TP Angular ®

Concepts fondamentaux V14

Hervé Le Cornec, herve.le.cornec@free.fr

Angular est basé sur une technologie unique, que ne possèdent pas les autres frameworks. En comprenant le principe de cette technologie, on parvient à coder des frontends de façon plus rapide, plus maintenable et plus évolutive.

Vous serons données ici les bases sur lesquelles fonder toutes vos applications Angular, en tirant bénéfice de cette technologie unique. Vous apprendrez notamment à baser votre architecture sur un store global dans un service partagé, afin de faire communiquer automatiquement les composants entre eux, et comment gérer les asynchronismes une fois pour toutes.

Une fois ce TP acquis, le lecteur pourra améliorer son savoir en lisant le cours complet **angular.v14.pdf**, où de nombreux détails sont donnés, et à vrai dire la très grande majorité de ce que vous devrez utiliser dans une web app.

Mais rappelez-vous, l'architecture que vous allez développer dans ce TP devra être le socle sur lequel fonder toutes vos autres applications Angular.

1) Prérequis

Nous allons utiliser Angula 14, qui requiert un Node 16. Il est donc nécessaire que votre machine soit équipée de cette version de Node. Nous vous conseillons la version 16.13.1, disponible dans votre centre logiciel. Vous pourrez l'installer même sans avoir les droits administrateur sur votre machine.

Sans les droits administrateur sur votre machine, vous ne pourrez pas installer Angular de façon globale (voir le cours **angular.v14.pdf**). Nous procéderons donc par une installation locale.

2) Création de l'application « tp-Angular-v14 »

Dans une console entrez les commandes suivantes :

```
# npx -p @angular/cli ng new tp-angular-v14
…
? Would you like to add Angular routing? Yes
…
```

```
? Which stylesheet format would you like to use? CSS
……
# cd tp-angular-v14
```

Comme vous ne possédez pas d'Angular de façon globale, vous ne pourrez pas lancer la commande:

```
# ng serve
```

Il vous faudra lancer :

```
# npm run ng serve
```

Cependant le fichier **`package.json`** vous indique les raccourcis **`npm`** que vous pouvez utiliser, ainsi la commande précédente peut être appelée autrement :

```
# npm run start
```

Notez qu'en utilisant le fichier **`package.json`**, vous pourrez créer vos propres scripts, par exemple :

```
"start4600": "ng serve –port 4600",
…
"BuildwForLondon" : "ng build –environement london"
…
```

Ouvrir un onglet dans le navigateur et lui indiquer l'url **`localhost:4200`**, la nouvelle application apparaît.

3) Structure de base du filesystem

Il est nécessaire d'utiliser l'arborescence de fichiers la plus simple possible

```
app
  header
  pages
     profile
     news
     home
  lib
     class
     pipes
     services
        api.service
        store.service
app-routing.module.ts
app.component.css
app.component.html
app.component.ts
app.module.ts
```

4) Création du composant header

Tout site possède un header, indépendant des pages et général à l'application. Nous créons donc ce composant :

```
# cd src/app
# npm run ng generate component header
CREATE src/app/header/header.component.css (0 bytes)
CREATE src/app/header/header.component.html (25 bytes)
CREATE src/app/header/header.component.spec.ts (628 bytes)
CREATE src/app/header/header.component.ts (269 bytes)
UPDATE src/app/app.module.ts (475 bytes) <<< le composant header a été déclaré
                                             dans app.module.ts
```

5) Création de la page d'accueil

A minima votre application possédera une page d'accueil. Cette page est un composant comme un autre, que nous générons :

```
# mkdir pages
# cd pages
# npm run ng generate component home
```

Le composant home sera une fois encore embarqué automatiquement dans **`app.module.ts`**.

6) Routage dans le AppRoutingModule

Il est de bonne pratique de détacher le routing de l'application dans un **`AppRoutingModule`** (fichier **`app-routing.module.ts`**) distinct du **`AppModule`** général. On déclare alors les composants non plus directement dans le **`AppModule`**, mais dans le **`AppRoutingModule`**.

• Déplacer l'import du composant home depuis **`app.module.ts`**, vers **`app-routing.module.ts`**

• Renseigner la route vers home dans **`app-routing.module.ts`** :

```
const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path : '**',
    redirectTo : '',
    pathMatch: 'full'
  }
];
```

7) Création de la page « news »

On pratique comme pour la page home

```
# cd pages
# npm run ng generate component news
```

Le composant home sera une fois encore embarqué automatiquement dans **app.module.ts**. On le déplace dans le **app-routing.module.ts**, comme on l'a fait pour la page home, et on y déclare une route « news ». Au final le **AppRoutingModule** donnera cela :

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './pages/home/home.component';
import { NewsComponent } from './pages/news/news.component';

const routes: Routes = [
  {
     path: '',
     component: HomeComponent
  },
  {
     path: 'news',
     component: NewsComponent
  },
  {
     path : '**',
     redirectTo : '',
     pathMatch: 'full'
  }
];

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

Par le navigateur on peut désormais afficher **localhost:4200/news** et vérifier que tout fonctionne.

8) Modifier `app.component.html` pour afficher le header et le workspace

Tout effacer et ne laisser que :

```
<app-header></app-header>
<router-outlet></router-outlet>
```

L'application affichera le header et en dessous les pages définies dans le routing.

9) Installer Bootstrap

Vous aurez certainement besoin d'une bibliothèque de CSS déjà existante, la plus commune étant Bootstrap.

L'installer vous fera immédiatement bénéficier d'un design très agréable pour l'application.

• Installer Bootstrap (voir https://ng-bootstrap.github.io) en lançant la commande

```
# npm i --save bootstrap
```

• Importer Bootstrap dans le fichier **`angular.json`** en y ajoutant

```
...
"styles": [
  "./node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
...
```

Notez que nous ne parlons pas ici de la bibliothèque de composants Ng-Bootstrap, mais seulement de la bibliothèque CSS Bootstrap.

10) Créer une navigation dans le header

• Insérons une navbar Bootstrap dans le header. Modifier **`header.component.html`** :

```
<nav class="navbar navbar-expand-lg navbar-light bg-light ps-4 pe-4">
  <a class="navbar-brand" href="#">Navbar</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse"
     data-target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup"
     aria-expanded="false" aria-label="Toggle navigation">
     <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
     <div class="navbar-nav">
        <a class="nav-item nav-link active" routerLink="">Home</a>
        <a class="nav-item nav-link" [routerLink]="['news']">News</a>
     </div>
  </div>
</nav>
```

• Embarquer le **`RouterModule`** dans **`app.module.ts`**

```
import { RouterModule } from '@angular/router';
...

  imports: [
     ...
     RouterModule
  ],
```

• Vérifier qu'on navigue d'une page à l'autre en cliquant sur les liens

11) Créer un service d'api récupérant les informations sur un user

• Créer un service api

```
# cd app
# mkdir lib
# cd lib
# mkdir services
# cd services
# npm run ng generate service api
```

• Comme ce service sera global à l'application, le déclarer dans les **`providers`** de **`app.module.ts`**

```
...
import { ApiService } from './lib/services/api.service';
...
  providers : [
     ApiService
  ]
...
```

• Créer un fichier **`user.json`** dans le répertoire **`src/asset`**

```
{
  "id": 123,
  "nom": "Doe",
  "prenom": "John",
  "code" : "abc123def456",
  "role": "user",
  "dateNaissance": "23/05/2003"
}
```

• Importer **`HttpClienModule`** dans **`app.module.ts`** :

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

...
  imports: [
     ...
     HttpClientModule
  ]
...
```

• Importer **`HttpClient`** dans **`api.service.ts`**

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

• Ajouter une méthode « **`getUser`** » au service qui récupère les données user

```
getUser = () => {
  setTimeout( () => {
     this.http.get('assets/user.json').subscribe({
        next: result => { console.log(result); },
        error: err => { console.error('Error: ' + err); },
        complete: () => { console.log('call ended'); }
     });
  },2000); //on impose un setTimeout pour simuler un réseau très lent
}
```

12) Créer un store global à l'application pour y stocker les données globales

• Créer un service store :

```
# cd lib/services
# npm run ng generate service store
```

• Modifier le service en déclarant qu'on y stockera des données et un user

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class StoreService {

  data: any = {}
  user: any = {}

  constructor() { }
}
```

• Déclarer le service dans les providers de **`app.module.ts`**, comme il a été fait pour ApiService
• Importer **`StoreService`** dans **`ApiService`**, afin de stocker le résultat d'un call api

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { StoreService } from './store.service';

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

  constructor(
     private http: HttpClient,
     private store: StoreService
  ) { }

  getUser = () => {
     setTimeout( () => {
        this.http.get('assets/user.json').subscribe({
           next: result => {
              this.store.user = result;
           },
```

```
          error: err => { console.error('Error: ' + err); },
          complete: () => { console.log('-- call ended --'); }
        });
    },2000); //on impose un setTimeout pour simuler un réseau très lent

  }
}
```

13) Utiliser le StoreService dans le header

Nous souhaitons récupérer les informations de l'utilisateur à la première connexion, et afficher son nom dans le header de l'application.

• Importer le service dans le composant **`header.component.ts`**

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../lib/services/api.service';
import { StoreService } from '../lib/services/store.service';

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent implements OnInit {

  constructor(
     private api: ApiService,
     public store: StoreService
  ) {}

  ngOnInit() {
     this.api.getUser();
  }
}
```

• Utiliser directement le store pour afficher le nom et le prénom de l'utilisateur dans le header, et donc insérer le code suivant dans le composant « navbar » de **`header.component.html`**

```
<div class="me-auto">
  Bonjour
  <span>{{store.user?.prenom}}</span>
  <span>{{store.user?.nom}}</span>
</div>
```

*Notez qu'aucune création de variable locale n'est nécessaire, pas plus que des **`@Input`** ou **`@Output`**.*

14) Afficher les caractéristiques du user aussi sur la page « news »

• Importer le **`StoreService`** dans le composant **`news.component.ts`**

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

```
import { StoreService } from '../../lib/services/store.service';

@Component({
  selector: 'app-news',
  templateUrl: './news.component.html',
  styleUrls: ['./news.component.css']
})
export class NewsComponent implements OnInit {

  constructor(
     private store: StoreService
  ) { }

  ngOnInit() {}
}
```

• Modifier le **`template news.component.html`**

```
<div>
  User: {{store.user.prenom}} {{store.user.nom}}
</div>
```

Notez qu'aucune création de variable locale n'est nécessaire, pas plus que des ***`@Input`*** *ou* ***`@Output`***.

15) Appeler la liste des news et la rendre disponible

Comme nous avons récupéré les caractéristiques du user, nous allons récupérer la liste des news, et la stocker dans store.data. Le composant **`NewsComponent`** devra être :

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../../lib/services/api.service';
import { StoreService } from '../../lib/services/store.service';

@Component({
  selector: 'app-news',
  templateUrl: './news.component.html',
  styleUrls: ['./news.component.css']
})
export class NewsComponent implements OnInit {
  constructor(
     private api: ApiService,
     public store: StoreService
  ) { }
  ngOnInit(): void {
     this.api.getNews();
  }
}
```

Et bien sûr il faudra ajouter la méthode **`getNews()`** au service api :

```
getNews = () => {
  setTimeout(() => {
     this.http.get('assets/news.json').subscribe({
        next: result => {
           this.store.data.news = result;
        },
        error: err => { console.error('Error: ' + err); },
```

```
      complete: () => { console.log('-- call ended --'); }
    });
  },2000); //on impose un setTimeout pour simuler un réseau très lent
}
```

On note qu'on résout directement l'asynchronisme au retour de l'api, dans store.data.et ce sont ces données du store que nous utiliserons dans l'application.

16) Afficher les news

Il nous faudra d'abord créer le fichier **`src/assets/news.json`** :

```
[
    {
        "id": 123,
        "titre": "Tintin au Tibet",
        "resume": "Un jeune reporter se lance à la recherche du Yeti avec le capitaine
Haddock",
        "auteur": "Hergé"
    },
    {
        "id": 345,
        "titre": "Asterix le gaulois",
        "resume": "Les aventures d'un guerrier Gaulois possédant une potion magique
rendant invincible",
        "auteur": "Gosigny"
    },
    {
        "id": 678,
        "titre": "le Capital",
        "resume": "Etude économique sur le travail et le capital",
        "auteur": "K. Marx"
    },
    {
        "id": 910,
        "titre": "Mécanique Analytique",
        "resume": "Les bases de la physique classique",
        "auteur": "J.L. Lagrange"
    },
    {
        "id": 112,
        "titre": "La byciclette Bleue",
        "resume": "Je sais pas, je l'ai pas lu",
        "auteur": "Deforge"
    }
]
```

Il faut maintenant afficher la liste des news. Comme cette liste est asynchrone, nous utiliserons le tag **<ng-container *ngIf=="condition">** pour gérer l'asynchronisme. Rien de ce qui est contenu dans ce container ne sera ni affiché, ni évalué, tant que la condition ne sera pas remplie. Ce tag est invisible, il n'a aucune influence sur la UI.

• Pour pouvoir utiliser la directive **ngIf** et **ngFor**, il faut au préalable avoir importé le **CommonModule** dans un module parent, pour nous le **AppModule** :

```
...
import { CommonModule } from '@angular/common';
...
  imports: [
...
    CommonModule
  ],
...
```

• Modifier **news.component.html** pour afficher la liste des documents, nous utilisons ici les facilités de Bootstrap pour le rendu des tables :

```
<ng-container *ngIf="store.data.news">
  <table class="table">
     <thead>
        <tr>
           <th scope="col">#</th>
           <th scope="col">Titre</th>
           <th scope="col">Auteur</th>
           <th scope="col">Résumé</th>
        </tr>
     </thead>
     <tbody>
        <tr *ngFor="let item of store.data.news">
           <th scope="row">{{item.id}}</th>
           <td>{{item.titre}}</td>
           <td>{{item.auteur}}</td>
           <td>{{item.resume}}</td>
        </tr>
     </tbody>
  </table>
</ng-container>
```

17) Lazy loading de la page « news »

Nous souhaitons que la section News de l'application soit montée en lazy loading, car certains utilisateurs ne la consulte jamais, et il est donc inutile d'en charger les codes, sauf à réduire les performances de l'interface utilisateur. Seuls les utilisateurs qui cliqueront sur « news » chargeront cette partie de l'application. Nous allons aussi équiper cette nouvelle route d'une possibilité de routage qui lui est uniquement dédiée, en considérant que l'utilisateur aura bien sûr accès au composant « news » que nous avons créé, mais aussi à d'autres composants.

La première chose à faire est de créer un module pour cette page.

• Se placer dans le répertoire de la page « news » et créer un module pour cette page :

```
# cd pages/news
# npm run ng g module news
# mv news/news.module.ts . <<< on place le module directement dans le répertoire
                             news.
# rm -r news <<< on détruit le répertoire devenu inutile
```

• Se placer ensuite dans le répertoire de la page « news » et créer un module de routage :

```
# cd pages/news
# npm run ng g module news-routing
# mv news-routing/news-routing.module.ts . <<< on place le module de routage
                                               directement
dans le répertoire news.
# rm -r news-routing <<< on détruit le répertoire devenu inutile
```

• Embarquer le **NewsRoutingModule** dans le **NewsModule** en modifiant **news.module.ts**

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { NewsRoutingModule } from './news-routing.module';
@NgModule({
  declarations: [],
  imports: [
     CommonModule,
     NewsRoutingModule
  ]
})
export class NewsModule { }
```

• Modifier **news-routing.module.ts** pour embarquer **NewsComponent**

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Routes, RouterModule } from '@angular/router';
import { NewsComponent } from './news.component';

const routes: Routes = [
  {
     path: '',
     component: NewsComponent
  },
  {
     path : '**',
     redirectTo : '',
     pathMatch: 'full'
  }
]
```

```
@NgModule({
  declarations: [
     NewsComponent
  ],
  imports: [
     RouterModule.forChild(routes),
     CommonModule
  ]
})
```

```
export class NewsRoutingModule { }
```

*Notez le routing « **forChild** » au lieu de « **forRoot** », nécessaire pour une route secondaire en lazy loading, car nous ne sommes plus dans le routage principal, mais secondaire.*

• Modifier **app-routing.module.ts** pour lui indiquer que la route « news » sera en lazy loading,et qu'il n'a plus besoin d'embarquer **NewsComponent**

```
// import { NewsComponent } from './pages/news/news.component';
...
//  {
//     path: 'news',
//     component: NewsComponent
//  },
{
    path: 'news',
    loadChildren: () => import('./pages/news/news.module')
       .then((mod : any) => mod.NewsModule)
},
...
```

18) Créer une page profile en lazy loading

On souhaite pouvoir modifier le nom de l'utilisateur dans sa page profile. Comme c'est une page peu utilisée, on la monte aussi en lazy loading. Répétez les opérations de créations de la page News pour obtenir la page « Profile ». Y importer le **StoreService.**

19) Intégrer un formulaire à la page profile

Puisque nous allons utiliser les tags HTML « input » sur cette page, il faut commencer par embarquer le module standard d'Angular **FormsModule** dans le module du lazy loading, c'est-à-dire l'importer dans **profile-routing.module.ts** :

```
import { FormsModule } from '@angular/forms';
...
  imports: [
     ...
     FormsModule
  ]
```

Choisir un code exemple dans la documentation Bootstrap pour générer le formulaire. Par exemple :

```
<div class="text-align: center;">
  <div class="card" style="width: 18rem;">
     <div class="card-body">
        <h5 class="card-title">User</h5>
        <h6 class="card-subtitle mb-2 text-muted">Change name</h6>
        <div class="input-group">
           <span class="input-group-text">First name</span>
           <input type="text" [(ngModel)]="store.user.prenom" class="form-control">
        </div>
        <div class="input-group">
```

```
          <span class="input-group-text">Last name</span>
          <input type="text" [(ngModel)]="store.user.nom" class="form-control">
        </div>
      </div>
  </div>
</div>
```

Notez que les input sont reliés à la variable ***store.user.nom*** *et* ***store.user.prenom*** *définies dans le service* ***StoreService****, que la page profile importe. The two-way data binding est procuré par le banana operator* ***[(ngModel)]***.

Vérifier qu'une modification dans les input se répercute immédiatement dans le header, et partout ailleurs dans l'application.

20) Sauvegarder l'état de l'application sur un serveur.

Puisque nous ne possédons pas de serveur, nous simulerons cette opération. Pour commencer il nous faut une api supplémentaire, et nous modifions donc le ApiService en lui ajoutant :

```
saveAppStatus() {
  console.log('Saving on server:',
     {user: this.store.user, data: this.store.data}, time: new Date()}
  );
  /*
     //example of post request to a server api
     this.http.post('myserver_URL', payload).subscribe({
        next: result => { console.log('Backup done'); },
        error: err => { console.error('Error: ' + err); },
        complete: () => { console.log('-- call ended --'); }
     });
  */
}
```

Ensuite nous allons demander à Angular d'appeler cette api à chaque fois que le contenu d'un input est modifié. Nous modifions donc le template HTML du composant **`profile.component.html`** :

```
<div class="text-align: center;">
  <div class="card" style="width: 18rem;">
     <div class="card-body">
        <h5 class="card-title">User</h5>
        <h6 class="card-subtitle mb-2 text-muted">Change name</h6>
        <div class="input-group">
           <span class="input-group-text">First name</span>
           <input type="text" [(ngModel)]="store.user.prenom"
              class="form-control" (change)="api.saveAppStatus()">
        </div>
        <div class="input-group">
           <span class="input-group-text">Last name</span>
           <input type="text" [(ngModel)]="store.user.nom"
              class="form-control" (keyup)="api.saveAppStatus()">
        </div>
     </div>
  </div>
</div>
```

Observer que chaque modification d'un input déclenche bien l'appel à l'api, notamment dans le second input taggué « keyup ».

Notez qu'à tout instant vous pouvez empiler sur le serveur les stores de votre application dans des états successifs, puis les réinstaller instantanément :

```
this.http.post('myserver/getStoreHistory', { backupId: 123}).subscribe((store: any) =>
{
    this.store.user = store.user;
    this.store.data = store.data;
});
```

Ceci est pratique dans des applications nécessitant la fonctionnalité back/forward, comme des formulaires sur plusieurs pages.

21) Intégration de PrimeNg

En javascript il existe de nombreuses bibliothèques de composant mais aucune qui satisfassent à tous vos besoins. Soit que le composant recherché est absent, soit qu'il ne convient pas, soit qu'il vous oblige à coder à sa façon plutôt que la votre. Aucun choix ne sera parfait, alors nous conseillons PrimeNg puisqu'il faut bien en choisir un. En tout état de cause l'installation et l'utilisation d'une telle bibliothèque externe reviendra à peu près au même.

D'abord on installe le package prime, en version 14 :

```
npm install primeng@^14 primeicons --save
```

Ensuite on édite le fichier angular.json pour lui ajouter :

```
"styles": [
"node_modules/primeicons/primeicons.css",
"node_modules/primeng/resources/themes/lara-light-blue/theme.css",
"node_modules/primeng/resources/primeng.min.css",
...
]
```

PrimeNg fonctionne par modules. Par exemple, si vous avez besoin d'un bouton, vous devrez déclarer l'import du module **`ButtonModule`** de PrimeNg dans le module support de votre composant. Ajoutons par exemple ce bouton à la page Profile.

Importons le **`ButtonModule`** dans le **`ProfileRoutingModule`** :

```
...
import { ButtonModule } from 'primeng/button';
...
  imports: [
     …
     ButtonModule
  ]
...
```

Et dans le template HTML de la page profile on insère :

```
<p-button label="Save on server" (click)="api.saveAppStatus()"></p-button>
```