UNIVERSITÉ DE RENNES 1

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

Projet JXC (Angular)
FR-Administration

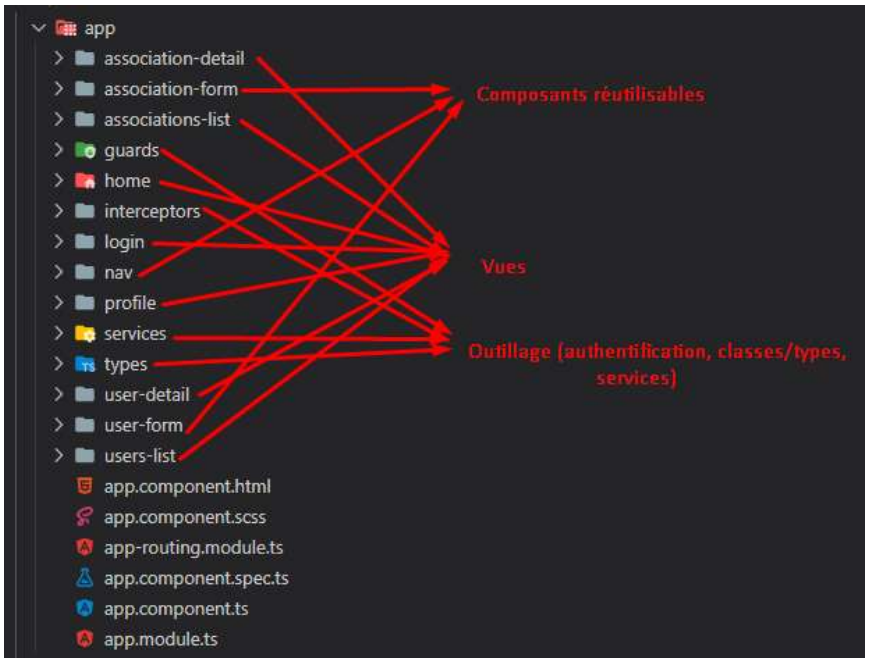
Léo Rolland et Nicolas Poirier

Enseignante : Hélène Feuillâtre
Lien du dépôt : https://github.com/leorolland/esir-tp-fr-administration-front

Année 2021 - 2022
ESIR 3 - Technologies de l'Information Spécialité Systèmes d'Information

Architecture globale

Nous avons choisi un nommage pertinent pour regrouper les composants qui concernent la même section.
NB : Dans une version plus avancée, on aurait pu créer un module par section, et effectuer du lazy-loading sur les modules pour ne pas charger toute l'application au premier chargement de l'app.



Composants réutilisables

Plutôt que de dupliquer certaines parties du code pour plusieurs usecase (et donc augmenter la quantité de code à maintenir, tester et le risque de bogue), nous avons choisi d'exploiter le principe de composant réutilisable d'Angular.
Certains composants sont réutilisés dans plusieurs vues, c'est le cas notamment du composant "user-form", qui sert

- Pour éditer les informations de son profil
- Pour créer un nouvel utilisateur
- Pour éditer les informations d'un utilisateur existant

En effet, en fonction des @Input passés aux composant, ce dernier se comporte de manière différente, mais en utilisant le même template d'affichage, voici un exemple :

```
@Input() id: number | undefined;

@Input() stayOnPage: boolean | undefined;
```

```
ngOnInit(): void {
  // If no input is passed, check in route
  if (!this.id) {
    this.route.paramMap.subscribe( res => {
      this.id = + (res.get('id') || -1)
      if (this.id ≥ 0) this.load(this.id)
    })
  } else {
    this.load(this.id)
  }
}
```

Keep It Simple Stupid

Nous essayons de privilégier l'utilisation des Observables de bout en bout (de la requête HTTP à l'affichage dans le template) pour que tout soit homogène, tout en utilisant le moins de code possible (moins de code = moins de bugs).

users.service.ts

```ts
getById(id: number): Observable<User> {
  return this.http.get<User>(`${BASE_URL}/users/id/${id}`)
}
```

user-detail.component.ts

```ts
user: Observable<User> | undefined

constructor(private usersService: UsersService,

ngOnInit(): void {
  this.route.paramMap.subscribe( res => {
    const id = + (res.get('id') || 0)
    this.user = this.usersService.getById(id)
  })
}
```

user-detail.component.html

```html
<mat-card *ngIf="user | async as u">
    <mat-card-subtitle>id: {{ u.id }}</mat-card-subtitle>
    <mat-card-title>{{ u.firstname }} {{u.lastname}}</mat-card-title>
    {{ u.age }} ans
</mat-card>
<div *ngIf="!user">
    Loading user ...
</div>      You, a week ago • feat(user details): page user details
```

Également, nous réutilisons des fonctionnalités déjà implémentées dans notre librairie de composants, telle que la prise en charge d'observable comme source de donnée, ainsi que le filtrage.

```ts
  this.usersServic
}

applyFilter(event:
  const filterValu
  this.dataSource.filter = filterValue.trim().toLowerCase();
}
```

(property) _MatTableDataSource<unknown, MatPaginator>.filter: string

Filter term that should be used to filter out objects from the data array. To override how data objects match to this filter string, provide a custom function for filterPredicate.

Des règles de routage claires, et explicites

Les routes suivent la convention de nommage REST (bien que cette convention soit normalement orientée backend). A l'exception près que l'on ajoute "/edit" plutôt que de faire une requête PUT/PATCH pour éditer une entité.

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'users', component: UsersListComponent, canActivate: [AuthGuard] },
  { path: 'users/create', component: UserFormComponent, canActivate: [AuthGuard] },
  { path: 'users/:id', component: UserDetailComponent, canActivate: [AuthGuard] },
  { path: 'users/:id/edit', component: UserFormComponent, canActivate: [AuthGuard] },
  { path: 'associations', component: AssociationsListComponent, canActivate: [AuthGuard] },
  { path: 'associations/:name', component: AssociationDetailComponent, canActivate: [AuthGuard] },
  { path: 'associations/:name/edit', component: AssociationFormComponent, canActivate: [AuthGuard] },
  { path: 'profile', component: ProfileComponent, canActivate: [AuthGuard] },
  { path: '**', redirectTo: 'home' },
];
```

Cela facilite la compréhension de l'utilisateur, le partage de liens (copier coller l'url emmène toujours au bon endroit), et permet de faciliter le code de la navigation :

```
<button mat-button routerLink="..">Retour</button>
```

Ergonomie et Design :

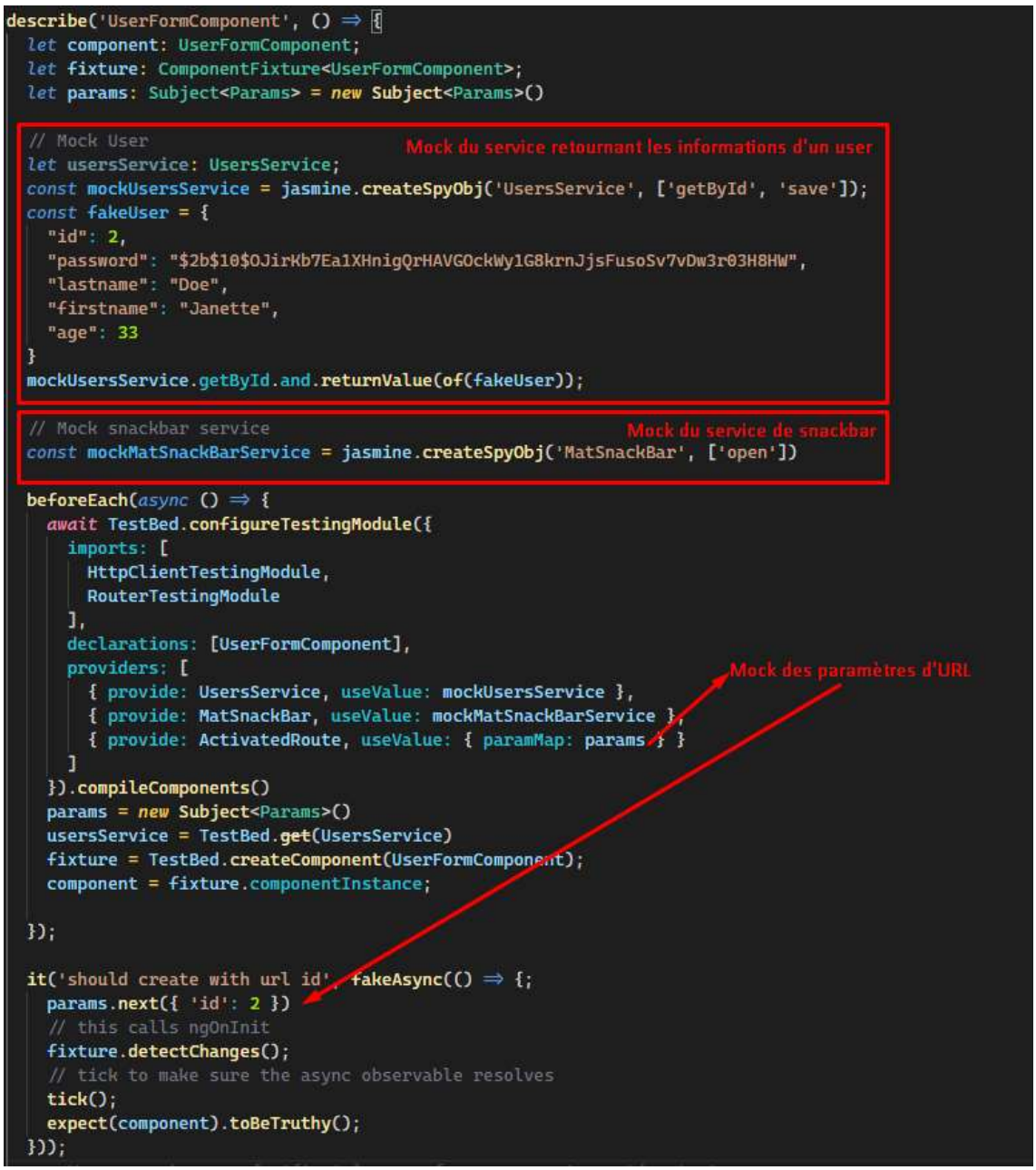
Nous avons utilisé Sass pour coder les différents styles de nos composants, et contenus. Parallèlement nous avons utilisé certains composants d'angular material dont le design est sûrement plus agréable que nos potentielles productions. Nous n'avons pas utilisé de collection supplémentaire pour le design comme Bootstrap. Nous estimons qu'un CSS bien utilisé est plus intéressant que l'import d'une ou plusieurs de ces collections. Bien que notre niveau de CSS soit limité, nous sommes persuadés que la pratique puis la maîtrise d'un tel langage permet une liberté bien plus grande pour notre créativité que les collections dans lesquelles les personnalisations de composants sont plus complexes.

```
a:after {
  background: none repeat scroll 0 0 transparent;
  content: "";
  display: block;
  height: 2px;
  left: 50%;
  background: linear-gradient(to right, rgb(0, 85, 164), rgb(255, 255, 255) 50%, rgb(239, 65, 53) 100%);
  transition: width 0.3s ease 0s, left 0.3s ease 0s;
  width: 0;
}

a:hover:after {
  width: 100%;
  left: 0;
}

.active a:after{
  background-color:linear-gradient(to right, rgb(0, 85, 164), rgb(255, 255, 255) 50%, rgb(239, 65, 53) 100%);
  height:2px;
  width:100%;
}
```

Exemple d'animation sur les liens sans utiliser de collection (type bootstrap ou material)

Tests unitaires

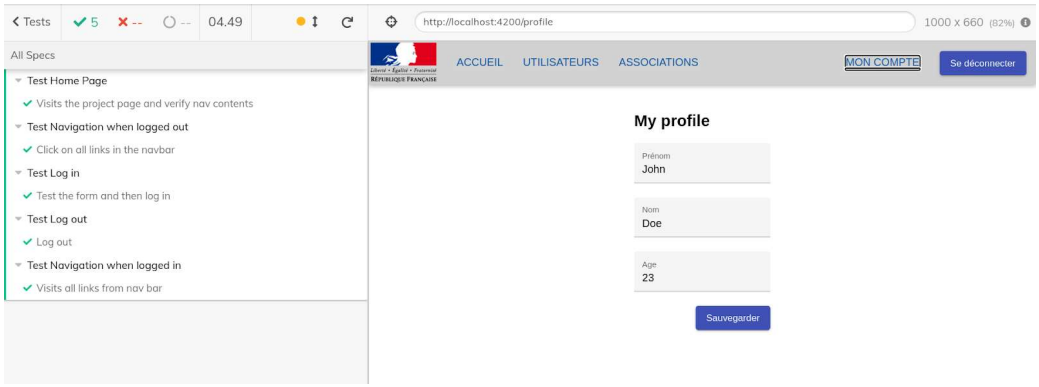
Les composants réutilisables sont des points critiques de l'application puisqu'ils servent pour plusieurs usages, il convient de les tester leur comportement en couvrant tout l'espace d'@Input possible. Ainsi, on peut les réutiliser en ayant entièrement confiance en leur comportement.




```
describe('UserFormComponent', () => {
  let component: UserFormComponent;
  let fixture: ComponentFixture<UserFormComponent>;
  let params: Subject<Params> = new Subject<Params>()

  // Mock User
  let usersService: UsersService;
  const mockUsersService = jasmine.createSpyObj('UsersService', ['getById', 'save']);
  const fakeUser = {
    "id": 2,
    "password": "$2b$10$OJirKb7Ea1XHnigQrHAVGOckWy1G8krnJjsFusoSv7vDw3r03H8HW",
    "lastname": "Doe",
    "firstname": "Janette",
    "age": 33
  }
  mockUsersService.getById.and.returnValue(of(fakeUser));

  // Mock snackbar service
  const mockMatSnackBarService = jasmine.createSpyObj('MatSnackBar', ['open'])

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        HttpClientTestingModule,
        RouterTestingModule
      ],
      declarations: [UserFormComponent],
      providers: [
        { provide: UsersService, useValue: mockUsersService },
        { provide: MatSnackBar, useValue: mockMatSnackBarService },
        { provide: ActivatedRoute, useValue: { paramMap: params } }
      ]
    }).compileComponents()
    params = new Subject<Params>()
    usersService = TestBed.get(UsersService)
    fixture = TestBed.createComponent(UserFormComponent);
    component = fixture.componentInstance;

  });

  it('should create with url id', fakeAsync(() => {;
    params.next({ 'id': 2 })
    // this calls ngOnInit
    fixture.detectChanges();
    // tick to make sure the async observable resolves
    tick();
    expect(component).toBeTruthy();
  }));
```

Test End 2 End :

Pour réaliser nos tests E2E nous avons utilisé Cypress qui permet de tester l'interface Homme Machine. Cypress suit un script de simulation de navigation sur notre site et permet potentiellement d'identifier des erreurs dans l'implémentation du site. C'est un outil très utile car la simulation de l'utilisation de l'IHM est bien plus rapide que manuellement. De plus, en exécutant le script après chaque nouvelle modification, on peut tester l'ensemble du site. On envisage néanmoins des limites dans l'utilisation de cet outil : il faut penser à toutes sortes d'interactions pour avoir le test le plus complet. Cela permet de correctement tester uniquement une utilisation normale du site.



Repository Git :

La gestion de notre repository est assez classique : nous avons suivi un équivalent à Gitflow: nous développons sur des branches de features pour les nouvelles fonctionnalités ou de fix pour les bugs après une release. Une fois ces corrections ou fonctionnalités terminées, nous les mergions sur la branche dev. Enfin nous mergions la branche dev sur main pour générer une nouvelle release, c'est-à-dire une version complète, fonctionnelle (et supposée sans bugs connus). Nos branches respectaient un format dans le nommage et dans le message de commit : les branches sont préfixées par leur type suivant d'un slash (par exemple : *feat/monnomdebranche*), les commits commencent également par leur type, puis leur sujet, puis la description des éléments modifiés (par exemple: *feat(mafonctionnalité): détails de l'étape X)*. Ces contraintes nous ont permis d'utiliser un bot pour la génération des commentaires lors des releases (voir le résultat ci-dessous), le bot permet aussi de clore ou commenter une issue du projet. Nous avons également mis en place sur notre repository une CI afin de tester le code envoyé sur la branche master, de vérifier qu'il fonctionne, c'est aussi dans cette CI qu'était appelé le bot qui génère des descriptions de release plus complètes.