PoiemaWeb

13.24 Angular Routing & Navigation
라우팅과 내비게이션

#1. SPA (Single Page Application)

단일 페이지 애플리케이션(Single Page Application, SPA)은 모던 웹의 패러다임으로 네이티브 앱과 유사한 사용자 경험(UX)을 제공할 수 있다는 장점이 있다. SPA는 기본적으로 단일 페이지로 구성되며, 기존의 서버 사이드 렌더링과 비교할 때 배포가 간단하다.

a 요소를 사용하는 전통적인 웹 방식은 새로운 페이지 요청이 있을 때마다 정적 리소스가 다운로드되고 전체 페이지를 다시 렌더링하는 방식으로 동작하므로 새로고침이 발생한다. 이것은 변경이 필요 없는 부분을 포함하여 전체 페이지를 갱신하는 것이어서 비효율적이다.

반면에 SPA는 웹 애플리케이션에 필요한 모든 정적 리소스를 애플리케이션이 최초로 기동할 때 한 번만 다운로드한다. 이후 새로운 페이지 요청 시, 페이지 갱신에 필요한 데이터만 전달받기 때문에 전체적인 트래픽을 감소시킬 수 있다. 또한, 페이지 전체를 다시 렌더링하지 않고 변경이 필요한 부분만 갱신하므로 새로고침이 발생하지 않아 네이티브 앱과 유사한 사용자 경험을 제공할 수 있다.

모바일의 사용이 증가하고 있는 현시점에 트래픽의 감소와 속도, 사용성, 반응성의 향상은 매우 중요한 이슈이다. SPA의 핵심 가치는 사용자 경험 향상에 있으며 부가적으로 애플리케이션 속도의 향상도 기대할 수 있어서 '모바일 퍼스트(Mobile First)' 전략에 부합한다.

모든 소프트웨어 아키텍처에는 트레이드오프(trade-off, 두 개의 정책목표 가운데 하나를 달성하려고 하면 다른 목표의 달성이 늦어지거나 희생되는 경우를 의미한다.)가 존재하며 모든 애플리케이션에 적합한 은 탄환(Silver bullet, 고질적인 문제를 단번에 해결할 수 있는 명쾌한 해결책을 의미한다.)은 없듯이, SPA 또한 구조적인 단점이 있다. 대표적인 단점은 아래와 같다.

초기 구동 속도
SPA는 웹 애플리케이션에 필요한 모든 정적 리소스를 애플리케이션 최초 기동 시에 모두 다운로드하기 때문에 초기 구동 속도가 상대적으로 느리다. 하지만 SPA는 웹페이지보다는 애플리케이션에 적합한 기술이므로 트래픽 감소, 속도, 사용성, 반응성의 향상 등의 장점을 생각한다면 결정적인 단점이라고 할 수는 없다.

SEO(검색엔진 최적화) 문제
SPA는 서버 사이드 렌더링 방식이 아닌 자바스크립트 기반 비동기 모델(클라이언트 사이드 렌더링 방식)이다. 따라서 SEO는 언제나 단점으로 부각되어 왔던 이슈이다. 하지만 SPA는 정보의 제공을 위한 웹페이지보다는 애플리케이션에 적합한 기술이므로 SEO 이슈는 심각한 문제로 볼 수 없다. Angular 또는 React 등의 SPA 프레임워크(React는 라이브러리로 구분된다)는 서버 사이드 렌더링을 지원하는 SEO 대응 기술이 이미 존재하고 있어 SEO 대응이 필요한 페이지에 대해서는 선별적 SEO 대응이 가능하다.

2. Routing

라우팅이란 출발지에서 목적지까지의 경로를 결정하는 기능이다. 애플리케이션의 라우팅은 사용자가 태스크를 수행하기 위해 어떤 화면(view)에서 다른 화면으로 화면을 전환하는 내비게이션을 관리하기 위한 기능을 의미한다. 일반적으로 사용자가 요청한 URL 또는 이벤트를 해석하고 새로운 페이지로 전환하기 위한 데이터를 취득하기 위해 서버에 필요 데이터를 요청하고 화면을 전환하는 일련의 행위를 말한다.

브라우저가 화면을 전환하는 경우는 아래와 같다.

1. 브라우저의 주소창에 URL을 입력하면 해당 페이지로 이동한다.
2. 웹페이지의 링크(a 요소)를 클릭하면 a 요소의 href 어트리뷰트를 기반으로 브라우저의 주소창의 URL을 변경하고 해당 페이지로 이동한다.
3. form 요소의 submit 버튼을 클릭하면 form 요소의 action 어트리뷰트에 지정한 URL로 입력 데이터(form data)가 전송되며 해당 페이지로 이동한다.
4. 브라우저의 '뒤로가기' 또는 '앞으로가기' 버튼을 클릭하면 사용자가 방문한 웹페이지 기록(history)의 뒤(history.back()) 또는 앞(history.forward())으로 이동한다.

이처럼 브라우저가 화면을 전환하는 경우, 모두 브라우저의 주소창의 URL을 변경한다. 하지만 AJAX 요청에 의해 서버로부터 데이터를 응답받아 화면을 생성하는 경우, 브라우저 주소창의 URL은 변경되지 않는다. 이는 사용자의 방문 기록(history)을 관리할 수 없음을 의미하며, SEO(검색엔진 최적화) 이슈의 발생 원인이기도 하다. history 관리를 위해서는 각 페이지는 브라우저의 주소창에서 구별할 수 있는 유일한 URL을 소유하여야 한다.

3. Angular Router 개요와 위치 정책(Location strategy)

3.1 개요

Angular는 단일 페이지 애플리케이션(SPA)을 위한 클라이언트 사이드 내비게이션 구현 방식으로 Angular 라우터를 제공한다. Angular 라우터는 선언적 방식으로 라우트를 구성하고 라우트에 해당하는 컴포넌트를 매핑한다. 즉, 사용자의 요청 URL을 해석하고 애플리케이션의 뷰를 담당하는 컴포넌트와 연결하는 역할을 한다.

Angular는 사용자의 요청 URL의 패스(path, 경로)와 컴포넌트의 쌍으로 구성된 라우트 설정을 참조하여 뷰를 출력한다. 라우트 설정의 예는 아래와 같다.

TYPESCRIPT

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
```

```
  { path: '**', component: NotFoundComponent }
];
```

3.2 Location strategy

a 요소의 href 어트리뷰트를 사용하지 않는 AJAX는 브라우저 주소창의 URL을 변경시키지 않는다. 이는 브라우저의 뒤로가기, 앞으로가기 등의 history 관리가 동작하지 않음을 의미한다. 물론 코드 상의 history.back(), history.go(n) 등도 동작하지 않는다. 새로고침을 클릭하면 주소창의 주소가 변경되지 않기 때문에 언제나 첫 페이지가 다시 로딩된다. 하나의 주소로 동작하는 AJAX 방식은 SEO 이슈에서도 자유로울 수 없다.

Angular는 이와 같은 문제점을 해결할 수 있는 2가지의 위치 정책(Location strategy)을 제공한다. 이로써 각 페이지는 브라우저의 주소창에서 구별할 수 있는 애플리케이션 전역에서 유일한 URL을 소유하게 된다.

- PathLocationStrategy (HTML5 Histroy API pushState 기반 내비게이션 정책)
- HashLocationStrategy (Hash 기반 내비게이션 정책)

3.2.1 PathLocationStrategy (HTML5 Histroy API pushState 기반 내비게이션 정책)

HTML5의 Histroy API pushState 메소드를 사용하는 정책으로 '/home', '/service', '/about'와 같이 URL 패스(path)를 기반으로 한다.

CODE

```
localhost:4200/service
```

PathLocationStrategy는 **Angular 라우터의 기본 정책**으로 pushState 메소드를 별도로 호출할 필요가 없다. 특별한 이유가 없는 한 사용자가 보다 쉽게 이해할 수 있는 이 정책의 사용을 권장한다. 또한, Angular Universal을 사용하여 서버 사이드 렌더링을 도입하려면 이 정책을 사용하여야 한다.

3.2.2 HashLocationStrategy (Hash 기반 내비게이션 정책)

URL 패스에 fragment identifier의 고유 기능인 앵커(anchor)를 사용하는 정책으로 '/#/service', '/#/about'과 같이 해시뱅을 기반으로 한다. fragment identifier는 hash mark 또는 hash라고 부르기도 한다.

URL이 동일한 상태에서 hash가 변경되면 브라우저는 서버에 어떠한 요청도 하지 않는다. 즉, hash는 변경되어도 서버에 새로운 요청을 보내지 않으며 따라서 페이지가 갱신되지 않는다.
HashLocationStrategy는 대부분의 브라우저에서 동작한다.

CODE

```
localhost:4200/#/service
```

Hash 기반 내비게이션 정책을 기본으로 사용하려면 루트 모듈의 imports 프로퍼티를 아래와 같이 수정한다.

TYPESCRIPT

```
// app.module.ts
...
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];

@NgModule({
  imports: [
    BrowserModule,
    // PathLocationStrategy (기본 정책)
    // RouterModule.forRoot(routes)

    // HashLocationStrategy
    RouterModule.forRoot(routes, { useHash: true })
...
```

라우팅 모듈을 사용하는 경우, imports 프로퍼티를 아래와 같이 수정한다.

TYPESCRIPT

```
// app-routing.module.ts
...
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];

@NgModule({
  // PathLocationStrategy (기본 정책)
  // imports: [RouterModule.forRoot(routes)],

  // HashLocationStrategy
  imports: [RouterModule.forRoot(routes, { useHash: true })],
  ...
})
...
```

4. 라우터 구성 요소

이제 라우터를 구성하는 요소에 대해 살펴보도록 하자. 일반적으로 라우터는 아래의 수순으로 작성한다.

1. 라우트 구성
   Route 인터페이스의 배열(Routes 타입)을 사용하여 요청 URL의 패스와 컴포넌트의 쌍으로 만들어진 라우트를 구성한다.
2. 라우트 등록
   RouterModule.forRoot 또는 RouterModule.forChild를 호출하여 라우트 구성이 포함된 모듈을 생성하고 루트 모듈 또는 기능 모듈에 등록한다.
3. 뷰의 렌더링 위치 지정

라우터가 컴포넌트를 렌더링하여 뷰를 표시할 영역인 `<router-oulet>` 을 구현하는 RouterOutlet 디렉티브를 선언하여 컴포넌트 뷰가 렌더링될 위치를 지정한다. RouterOutlet 디렉티브는 루트 컴포넌트 또는 기능 모듈의 컴포넌트에 선언한다.

4. 내비게이션 작성

   RouterLink 디렉티브를 사용한 HTML a 요소를 사용하여 내비게이션을 작성한다.

우선 프로젝트를 생성하고 라우팅 대상인 HomeComponent, ServiceComponent, AboutComponent, NotFoundComponent 컴포넌트를 작성하여 보자.

BASH

```
$ ng new routing-exam -t -s -S
$ cd routing-exam
$ ng generate component pages/home --flat
$ ng generate component pages/service --flat
$ ng generate component pages/about --flat
$ ng generate component pages/not-found --flat
```

각 컴포넌트를 아래와 같이 수정한다.

TYPESCRIPT

```
// pages/home.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-home',
  template: `<div class="home">Home</div>`,
  styles: [`.home { font-size: 2em; padding: 60px; }`]
})
export class HomeComponent {}
```

TYPESCRIPT

```
// pages/service.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-service',
  template: `<div class="service">Service</div>`,
```

```
  styles: [`.service { font-size: 2em; padding: 60px; }`]
})
export class ServiceComponent {}
```

TYPESCRIPT

```
// pages/about.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-about',
  template: `<div class="about">About</div>`,
  styles: [`.about { font-size: 2em; padding: 60px; }`]
})
export class AboutComponent {}
```

TYPESCRIPT

```
// pages/not-found.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-not-found',
  template: `<div class="not-found">Not Found</div>`,
  styles: [`.not-found { font-size: 2em; padding: 60px; }`]
})
export class NotFoundComponent {}
```

그리고 pages 폴더 내의 컴포넌트들을 일괄 공개하는 index.ts를 생성한다.

TYPESCRIPT

```
// pages/index.ts
export * from './home.component';
export * from './about.component';
export * from './service.component';
export * from './not-found.component';
```

이제 위에서 설명한 4개의 수순(라우트 구성 → 라우트 등록 → 뷰의 렌더링 위치 지정 → 내비게이션 작성)을 거치면서 라우터를 구성해 보도록 하자.

4.1 라우트 구성

Angular는 사용자의 요청 URL의 패스(path, 경로)와 컴포넌트의 쌍으로 만들어진 라우트 구성을 참조하여 뷰를 출력한다. 뷰를 출력한다는 것은 요청 URL의 패스에 해당하는 라우트 구성의 컴포넌트를 활성화하는 것을 의미한다.

앞서 생성한 프로젝트 컴포넌트들의 라우트 구성의 예는 아래와 같다.

TYPESCRIPT

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];
```

위 라우트 구성의 의미는 아래와 같다.

요청 URL 패스	URL 범례	활성화될 컴포넌트
home	localhost:4200/home	HomeComponent
service	localhost:4200/service	ServiceComponent
about	localhost:4200/about	AboutComponent
상기 패스 이외	localhost:4200/some	NotFoundComponent

라우트 구성은 Route 인터페이스를 사용하여 배열로 구성한다. Routes 타입은 Route 인터페이스 배열의 Type Aliase이다.

TYPESCRIPT

```
// @angular/router/src/config.d.ts
export declare type Routes = Route[];
```

Route 인터페이스는 @angular/router에 포함되어 있으며 코드는 아래와 같다.

TYPESCRIPT

```typescript
// @angular/router/src/config.d.ts
export interface Route {
  path?: string;
  pathMatch?: string;
  matcher?: UrlMatcher;
  component?: Type<any>;
  redirectTo?: string;
  outlet?: string;
  canActivate?: any[];
  canActivateChild?: any[];
  canDeactivate?: any[];
  canLoad?: any[];
  data?: Data;
  resolve?: ResolveData;
  children?: Routes;
  loadChildren?: LoadChildren;
  runGuardsAndResolvers?: RunGuardsAndResolvers;
}
```

Route 인터페이스의 path 프로퍼티는 URL 패스(경로)를 나타내는 문자열이고 component 프로퍼티는 컴포넌트의 타입을 나타낸다.

예를 들어 '/home'라는 URL 패스가 요청되면 HomeComponent를 활성화하고, '/service'라는 URL 패스가 요청되면 ServiceComponent를 활성화하고, '/about'라는 URL 패스가 요청되면 AboutComponent를 활성화하여 뷰를 출력하는 경우, 라우트 구성은 아래와 같다. 이때 **패스의 '/'는 기술하지 않는다.**

TYPESCRIPT

```typescript
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent }
];
```

URL 패스에 매칭하는 라우트 정보가 없다면 path 프로퍼티에 '**'를 사용하여 아래와 같이 라우트를 구성한다.

TYPESCRIPT

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];
```

`path: '**'` 는 반드시 라우트 구성의 가장 마지막에 위치하여야 한다. 만약 가장 앞에 위치하면 '**'는 모든 URL 패스에 매칭되어 이후에 선언된 라우트가 적용되지 않는다.

redirectTo 프로퍼티는 요청을 리다이렉트할 때 사용한다. redirectTo 프로퍼티는 일반적으로 pathMatch 프로퍼티와 함께 사용한다.

pathMatch 프로퍼티에 문자열 'full'을 설정하면 path 프로퍼티의 패스와 요청 URL 패스 전체가 정확하게 매칭할 때 리다이렉트한다. pathMatch 프로퍼티에 문자열 'prefix'를 설정하면 path 프로퍼티의 패스와 요청 URL 패스가 앞부분만 매칭하여도 리다이렉트한다.

예를 들어 루트 패스 '/'가 요청되면 '/home'으로 리다이렉트하여 HomeComponent를 활성화하여 뷰를 표시하는 경우, 라우트 구성은 아래와 같다. **path의 값이 ''인 경우, 리다이렉트 라우트는 반드시 pathMatch: 'full'을 설정하여야 한다.**

TYPESCRIPT

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];
```

4.2 라우트 등록

라우트는 모듈 단위로 구성한다. 그리고 구성된 라우트는 모듈에 등록한다.

루트 모듈에 라우트 구성을 추가하여 보자. 먼저 Routes와 RouterModule을 import하고 라우트 구성을 추가한다.

TYPESCRIPT

```
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
// 라우터 모듈
import { Routes, RouterModule } from '@angular/router';

import { AppComponent } from './app.component';

// 컴포넌트 임포트
import {
  HomeComponent,
  ServiceComponent,
  AboutComponent,
  NotFoundComponent
} from './pages';

// 라우트 구성
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];
...
```

@NgModule 데코레이터의 imports 프로퍼티에 RouterModule.forRoot 메소드를 호출하여 모든 라우트 구성을 포함한 라우터 모듈을 생성하고 루트 모듈에 등록한다. forRoot 메소드는 루트 모듈에 라우트 구성을 등록할 때 사용한다.

TYPESCRIPT

```
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
```

```
import { NgModule } from '@angular/core';
// 라우터 모듈
import { Routes, RouterModule } from '@angular/router';

import { AppComponent } from './app.component';

// 컴포넌트 임포트
import {
  HomeComponent,
  ServiceComponent,
  AboutComponent,
  NotFoundComponent
} from './pages';

// 라우트 구성
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ServiceComponent,
    AboutComponent,
    NotFoundComponent
  ],
  imports: [
    BrowserModule,
    /* 모든 라우트 구성을 포함한 라우터 모듈을 생성하고 루트 모듈에 등록 */
    RouterModule.forRoot(routes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

위와 같이 라우트 구성을 루트 모듈에 직접 등록할 수도 있으나 라우트 구성이 커지면 라우팅 모듈을 별도 작성하고 그것을 루트 모듈에 등록하는 방식이 유리할 수 있다. 라우팅 모듈을 사용하여 라우트 구성을 루트 모듈로부터 분리하도록 하자.

참고로 Angular CLI의 ng new 명령어로 프로젝트를 생성할 때, routing 옵션(`--routing`)을 추가하면 아래의 내용이 자동 처리된다.

- 라우팅 모듈 app-routing.module.ts를 자동 생성한다.
- app.component.ts에 `<router-outlet></router-outlet>` 을 추가한다.
- 루트 모듈 app.module.ts에 라우팅 모듈을 import한다.

라우트 구성을 위한 AppRoutingModule을 아래와 같이 작성한다.

TYPESCRIPT

```
// app-routing.module.ts
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

// 컴포넌트 임포트
import {
  HomeComponent,
  ServiceComponent,
  AboutComponent,
  NotFoundComponent
} from './pages';

// 라우트 구성
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'service', component: ServiceComponent },
  { path: 'about', component: AboutComponent },
  { path: '**', component: NotFoundComponent }
];

@NgModule({
  /* 모든 라우트 구성을 포함한 모듈을 생성하고 라우팅 모듈에 추가 */
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

루트 모듈에 라우팅 모듈을 등록한다.

TYPESCRIPT

```typescript
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
// 라우팅 모듈 임포트
import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

// 컴포넌트 임포트
import {
  HomeComponent,
  ServiceComponent,
  AboutComponent,
  NotFoundComponent
} from './pages';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ServiceComponent,
    AboutComponent,
    NotFoundComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule /* 라우팅 모듈 등록 */
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

4.3 뷰의 렌더링 위치 지정과 내비게이션 작성

이로서 라우트 구성과 등록이 완성되었다. 하지만 아직 뷰를 어디에 표시할 것인지 위치를 지정하지 않았다. 그리고 뷰의 전환을 위한 네비게이션 또한 작성되지 않았다.

RouterOutlet 디렉티브를 사용하여 뷰의 렌더링 위치를 지정하고 RouterLink, RouterLinkActive 디렉티브를 사용하여 내비게이션을 작성하는 방법에 대해 살펴보자.

4.3.1 RouterOutlet

RouterOutlet는 라우터가 컴포넌트를 렌더링하여 뷰를 표시할 영역인 `<router-oulet>` 을 구현한 디렉티브로 컴포넌트의 뷰를 렌더링할 위치를 설정한다.

HTML

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

위와 같이 RouterOutlet 디렉티브는 루트 컴포넌트 또는 기능 컴포넌트의 템플릿 내에 선언하며 선언된 위치에 해당 뷰를 표시한다.

4.3.2 RouterLink

뷰의 전환을 위한 내비게이션을 작성할 때 많이 사용되는 것이 a 요소이다. Angular 라우터를 사용하려면 컴포넌트의 템플릿에는 뷰를 전환하기 위한 a 요소의 href 어트리뷰트 대신 RouterLink 디렉티브를 사용하여 URL 경로를 지정한다. a 요소의 href 어트리뷰트를 사용한다면 서버로 페이지 요청이 발생하기 때문이다.

HTML

```
<!-- app.component.ts -->
<nav>
  <a routerLink="/home">Home</a>
  <a routerLink="/service">Service</a>
  <a routerLink="/about">About</a>
```

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

RouterLink 디렉티브는 자신의 값을 라우터에 전달한다. 라우터는 이를 전달받아 해당 컴포넌트를 활성화하여 뷰를 출력한다.

위 예제에서 nav 요소의 자식 요소인 `<a routerLink="/home">Home</a>` 을 클릭하면 RouterLink 디렉티브는 자신의 값 "/home"을 라우터에 전달한다. 이때 라우터는 전달된 값 "/home"을 요청 URL의 패스로 인식하고 이에 해당하는 컴포넌트를 라우트 구성에서 검색하여 활성화한다. 그리고 이 컴포넌트의 뷰는 `<router-outlet></router-outlet>` 에 표시된다.

앞서 생성한 예제의 루트 컴포넌트에 RouterLink 디렉티브를 선언하여 보자.

TYPESCRIPT

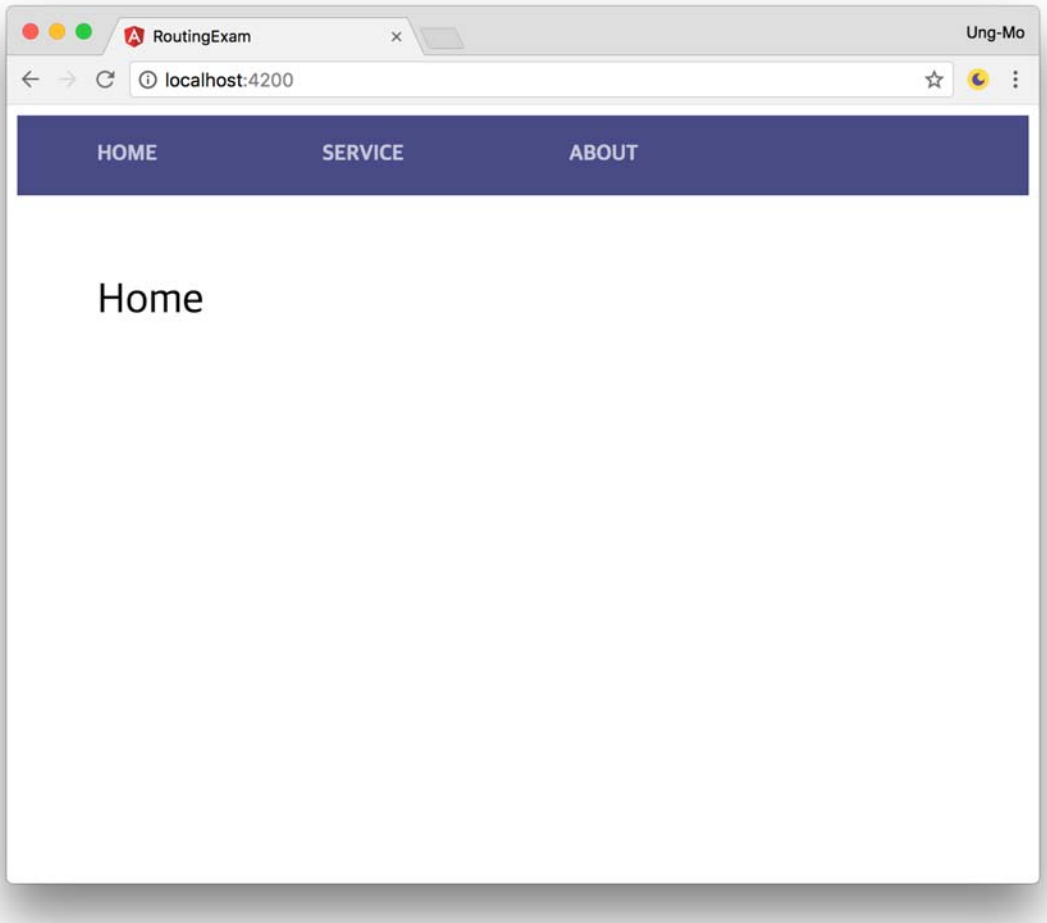
```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <nav>
      <a routerLink="/home">Home</a>
      <a routerLink="/service">Service</a>
      <a routerLink="/about">About</a>
    </nav>
    <router-outlet></router-outlet>
  `,
  styles: [`
    nav {
      height: 60px;
      background-color: #4a4c88;
    }
    nav > a {
      line-height: 60px;
      margin: 0 60px;
      color: #fff;
      text-decoration: none;
      font-weight: bold;
      text-transform: uppercase;
```

```
      opacity: 0.7;
    }
    nav > a:hover {
      opacity: 1.0;
    }
  `]
})
export class AppComponent {}
```

RouterLink 디렉티브가 선언된 링크를 클릭하면 routerLink의 값이 라우터로 전달된다.

예를 들어 애플리케이션이 초기 기동하면 브라우저 주소창에는 localhost:4200이 지정되고 이 URL이 요청된다. 이때 요청 URL 패스는 빈 문자열 “이고 라우터는 이 요청 URL 패스를 전달받아 활성화해야할 컴포넌트 HomeComponent를 라우터 구성에서 검색하고 활성화하여 RouterOutlet 영역에 뷰를 렌더링할 것이다. 두번째 링크를 클릭하면 ‘/service’가 라우터로 전달되고 이에 해당하는 SeviceComponent를 활성화하여 RouterOutlet 영역에 뷰를 렌더링할 것이다.

위 예제를 실행 결과는 아래와 같다.



4.3.3 RouterLinkActive

RouterLinkActive 디렉티브는 현재 브라우저의 URL 패스가 RouterLink 디렉티브에서 **지정한 URL 패스의 트리에 포함**되는 경우, RouterLinkActive에 지정된 클래스명을 DOM에 자동으로 추가한다.

HTML

```
<a routerLink="/service" routerLinkActive="active">Service</a>
```

예를 들어 위의 경우, 현재 브라우저의 URL 패스가 '/' 또는 '/service'인 경우, routerLinkActive 디렉티브가 지정한 active 클래스가 DOM에 자동 추가된다.

브라우저의 URL 패스가 RouterLink 디렉티브에서 **지정한 URL 패스와 정확히 일치**하는 경우, RouterLinkActive에 지정된 클래스명을 DOM에 자동으로 추가하려면 아래와 같이 routerLinkActiveOptions 디렉티브를 사용한다.

HTML

```
<a routerLink="/service"
  [routerLinkActiveOptions]="{ exact: true }"
  routerLinkActive="active">Service</a>
```

한 개 이상의 클래스를 지정할 경우, 아래와 같이 지정한다.

HTML

```
<a routerLink="/service" routerLinkActive="class1 class2">Service</a>
<a routerLink="/service" [routerLinkActive]="['class1', 'class2']">Service</
a>
```

RouterLinkActive 디렉티브를 적용한 루트 컴포넌트는 아래와 같다.

TYPESCRIPT

```
// app.component.ts
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  template: `
    <nav>
      <a routerLink="/home"
        routerLinkActive="active"
        [routerLinkActiveOptions]="{ exact: true }">
        Home</a>
      <a routerLink="/service"
        routerLinkActive="active"
        [routerLinkActiveOptions]="{ exact: true }">
        Service</a>
      <a routerLink="/about"
        routerLinkActive="active"
        [routerLinkActiveOptions]="{ exact: true }">
        About</a>
    </nav>
    <router-outlet></router-outlet>
  `,
  styles: [`
    nav {
      height: 60px;
      background-color: #4a4c88;
    }
    nav > a {
      line-height: 60px;
      margin: 0 60px;
      color: #fff;
      text-decoration: none;
      font-weight: bold;
      text-transform: uppercase;
      opacity: 0.7;
    }
    nav > a:hover, nav > a.active  {
      opacity: 1.0;
    }
  `]
})
export class AppComponent {}
```

5. navigate 메소드

템플릿의 a 요소를 사용하지 않고 컴포넌트 클래스의 코드만으로 화면을 전환하기 위해서는 navigate 메소드를 사용한다. 간단한 예제를 통해 navigate 메소드를 살펴보도록 하자.

루트 모듈은 아래와 같다.

TYPESCRIPT

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { Routes, RouterModule } from '@angular/router';
```

```
import { AppComponent } from './app.component';
import { TodosComponent } from './todos.component';

// 라우트 구성
const routes: Routes = [
  { path: 'todos', component: TodosComponent }
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes)
  ],
  declarations: [
    AppComponent,
    TodosComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

라우트 패스 /todos의 뷰를 담당하는 TodosComponent는 아래와 같다.

TYPESCRIPT

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

@Component({
  selector: 'app-todos',
  template: `<p>Todos</p>`
})
export class TodosComponent {}
```

루트 컴포넌트를 아래와 같이 작성한다.

TYPESCRIPT

```
// app.component.ts
import { Component } from '@angular/core';
import { Router } from '@angular/router';
```

```
@Component({
  selector: 'app-root',
  template: `
    <button (click)="gotoTodos()">goto todos</button>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {
  constructor(private router: Router) {}

  gotoTodos() {
    // /todos로 이동
    this.router.navigate(['todos']);
  }
}
```

'goto todos' 버튼을 클릭하면 '/todos'로 이동한다. 이때 a 요소를 사용하지 않았지만 Router 클래스의 멤버인 navigate 메소드를 통해 화면 전환이 가능하다. Router 클래스의 인스턴스는 의존성 주입을 통해 컴포넌트로 주입받아 사용한다.

navigate-method-exam
By ungmo2

Run Project

Reference

- Javascript SPA & Routing
- Angular Routing & Navigation
- Route
- RouterOutlet
- RouterLink
- RouterLinkActive
- Router
- ActivatedRoute