INTRODUCTION AU ROUTAGE

OBJECTIF DU ROUTAGE

Le routage permet de **naviguer** entre les différentes **vues** et d'**organiser** les composants au sein d'une application Angular.

NAVIGATION ENTRE LES DIFFÉRENTES VUES

Le **routage** permet de transformer une application Angular en une application à **plusieurs pages**.

ORGANISATION DES COMPOSANTS

Le **routage** facilite l'organisation des composants et des **fonctionnalités** en les associant à des **URL**.

NGMODULE

Pour utiliser le **routage** en Angular, un **NgModule** est nécessaire.

DÉCLARATION DES ROUTES

Les routes sont déclarées dans un **NgModule** en tant que tableau d'objets.

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

const routes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'about', component: AboutComponent },
    { path: 'contact', component: ContactComponent },
    { path: '**', redirectTo: '' }
];

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

IMPORTATION DU ROUTERMODULE

Le **RouterModule** est importé et ajouté dans les `imports` du **NgModule** pour activer le **routage**.

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

@NgModule({
  imports: [
    RouterModule.forRoot(ROUTES)
  ]
})
export class AppModule { }
```

ROUTER-OUTLET

router-outlet est un élément qui indique à **Angular** où afficher le contenu du composant lié à la route.

SYNTAXE

Pour utiliser `router-outlet`, ajoutez cette balise dans le **template HTML** :

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

EMPLACEMENT DANS L'APPLICATION

Il est généralement placé dans le **composant principal (app.component)** de l'application.

CONFIGURATION DES ROUTES

SYNTAXE DES ROUTES

CHEMIN

Pour définir le **chemin** d'une route, utilisez la propriété `path` :

```
{ path: 'chemin', component: MonComposant }
```

COMPOSANT

Associez un composant à une route en utilisant la propriété `component` :

```
{ path: 'chemin', component: MonComposant }
```

REDIRECTION

Pour rediriger d'une route vers une autre, utilisez la propriété `**redirectTo**` :

```
{ path: '', redirectTo: '/autre-chemin', pathMatch: 'full' }
```

PARAMÈTRES DE ROUTE

SYNTAXE

Pour ajouter des **paramètres** à une route, utilisez `:nom_parametre` dans le champ `path` :

```
{ path: 'chemin/:id', component: MonComposant }
```

RÉCUPÉRATION DES PARAMÈTRES

Pour récupérer les paramètres de la route, utilisez `**ActivatedRoute**` :

```
constructor(private route: ActivatedRoute) {
    this.route.params.subscribe(params => {
        this.id = params['id'];
    });
}
```

ROUTES IMBRIQUÉES

SYNTAXE

Pour imbriquer des routes, utilisez la propriété `**children**` :

```
{
    path: 'parent', component: ParentComposant, children: [
        { path: 'enfant-1', component: Enfant1Composant }
    ]
}
```

UTILISATION DE "CHILDREN"

Les routes enfants sont déclarées avec la propriété `children` :

```
{
    path: 'parent', component: ParentComposant, children: [
        { path: 'enfant-1', component: Enfant1Composant },
        { path: 'enfant-2', component: Enfant2Composant }
    ]
}
```

EXEMPLES D'UTILISATION

Imaginons un scénario où vous avez une **page principale** avec un **panneau latéral** et plusieurs **sous-pages**.

```
{
    path: 'dashboard', component: DashboardComposant, children: [
        { path: 'page1', component: Page1Composant },
        { path: 'page2', component: Page2Composant }
    ]
}
```

NAVIGATION

ROUTERLINK

Le **routerLink** est une directive qui permet de créer une navigation entre les différents **composants** de l'application.

```
<a routerLink="/mon-composant">Naviguer vers Mon Composant</a>
```

SYNTAXE

Utiliser la directive **routerLink** avec les balises HTML pour la **navigation** :

```
<a routerLink="/chemin">Texte</a>
```

UTILISATION AVEC LES BALISES HTML

```
<nav>
  <a routerLink="/accueil">Accueil</a>
  <a routerLink="/contact">Contact</a>
</nav>
```

routerLink est une directive Angular qui permet de naviguer entre les différentes routes de l'application.

ROUTER.NAVIGATE

Le **Router.navigate** est une méthode pour naviguer entre les **composants** en utilisant du code au lieu des balises HTML.

```
import { Router } from '@angular/router';

constructor(private router: Router) {}

navigateToComponent(componentName: string) {
  this.router.navigate([componentName]);
}
```

SYNTAXE

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

Note : Cette syntaxe est utilisée pour naviguer vers une nouvelle route dans une application Angular. "chemin" doit être remplacé par le chemin de la route souhaitée.

NAVIGATION EN UTILISANT LE CODE

```
import { Router } from '@angular/router';

constructor(private router: Router) { }

allerAContact() {
  this.router.navigate(['/contact']);
}
```

Note: L'exemple montre comment naviguer vers une route spécifique ('/contact') en utilisant le service **Router** d'**Angular**. Cette approche est utile lorsque la navigation doit être déclenchée par un événement ou une action de l'utilisateur.

ACTIVATEDROUTE

L'**ActivatedRoute** est un service qui permet de récupérer des informations sur la route courante en cours de navigation.

SYNTAXE

```
import { ActivatedRoute } from '@angular/router';
```

RÉCUPÉRATION DES INFORMATIONS DE LA ROUTE COURANTE

```
constructor(private route: ActivatedRoute) { }

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

GUARDS

INTRODUCTION AUX GUARDS

Les **Guards** permettent de protéger l'accès aux routes en fonction de certaines conditions, comme l'authentification de l'utilisateur.

Types de Guards :

- **CanActivate**
- **CanDeactivate**
- **CanLoad**

CANACTIVATE

`**CanActivate**` décide si une **route** peut être activée.

```
import { CanActivate } from '@angular/router';

export class AuthGuard implements CanActivate {
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    // Votre logique pour déterminer si la route peut être activée
  }
}
```

SYNTAXE

Créez un service qui implémente l'interface `**CanActivate**` :

```
import { CanActivate } from '@angular/router';

@Injectable()
export class AuthGuard implements CanActivate { ... }
```

EXEMPLES D'UTILISATION

```
const routes: Routes = [
  { path: 'private', component: PrivateComponent, canActivate: [AuthGuard] },
];
```

CANDEACTIVATE

`CanDeactivate` décide si on peut **quitter** une route actuellement **active**.

Méthode de l'interface	Description
canActivate()	Retourne un `boolean` indiquant si on peut naviguer vers une autre route.
canActivateChild()	Détermine si l'accès à une route enfant est autorisé.

```
class CanDeactivateGuard {
  canDeactivate(component: Component, currentRoute: ActivatedRouteSnapshot, currentState: RouterStateSnapshot, nextState?: RouterSt
    // Code pour déterminer si le composant peut être désactivé
  }
}
```

SYNTAXE

Créez un service qui implémente l'interface `**CanDeactivate**` :

```
import { CanDeactivate } from '@angular/router';

@Injectable()
export class UnsavedChangesGuard implements CanDeactivate<T> { ... }
```

EXEMPLES D'UTILISATION

```
const routes: Routes = [
  { path: 'edit', component: EditComponent, canDeactivate: [UnsavedChangesGuard] },
];
```

CANLOAD

`CanLoad` décide si un module de **chargement différé** (**Lazy Loading**) peut être chargé.

```
import { CanLoad, Route } from '@angular/router';

@Injectable()
export class AuthGuardService implements CanLoad {

  constructor(private authService: AuthService) { }

  canLoad(route: Route): boolean {
    // Vérifie si l'utilisateur est authentifié
    const isAuthenticated = this.authService.isAuthenticated();
    if (isAuthenticated) {
      return true;
    } else {
      // Redirection vers la page de connexion
      this.authService.redirectToLogin();
```

SYNTAXE

Créez un service qui implémente l'**interface** `CanLoad` :

```
import { CanLoad } from '@angular/router';

@Injectable()
export class CanLoadGuard implements CanLoad { ... }
```

EXEMPLES D'UTILISATION

```
const routes: Routes = [
  {
    path: 'lazy',
    loadChildren: () => import('./lazy.module').then(m => m.LazyModule),
    canLoad: [CanLoadGuard],
  },
];
```

LAZY LOADING

INTRODUCTION

OBJECTIF

Le **Lazy Loading** permet de charger les modules **Angular** à la demande, lorsqu'ils sont nécessaires.

AVANTAGES

- **Améliore les performances** de l'application
- **Réduit le temps de chargement initial**

SYNTAXE

LOADCHILDREN

Utilisation de `loadChildren` pour définir les **modules** à **charger à la demande**.

```
const routes: Routes = [
  {
    path: 'section',
    loadChildren: () => import('./section/section.module').then(m => m.SectionModule)
  }
];
```

IMPORTATION DES MODULES

Importer les modules à utiliser avec le **Lazy Loading**.

EXEMPLES D'UTILISATION

```
// app-routing.module.ts
const routes: Routes = [
  {
    path: 'module-lazy',
    loadChildren: () => import('./module-lazy/module-lazy.module').then(m => m.ModuleLazyModule)
  }
];

// module-lazy-routing.module.ts
const routes: Routes = [
  {
    path: '',
    component: ModuleLazyComponent
  }
];
```

Dans cet exemple, le **ModuleLazy** sera chargé uniquement lorsque l'utilisateur naviguera vers le chemin '**module-lazy**'.

ROUTE RESOLVER

INTRODUCTION

OBJECTIF

Le **Route Resolver** permet de charger des **données** avant de naviguer vers une vue, ce qui évite d'afficher des données partielles ou manquantes.

AVANTAGES

- **Améliore l'expérience utilisateur**
- Simplifie la **gestion des erreurs** de chargement de données

SYNTAXE

`resolve`

- Implémenter l'**interface** `Resolve`
- Définir la **méthode** `resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot)`

INJECTION DE DÉPENDANCES

- Injecter les **services** nécessaires dans le **constructeur**
- Les utiliser dans la méthode `**resolve**`

```
import { Injectable } from '@angular/core';
import { Resolve } from '@angular/router';
import { MonService } from './mon-service.service';

@Injectable()
export class MonResolver implements Resolve<any> {

  constructor(private monService: MonService) { }

  resolve() {
    return this.monService.getData();
  }
}
```

EXEMPLES D'UTILISATION

IMPLÉMENTATION D'UN RESOLVER

```
import { Injectable } from '@angular/core';
import { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs';
import { DataService } from './data.service';

@Injectable({
  providedIn: 'root'
})
export class DataResolver implements Resolve<Observable<string>> {
  constructor(private dataService: DataService) {}

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.dataService.loadData();
  }
}
```

DÉCLARATION DU RESOLVER DANS LES ROUTES

```
const routes: Routes = [
  {
    path: 'data',
    component: DataComponent,
    resolve: { data: DataResolver }
  }
];
```

Note: Le resolver `DataResolver` est utilisé pour **récupérer** et **préparer** les données nécessaires pour le composant `DataComponent` avant de charger la route.

RÉCUPÉRATION DES DONNÉES RÉSOLUES DANS LE COMPOSANT

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

@Component({
  selector: 'app-data',
  templateUrl: './data.component.html',
  styleUrls: ['./data.component.scss']
})
export class DataComponent implements OnInit {
  data: string;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.data = this.route.snapshot.data['data'];
  }
```

STRATÉGIES DE NAVIGATION

STRATÉGIES DE NAVIGATION

Il existe **deux stratégies principales** pour contrôler la navigation dans une application **Angular** :

- Stratégie de **hachage**
- Stratégie de **chemin**

STRATÉGIE DE HACHAGE

La stratégie de hachage utilise des URLs contenant un `#`. L'avantage est que les URLs fonctionnent sur tous les serveurs, même ceux qui ne sont pas configurés pour le **routage côté serveur**.

SYNTAXE

Pour utiliser la stratégie de **hachage**, il faut l'**importer** et l'**ajouter** au `RouterModule` dans le fichier `app.module.ts`.

```
import { HashLocationStrategy, LocationStrategy } from '@angular/common';

@NgModule({
  // ...
  providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],
  // ...
})
export class AppModule { }
```

EXEMPLES D'UTILISATION

Avec la **stratégie de hachage**, les URLs auront le format suivant :

- http://example.com/#/route1
- http://example.com/#/route2

STRATÉGIE DE CHEMIN

La stratégie de chemin utilise des **URLs propres**, sans `#`, qui sont plus lisibles et mieux référencées par les moteurs de recherche. Cependant, cette approche nécessite une configuration appropriée du **serveur** pour la prise en charge du **routage côté serveur**.

SYNTAXE

La **stratégie de chemin** est la stratégie par défaut d'Angular, il n'est pas nécessaire de l'importer explicitement. Pour l'utiliser, il suffit de ne rien spécifier dans les options du **RouterModule**.

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

EXEMPLES D'UTILISATION

Avec la **stratégie de chemin**, les URLs auront le format suivant :

- http://example.com/route1
- http://example.com/route2