

Routage

Single Page Application

- Historiquement, les sites webs étaient constitués d'une multitude de pages, avec une architecture qui ressemble à celle d'un répertoire de fichiers
- Le problème de cette approche est que une nouvelle page entière doit être chargée dès que l'utilisateur change de page, ce qui peut être lent
- Pour résoudre ce problème, les SPA (Single Page Application) ont vu le jour
- Le but d'une SPA est d'avoir une page unique, qui est modifiée dynamiquement par le navigateur, sans avoir à charger une nouvelle page, avec éventuellement des données récupérées depuis le serveur
- Le principe du routage dans une SPA permet d'associer des URL à des états de l'application. Il va permettre à l'utilisateur de naviguer sur l'application grâce à l'URL, comme il le ferait sur un site classique

Routage

- Dans une application Angular, un service gère le routage. Comme le HttpClient, il n'est pas activé par défaut
- En général, on crée un module à part :

```
const routes: Routes = [  
  { path: 'colis', component: ColisComponent },  
  { path: 'livraisons', component: LivraisonsComponent }  
]  
  
@NgModule({  
  imports: [RouterModule.forRoot(routes)],  
  exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

- Le RouterModule associe des composants à des URLs

Routage

- Il faut indiquer à Angular où insérer le composant, pour cela, on utilise une balise spécifique dans le template :

```
<router-outlet></router-outlet>
```

- RouterOutlet est une directive, qui va être remplacée par le module de routage d'Angular par le composant correspondant à la route

Routes

- Le type Routes est un tableau de Route

```
const routes: Routes = [  
  { path: 'colis', title: 'Colis', component: ColisComponent },  
  { path: 'livraisons', title: 'Livraisons', component: LivraisonsComponent },  
  { path: '', redirectTo: '/colis', pathMatch: 'full' },  
  { path: '**', component: PageNotFoundComponent },  
]
```

- Le router va essayer de matcher l'URL avec ses paths, dans l'ordre du tableau
- title permet de changer le titre de la page pour une route spécifique
- pathMatch: 'full' indique que l'URL complète doit matcher avec le path de la route, l'autre valeur (par défaut est 'prefix')
- ** est une wildcard qui va matcher avec toutes les URLs

Navigation

- Il est possible de faire des liens classiques dans l'application pour naviguer, mais dans ce cas, la page est rechargée, ce qui annule l'intérêt du routage
- Angular met à disposition la directive `routerLink` pour naviguer dans l'application
- Une autre directive, `routerLinkActive` permet d'associer une classe à un élément si le lien pointe sur la route courante

```
<button routerLink="/colis" routerLinkActive="selected-item">Gestion des colis</button>
```

- Dans un composant, on peut injecter le service Router et utiliser la méthode `navigate()` pour rediriger l'utilisateur

```
router = inject(Router)  
...  
this.router.navigate(['/colis']);
```

- Un `/` dans l'URL de navigation indique un chemin absolu

Découpage modulaire et lazy-loading

```
app/  
  feature/  
    feature.module.ts  
    feature.routing.module.ts  
  other-feature/  
    .../  
    other-feature.module.ts  
    other-feature.routing.module.ts  
app.module.ts  
app.routing.module.ts
```

- Dans une application modulaire, le module principal gère le routing jusqu'à chaque feature et chaque module feature gère son propre sous-routing
- En général les features modules sont lazy-loadés
- ⚠ La configuration des sous-routes est différente si le module est lazy-loadés

FeatureRoutingModule (pas de lazyload)

```
const routes: Routes = [
  {
    path: 'colis',
    title: 'Colis',
    component: ColisComponent,
  }
]

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

- Chaque feature module à son propre sous module de routage
- La seule différence avec le routing principal est l'utilisation de `forChild()` au lieu de `forRoot()` dans les imports
- Dans ce cas, les routes sont au même niveau que celles du routeur principal

Child routes

- L'objet Route possède un paramètre children qui est un tableau de Route, il est donc possible de définir une arborescence de routes :

```
const routes: Routes = [  
  {  
    path: 'colis',  
    title: 'Colis',  
    component: ColisComponent,  
    children: [  
      { path: 'details', component: ColisDetailsComponent },  
    ],  
  },  
  ...  
]
```

- Pour que les routes filles soient utilisées, le composant ColisComponent doit avoir lui aussi un appel à la directive RouterOutlet dans son template

Lazy-loading

- Lorsque l'application devient plus complexe, charger toute l'application en une seule fois peut devenir long
- Il devient nécessaire de découper l'application en plusieurs sous-parties, qui seront téléchargées uniquement lorsque elles sont nécessaires
- Ce principe s'appelle le lazy-loading
- Le lazy-loading a l'avantage de réduire le chargement initial de l'application, mais oblige le client à faire des appels réguliers vers le serveur pour récupérer les parties manquantes

AppRoutingModule

```
const routes: Routes = [  
  {  
    path: 'colis',  
    title: 'Colis',  
    loadChildren: () => import('../features/colis/colis.module').then(m => m.ColisModule)  
  },  
  {  
    path: 'livraisons',  
    title: 'Livraisons',  
    loadChildren: () => import('../features/livraisons/livraisons.module').then(m => m.LivraisonsModule)  
  }  
]
```

- Dans le AppRoutingModule, on ne charge plus des composants, mais des modules
- ⚠ Attention à ne pas importer en plus les features modules dans le AppModule

FeatureRoutingModule (lazyload)

```
const routes: Routes = [
  {
    path: '',
    title: 'Colis',
    component: ColisComponent,
    children: [
      { path: 'details', component: ColisDetailsComponent },
    ]
  }
]

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

- Le path utilisé pour chargé le module est réutilisé en préfixe

Paramètres

- Il est possible d'avoir des routes avec des paramètres dans l'URL
- Par exemple pour consulter un colis en particulier

```
{ path: ':colisId', component: ColisDetailComponent },
```

- Le composant peut récupérer les paramètres d'url avec le service ActivatedRoute :

```
router = inject(ActivatedRoute)  
  
...  
  
id = route.snapshot.paramMap.get('colisId')!;
```

- Snapshot représente un instantané de l'état de la route


Paramètres

- Lorsque l'on navigue avec le router, Angular ne recrée pas un composant si les paramètres changent
- Dans ce cas, le snapshot n'est pas adapté car il ne reflètera pas le changement de paramètre
- Mais il est possible de récupérer les paramètres sous forme d'observable :

```
route = inject(ActivatedRoute)

constructor() {
  route.paramMap.subscribe((params: ParamMap) => {
    const id = params.get('colisId');
  });
}
```

Guards

- Les guards permettent de restreindre l'accès à certaines parties de l'application
- Il existe plusieurs types de guards, nous allons voir le plus simple `canActivate`, qui permet de restreindre l'accès à une route
-  Les Guards ne dispensent pas de faire un contrôle d'accès au niveau du serveur backend

CanActivate (déprécié)

- Les guards sont définis sous la forme de service
- Angular fournit une interface CanActivate, qui définit ce que notre service doit implémenter pour pouvoir être utilisé comme guard canActivate
- Il s'agit d'une méthode : canActivate()

```
@Injectable()
class CanActivateLivraisons implements CanActivate {
    ...
    canActivate(
        route: ActivatedRouteSnapshot, state: RouterStateSnapshot
    ): MaybeAsync<GuardResult> {
        ...
    }
}
```


Guards fonctionnels

- L'implémentation des Guards sous forme de service est dépréciée depuis Angular 16
- De la même manière que les intercepteurs, il faut maintenant définir les guards de manière fonctionnelle :

```
export const isAuthenticatedGuard: CanActivateFn = (route, state) => {  
  const authenticationService = inject(AuthenticationService)  
  const router = inject(Router);  
  
  return authenticationService.getStatus()  
    ? true  
    : router.parseUrl("/connexion")  
}
```

CanActivateFn

```
type CanActivateFn = (route: ActivatedRouteSnapshot, state: RouterStateSnapshot) => MaybeAsync<GuardResult>
```

- La méthode canActivate() renvoie un objet de type MaybeAsync<GuardResult>

```
type MaybeAsync<T> = T | Observable<T> | Promise<T>
```

- Le retour d'un guard peut être synchrone ou asynchrone, dans le cas d'un retour asynchrone, le routeur attend une émission avant de continuer

```
type GuardResult = boolean | UrlTree
```

- Le retour d'un guard peut être soit un boolean, qui représente si l'utilisateur peut accéder à la route ou non. Au lieu de renvoyer false, le guard peut renvoyer une url pour rediriger l'utilisateur

Fin