ne sert plus vraiment, avec tous les styles que vous avez ajoutés pour signaler l'état d'un étudiant.

Testez !

Vos components communiquent entre eux à l'aide du service, qui centralise les données et certaines fonctionnalités.

Gérez la navigation avec le Routing

Je vous propose de créer un component pour l'authentification (qui restera simulée pour l'instant) et vous créerez un menu permettant de naviguer entre les views.

Tout d'abord, créez le component avec le CLI :

ng g c auth

Vous allez également devoir modifier un peu l'organisation actuelle afin d'intégrer plus facilement le routing : vous allez créer un component qui contiendra toute la view actuelle et qui s'appellera studentViewComponent :

ng g c student-view

Ensuite, coupez tout le contenu de la colonne dans app.component.html, enregistrez-le dans student-view.component.html, et remplacez-le par la nouvelle balise <app-student-view> :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <app-student-view></app-student-view>
    </div>
  </div>
</div>
```

Il faut également déménager la logique de cette view pour que tout marche :

Injectez studentService, créez l'array students, intégrez la logique ngOnInit et déplacez les fonctions allPresent() et allAbsent() :

Vous pouvez faire le ménage dans AppComponent, en retirant tout ce qui n'y sert plus.

Créez également un boolean isAuth dans studentViewComponent, et déclarez-le comme false.

Testez ! (au cas-où sa marche plus :/)

Ajoutez la barre de navigation suivante à AppComponent :

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-collapse">
      <ul class="nav navbar-nav">
```

```
      <li><a href="#">Authentification</a></li>
      <li class="active"><a href="#">students</a></li>
    </ul>
   </div>
  </div>
</nav>
```

Testez !

Créez des routes

Dans app-routing.module.ts.

On crée une constante de type Routes qui est un array d'objets JS qui prennent une certaine forme. On importe les vues que l'on veut router, et on import Routes et RouterModule :

```
import { Routes, RouterModule } from '@angular/router';
import …. des 2 vues

const appRoutes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'students' },
  { path: 'students', component: StudentViewComponent },
  { path: 'auth', component: AuthComponent }
];
```

On a ajouté un path vide, qui correspond tout simplement à localhost:4200 (ou à la racine de l'application seule).

Modifier le @NgModule :

```
@NgModule({
        imports : [RouterModule.forRoot(appRoutes)),
        exports : [RouterModule]
})
```

Puis ajoutez l'export d'un array des components routés, après l'export de class.

```
export const routedComponents = [StudentViewComponent, AuthComponent] ;
```

Il faut maintenant mettre à jour votre app.module.ts :

Rajoutez à l'import app-routing.module l'array routedComponent.

Remplacez dans la déclaration de NgModule, les 2 modules routés par routedComponent. Supprimez les 2 imports de trop.

Et ajouter dans l'imports de NgModule AppRoutingModule.

Maintenant que les routes sont enregistrées, il ne reste plus qu'à dire à Angular où vous souhaitez afficher les components dans le template lorsque l'utilisateur navigue vers la route en question. On utilise la balise <router-outlet> dans le html de app.component :

```
   <div class="col-xs-12">
     <router-outlet></router-outlet>
```

```
    </div>
```

Testez !

Lorsque vous changez de route, en modifiant l'URL directement dans la barre d'adresse du navigateur, la page n'est pas rechargée, mais le contenu sous la barre de navigation change.

Naviguez avec les routerLink

Dans la navbar, on retire l'attribut href et on le remplace par l'attribut routerLink :

```
<li><a routerLink="auth">Authentification</a></li>
<li class="active"><a routerLink="students">Etudiants</a></li>
```

Ainsi, les routes sont chargées instantanément : on conserve l'ergonomie d'une SPA.

Pour finaliser cette étape, rajoutez l'attribut routerLinkActive="active" et enlevez class="active">.

Testez !

Maintenant, les liens s'activent visuellement.

Naviguez avec le Router

Tout d'abord, créez un nouveau fichier auth.service.ts dans le dossier services pour gérer l'authentification (n'oubliez pas de l'ajouter également dans l'array providers dans AppModule) :

```
export class AuthService {
  isAuth = false;

  signIn() {
    return new Promise(
      (resolve, reject) => {
        setTimeout(
          () => {
            this.isAuth = true;
            resolve(true);
          }, 3000
        );
      }
    );
  }

  signOut() {
    this.isAuth = false;
  }
}
```

La variable isAuth donne l'état d'authentification de l'utilisateur.

La méthode signOut() "déconnecte" l'utilisateur.

La méthode signIn() authentifie automatiquement l'utilisateur au bout de 3 secondes, simulant le délai de communication avec un serveur.

Dans le component AuthComponent, vous allez simplement créer deux boutons et les méthodes correspondantes pour se connecter et se déconnecter.

D'abord dans le Ts.

Importer auth.service et injectez-le dans le constructeur.

Ajoutez dans la class une variable authStatus de type Boolean.

Affectez la valeur de authService.isAuth à authStatus dans NgOnInit.

Puis ajoutez les 2 fonctions :

```
onSignIn() {
  this.authService.signIn().then(
    () => {
      alert('Sign in successful!');
      this.authStatus = this.authService.isAuth;
    }
  );
}

onSignOut() {
  this.authService.signOut();
  this.authStatus = this.authService.isAuth;
}
```

Puisque la méthode signIn() du service retourne une Promise, on peut employer une fonction callback asynchrone avec .then() pour exécuter du code une fois la Promise résolue.

Ajoutez simplement les boutons, et tout sera prêt pour intégrer la navigation :

```
<h2>Authentification</h2>
<button class="btn btn-success" *ngIf="!authStatus" (click)="onSignIn()">Se
connecter</button>
<button class="btn btn-danger" *ngIf="authStatus" (click)="onSignOut()">Se déconnecter</
button>
```

Le comportement recherché serait qu'une fois l'utilisateur authentifié, l'application navigue automatiquement vers la view des students. Pour cela, il faut injecter le Router dans le constructeur (importé depuis angular/router) pour accéder à la méthode navigate() :

```
this.router.navigate(['students']);
```

Testez !

La fonction navigate prend comme argument un array d'éléments (ce qui permet de créer des chemins à partir de variables, par exemple) qui, dans ce cas, n'a qu'un seul membre : le path souhaité.

Paramètres de routes

Nous souhaitons pouvoir visualiser éditer les étudiants un par un.

Tout d'abord, vous allez ajouter la route dans App-routing.module, après la route ‘students’ :

```
  { path: 'students/:id', component: SingleStudentComponent },
```

L'utilisation des deux-points : avant un fragment de route déclare ce fragment comme étant un paramètre : tous les chemins de type students/* seront renvoyés vers SingleStudentComponent, que vous allez maintenant créer :

ng g c singleStudent

Importez et injecter dans le constructeur studentService.

Créez les variables name et status de type string affectez leur respectivement ‘etudiant’ et ‘status’.

Remplacez le Html par :

```
<h2>{{ name }}</h2>
<p>Statut : {{ status }}</p>
<a routerLink="/students">Retour à la liste</a>
```

Pour l'instant, si vous naviguez vers /students/nom, peu importe le nom que vous choisissez, vous avez accès à SingleStudentComponent. Maintenant, vous allez y injecter ActivatedRoute, importé depuis @angular/router, afin de récupérer le fragment id de l'URI.

Puis, dans ngOnInit(), vous allez utiliser l'objet snapshot qui contient les paramètres de l'URI et, pour l'instant, attribuer le paramètre id à la variable name :

```
ngOnInit() {
    this.name = this.route.snapshot.params['id'];
}
```

Puis ajouter, dans studentService, un identifiant unique pour chaque étudiant et une méthode qui donnera l’étudiant correspondant à un identifiant :

```
students = [
    {
      id: 1,
      name: 'Henri',
      status: 'absent'
    }, etc ...
```

```
getStudentById(id: number) {
    const student = this.students.find(
      (s) => {
        return s.id === id;
      }
    );
    return student;
}
```

Maintenant, dans SingleStudentComponent, vous allez récupérer l'identifiant de l'URI et l'utiliser pour récupérer l'étudiant correspondant :

```
ngOnInit() {
    const id = this.route.snapshot.params['id'];
    this.name = this.studentService.getstudentById(+id).name;
    this.status = this.studentService.getstudentById(+id).status;
}
```

Puisqu'un fragment d'URL est forcément de type string, et que la méthode getStudentById() prend un nombre comme argument, il ne faut pas oublier d'utiliser + avant id dans l'appel pour **caster** la variable comme nombre.

Pour finaliser cette fonctionnalité, intégrez l'identifiant unique dans studentComponent et dans studentViewComponent, puis ajouter un routerLink pour chaque étudiant qui permet d'en regarder le détail.

@Input() id: number

puis :

[id]="student.id"></app-student>

Et à la fin :

<a [routerLink]="[id]">Détail

Ici, vous utilisez le format array pour routerLink en property binding afin d'accéder à la variable id.

Testez !

Redirection

Il peut y avoir des cas de figure où l'on souhaiterait rediriger un utilisateur, par exemple pour afficher une page 404 lorsqu'il entre une URL qui n'existe pas.

Commencez par créer un component four-oh-four.component.ts :

```
<h2>Erreur 404</h2>
<p>La page que vous cherchez n'existe pas !</p>
```

Ensuite, vous allez ajouter la route "directe" vers cette page, ainsi qu'une route "wildcard", qui redirigera toute route inconnue vers la page d'erreur :

```
{ path: 'not-found', component: FourOhFourComponent },
{ path: '**', redirectTo: 'not-found' }
```

Testez !

Ainsi, quand vous entrez un chemin dans la barre de navigation qui n'est pas directement pris en charge par votre application, vous êtes redirigé vers /not-found et donc le component 404.

Guards

Dans le shell ajouter au projet Rxjs-compat :

npm install rxjs-compat

Une guard est en effet un service qu'Angular exécutera au moment où l'utilisateur essaye de naviguer vers la route sélectionnée. Ce service implémente l'interface canActivate, et donc doit contenir une méthode du même nom qui prend les arguments ActivatedRouteSnapshot et RouterStateSnapshot (qui lui seront fournis par Angular au moment de l'exécution) et retourne une valeur booléenne, soit de manière synchrone (boolean), soit de manière asynchrone (sous forme de Promise ou d'Observable) :

Créez un service auth-guard.service.ts.

Ajoutez le code suivant :

```
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot } from
'@angular/router';
import { Observable } from 'rxjs/Observable';

export class AuthGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

  }
}
```

Ensuite, il faut injecter le service AuthService dans ce nouveau service. Pour injecter un service dans un autre service, il faut que le service dans lequel on injecte un autre ait le décorateur @Injectable, à importer depuis @angular/core :

@Injectable() avant la class

Puis importez et injectez le AuthService dans le constructeur

Maintenant imaginez une fonction qui renverra true si l'utilisateur est authentifié ou qui redirigera l'utilisateur vers la page /auth.

Placez là dans canActivate.

Pour appliquer cette garde à la route /students et à toutes ses routes enfants, il faut l'ajouter dans app-routing.module. Il faut ajouter AuthGuard à l'array providers du @NgModule puisqu'il s'agit d'un service.

Modifier les Routes :

```
{ path: 'students', canActivate: [AuthGuard], component: StudentViewComponent },
{ path: 'students/:id', canActivate: [AuthGuard], component: SinglestudentComponent },
```

Testez !

Maintenant, si vous essayez d'accéder à /students sans être authentifié, vous êtes automatiquement redirigé vers /auth. Si vous cliquez sur "Se connecter", vous pouvez accéder à la liste d'étudiant sans problème.

Dans ce chapitre, vous avez appris à gérer le routing de votre application avec des routerLink et router.navigate(). Vous avez également vu comment rediriger un utilisateur, et comment protéger des routes de votre application avec les guards.

Si vous avez encore un peu de temps. Il est temps de rendre l'application un peu plus jolie !

Rdv sur W3schools pour un peu de Css : https://www.w3schools.com/w3css/default.asp

et de bootstrap : https://www.w3schools.com/bootstrap/default.asp