PoiemaWeb

13.25 Angular Router state & Guard

라우터 상태, 자식 라우트, 가드

#1. 라우터 상태(Router state)

1.1 라우트 파라미터(Route Parameter) 전달

화면 전환시에 라우트 파라미터(Route Parameter)를 사용하여 활성화될 컴포넌트에 데이터를 전달하는 방법에 대해 살펴보도록 하자.

RouterLink 디렉티브는 자신의 값 즉 URL 패스를 라우터에 전달하고 라우터는 이를 전달받아 라우트 구성에서 전달받은 값(URL 패스)에 해당하는 컴포넌트를 검색하고 활성화하여 `<router-outlet>`

</router-outlet> 영역에 뷰를 출력한다.

HTML

```
<a routerLink="/todo">...</a>
```

이때 라우트 파라미터를 컴포넌트에 전달할 수 있다. 예를 들어 URL 패스가 '/todo/:id'인 경우, URL 패스의 두번째 세그먼트 :id는 라우터 파라미터이며 컴포넌트에게 전달하고자 하는 값을 할당한다. 만일 '/todos/10'과 같이 :id에 값 10을 할당하면 활성화될 컴포넌트에게 id 값으로 10이 전달된다.

HTML

```
<a routerLink="/todo/:id">...</a>
```

이때 라우트 구성이 아래와 같다면 TodoDetailComponent가 활성화될 것이다.

TYPESCRIPT

```
const routes: Routes = [
  { path: '', component: TodosComponent },
  { path: 'todo/:id', component: TodoDetailComponent }
];
```

라우터 파라미터로 전달할 값은 대부분 변수 또는 객체의 프로퍼티에 담겨있는 동적인 값일 것이다. 이러한 경우, RouterLink 디렉티브에 URL 패스의 세그먼트로 구성된 배열을 할당한다. 예를 들어 위 라우트 구성의 라우터 파라미터 :id의 값이 컴포넌트 클래스에서 동적으로 생성되는 경우, RouterLink 디렉티브는 아래와 같이 구성한다.

HTML

```
<a [routerLink]="['/todo', todoId]">...</a>
```

위 예제의 경우, RouterLink 디렉티브에 할당된 배열의 첫번째 요소는 URL 패스의 첫번째 세그먼트이며 두번째 요소가 URL 패스의 두번째 세그먼트인 라우터 파라미터의 값이다. 예를 들어 컴포넌트 클래스의 프로퍼티 todoId의 값이 10이라면 위 코드는 아래와 동일한 위미를 갖는다.

HTML

```
<a routerLink="/todo/10">...</a>
```

navigate 메소드를 사용할 경우, 아래와 같이 URL 패스의 세그먼트로 구성된 배열을 인수로 전달한다.

TYPESCRIPT

```
this.router.navigate(['/todo', todoId]);
```

라우트 파라미터를 컴포넌트에 전달하는 예제를 작성해 보자.

작성할 예제는 2개의 컴포넌트 TodosComponent, TodoDetailComponent로 구성되어 있다. TodosComponent는 모든 할일의 리스트의 링크를 표시한다. 할일을 클릭하면 해당 할일의 상세 페이지인 TodoDetailComponent로 이동하는 간단한 예제이다.

프로젝트를 생성하고 2개의 컴포넌트를 생성한다.

BASH

```
$ ng new route-parameter-exam -t -s -S --routing
$ cd route-parameter-exam
$ ng generate component todos
$ ng generate component todos/todo-detail --flat
```

우선 루트 컴포넌트를 아래와 같이 수정한다.

TYPESCRIPT

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

@Component({
  selector: 'app-root',
  template: '<router-outlet></router-outlet>'
})
export class AppComponent {}
```

`--routing` 옵션을 통해 생성된 AppRoutingModule에 라우트 구성을 추가하도록 하자.

TYPESCRIPT

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

// 컴포넌트 임포트
import { TodosComponent } from './todos/todos.component';
import { TodoDetailComponent } from './todos/todo-detail.component';

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

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

이제 프로젝트를 실행하고 루트 URL(localhost:4200)로 접근하면 루트 컴포넌트의 `<router-outlet></router-outlet>` 영역에 TodosComponent가 표시되고 localhost:4200/todo/1과 같이 접근하면 `<router-outlet></router-outlet>` 영역에 TodoDetailComponent이 표시될 것이다.

TodosComponent를 수정하여 모든 할일 리스트의 링크를 표시하도록 하자.

TYPESCRIPT

```
// todos/todos.component.ts
import { Component, OnInit } from '@angular/core';

interface Todo {
  id: number;
  content: string;
  completed: boolean;
}
```

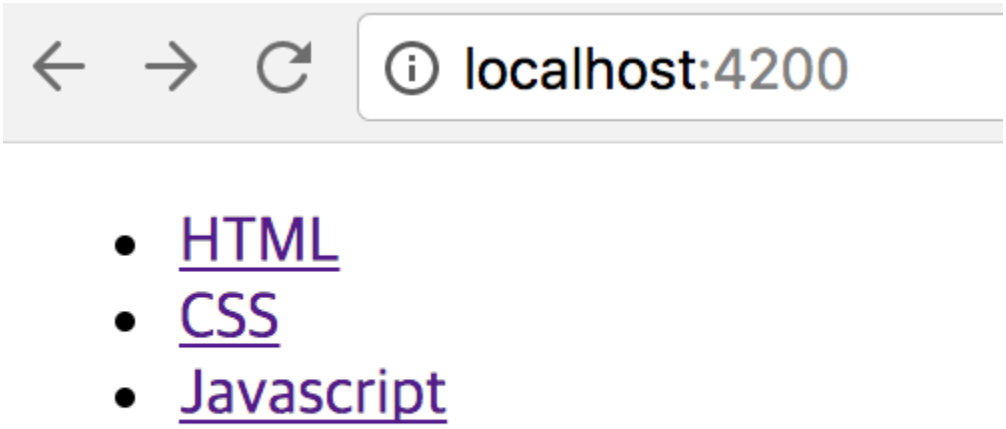
```
@Component({
  selector: 'app-todos',
  template: `
    <ul>
      <li *ngFor="let todo of todos">
        <a [routerLink]="['/todo', todo.id]">{{ todo.content }}</a>
      </li>
    </ul>
  `
})
export class TodosComponent implements OnInit {
  todos: Todo[];

  ngOnInit() {
    // 잠정 처리. 실제 환경에서는 서비스를 통해 서버로 부터 데이터를 취득할 것이다.
    this.todos = [
      { id: 3, content: 'HTML', completed: false },
      { id: 2, content: 'CSS', completed: true },
      { id: 1, content: 'Javascript', completed: false }
    ];
  }
}
```

이 예제에서는 컴포넌트 클래스의 todos 배열에 고정된 값을 할당하였다. 실제 환경에서는 서비스를 통해 서버로 부터 데이터를 동적으로 취득할 것이므로 todos 배열이 동적으로 생성된다고 가정하자. 이처럼 동적인 값을 사용하여 URL 패스를 생성하는 경우, RouterLink 디렉티브에 URL 패스의 세그먼트로 구성된 배열을 할당한다.

HTML

```
<li *ngFor="let todo of todos">
  <a [routerLink]="['/todo', todo.id]">{{ todo.content }}</a>
</li>
```

프로젝트를 실행하면 루트 URL에 해당하는 할일 리스트의 링크를 갖는 TodosComponent가 화면에 렌더링될 것이다.



1.2 라우트 파라미터(Route Parameter) 취득

라우트 파라미터를 전달하는 방법에 대해 살펴보았다. 앞서 작성한 예제는 라우터 파라미터를 활성화될 컴포넌트 TodoDetailComponent에 전달한다. 전달한 라우트 파라미터를 취득하는 방법에 대해 알아보도록 하자.

`<router-oulet>` 영역에 렌더링된 컴포넌트 다시 말해 활성화된 컴포넌트는 ActivatedRoute 객체를 통해 라우터 상태(Router state)에 접근할 수 있다. 즉, ActivatedRoute 객체는 다양한 라우터 상태를 가지며 이 중에서 라우트 파라미터를 추출할 수 있다. ActivatedRoute는 아래와 같은 프로퍼티를 제공한다.

TYPESCRIPT

```
interface ActivatedRoute {
  snapshot: ActivatedRouteSnapshot
  url: Observable<UrlSegment[]>
  params: Observable<Params>
  queryParams: Observable<Params>
  fragment: Observable<string>
  data: Observable<Data>
  outlet: string
  component: Type<any>|string|null
  ...
}
```

ActivatedRoute 객체의 인스턴스는 의존성 주입을 통해 컴포넌트로 주입받는다. 앞에서 생성한 프로젝트의 TodoDetailComponent에 ActivatedRoute 객체의 인스턴스를 주입받도록 하자.

TYPESCRIPT

```
// todo-detail.component.ts
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-todo-detail',
  template: `
    <h3>todo detail</h3>
    <p>todo id : {{ todoId }}</p>
    <a routerLink="/">Back to Todos</a>
  `
})
export class TodoDetailComponent implements OnInit {
  // ActivatedRoute 객체의 인스턴스를 의존성 주입을 통해 주입받는다.
  constructor(private route: ActivatedRoute) { }
}
```

라우트 파라미터의 값을 취득할 때는 ActivatedRoute의 paramMap 프로퍼티를 사용한다. paramMap 프로퍼티는 라우터에 전달된 라우트 파라미터의 맵을 포함하는 옵저버블이다.

Angular는 URL 패스가 변경되었지만 활성화 대상 컴포넌트가 변경되지 않는 경우, 만약 활성화 대상 컴포넌트가 존재하면 다시 생성하지 않고 재사용한다. 따라서 컴포넌트가 소멸되지 않은 상태에서 라우터 파라미터만 변경된 라우터 상태를 연속으로 수신할 수 있기 때문에 paramMap을 옵저버블로 제공한다. paramMap의 get 메소드에 라우트 파라미터의 키값을 인자로 전달하여 라우트 파라미터의 값을 취득한다.

TYPESCRIPT

```
// todo-detail.component.ts
...
export class TodoDetailComponent implements OnInit {

  todoId: number;

  // ActivatedRoute 객체의 인스턴스를 의존성 주입을 통해 주입받는다.
```

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

  ngOnInit() {
    // 라우트 파라미터 값의 취득
    this.route.paramMap
      .subscribe(params => this.todoId = +params.get('id'));
  }
}
```

옵저버블 스트림이 아닌 특정 시점의 상태만을 조회하는 경우, ActivatedRoute의 snapshot 프로퍼티를 사용할 수 있다. snapshot 프로퍼티는 옵저버블로 래핑되지 않은 paramMap 객체를 반환한다. 완성된 TodoDetailComponent는 아래와 같다.

TYPESCRIPT

```
// todo-detail.component.ts
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-todo-detail',
  template: `
    <h3>todo detail</h3>
    <p>todo id : {{ todoId }}</p>
    <a routerLink="/">Back to Todos</a>
  `
})
export class TodoDetailComponent implements OnInit {

  todoId: number;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    // this.route.paramMap
      // .subscribe(params => this.todoId = +params.get('id'));

    this.todoId = +this.route.snapshot.paramMap.get('id');
  }
}
```

1.3 라우트 정적 데이터(Route static data)

Route 인터페이스의 data 프로퍼티는 컴포넌트로 전송할 라우트 정적 데이터로서 일반적으로 애플리케이션 운영에 필요한 데이터를 전달할 때 사용한다. 예를 들어 해당 화면의 타이틀과 사이드바의 표시 여부를 전달하고자 하는 경우, 아래와 같이 data 프로퍼티에 라우트 정적 데이터를 설정한다.

TYPESCRIPT

```
const routes: Routes = [
  {
    path: 'todos',
    component: TodosComponent,
```

```
    data: { title: 'Todos', sidebar: true } /* 라우트 정적 데이터 */
  }
];
```

앞서 작성한 AppRoutingModule의 라우트 구성에 라우트 정적 데이터를 추가해보자.

TYPESCRIPT

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

// 컴포넌트 임포트
import { TodosComponent } from './todos/todos.component';
import { TodoDetailComponent } from './todos/todo-detail.component';

// 라우트 구성
const routes: Routes = [
  { path: '', component: TodosComponent },
  {
    path: 'todo/:id',
    component: TodoDetailComponent,
    // 라우트 정적 데이터
    data: { title: 'Todos', sidebar: true }
  }
];

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

활성화된 컴포넌트는 ActivatedRoute 객체를 통해 라우터 상태(Router state)에 접근할 수 있다. 라우트 정적 데이터 또한 라우트 파라미터와 마찬가지로 ActivatedRoute 객체를 통해 접근할 수 있다.

앞서 작성한 TodoDetailComponent를 수정하여 라우트 정적 데이터를 취득하여 보자.

TYPESCRIPT

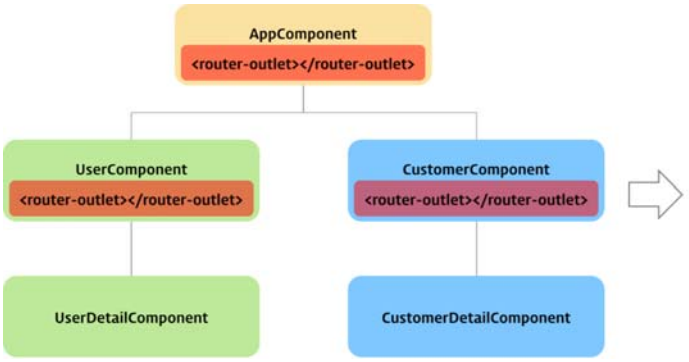
```typescript
// todo-detail.component.ts
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

interface config {
  title: string;
  sidebar: boolean
}

@Component({
  selector: 'app-todo-detail',
  template: `
    <p>todo detail</p>
    <p>todo id : {{ todoId }}</p>
    <p>data : {{ data | json }}</p>
    <p>title : {{ data.title }}</p>
    <p>sidebar : {{ data.sidebar ? 'show' : 'hidden' }}</p>
    <a routerLink="/">Back to Todos</a>
  `
})
export class TodoDetailComponent implements OnInit {

  todoId: number;
  data: config;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    // 라우트 파라미터 취득
    this.todoId = +this.route.snapshot.paramMap.get('id');

    // 라우트 정적 데이터 취득
    this.data = this.route.snapshot.data as config;
    this.data.title = this.route.snapshot.data.title;
    this.data.sidebar = this.route.snapshot.data.sidebar;
  }
}
```

route-static-data-exam
By ungmo2

Run Project

2. 자식 라우트(Child Route)

지금까지는 루트 컴포넌트에 하나의 `<router-outlet>` 을 가진 예제만을 살펴보았다. 자식 컴포넌트도 루트 컴포넌트의 `<router-outlet>` 와는 별도로 자신의 자식 컴포넌트를 위한 `<router-outlet>` 을 가질 수 있다. 예를 들어 아래의 그림을 살펴보자.




```
const routes: Routes = [
  { path: '', redirectTo: '/user',  pathMatch: 'full' },
  {
    path: 'user',
    component: UserComponent,
    children: [
      { path: ':id', component: UserDetailComponent },
    ]
  },
  {
    path: 'customer',
    component: CustomerComponent,
    children: [
      { path: ':id', component: CustomerDetailComponent },
    ]
  }
];
```

자식 라우트

위 그림의 경우, 루트 컴포넌트의 `<router-outlet>` 에는 UserComponent 또는 CustomerComponent가 표시된다. 이때 UserComponent와 CustomerComponent는 자신의 `<router-outlet>` 을 가지고 있으며 이 영역에는 자식 라우트 구성을 위한 children 프로퍼티에 선언한 컴포넌트가 표시된다. 이와 같은 관계를 구성한 라우트는 아래와 같다.

TYPESCRIPT

```
const routes: Routes = [
  /* ① */
  { path: '', redirectTo: '/user',  pathMatch: 'full' },
  /* ② */
  {
    path: 'user',
    component: UserComponent,
    children: [
      /* UserComponent의 <router-outlet>에 표시 */
      { path: ':id', component: UserDetailComponent }
    ]
  },
  /* ③ */
  {
    path: 'customer',
    component: CustomerComponent,
    children: [
      /* CustomerComponent의 <router-outlet>에 표시 */
      { path: ':id', component: CustomerDetailComponent }
    ]
  }
];
```

위 라우트 구성의 ①, ②, ③은 모두 루트 컴포넌트의 `<router-outlet>` 영역을 위한 것이다. 즉, 루트 컴포넌트의 `<router-outlet>` 영역에는 UserComponent 또는 CustomerComponent이 표시된다.

children 프로퍼티는 자식 라우트를 구성할 때 사용한다. 라우트 구성의 component 프로퍼티에 선언된 컴포넌트(②의 경우, UserComponent)는 children 프로퍼티에 선언된 컴포넌트들(②의 경우, UserDetailComponent)의 부모 컴포넌트이다. 부모 컴포넌트는 루트 컴포넌트와는 별도의

`<router-outlet>`을 가지며 자식 컴포넌트는 부모 컴포넌트의 `<router-outlet>` 영역에 표시된다.

라우트 구성 ②의 의미는 아래와 같다.

- UserComponent는 루트 컴포넌트 AppComponent의 `<router-outlet>` 영역에 표시한다.
- UserDetailComponent는 부모 컴포넌트 UserComponent의 `<router-outlet>` 영역에 표시한다.

라우트 구성 ③의 의미는 아래와 같다.

- CustomerComponent는 루트 컴포넌트 AppComponent의 `<router-outlet>` 영역에 표시한다.
- CustomerDetailComponent는 부모 컴포넌트 CustomerComponent의 `<router-outlet>` 영역에 표시한다.

자식 라우트를 사용한 예제를 작성해보자. 앞서 설명한 자식 라우트를 포함한 라우트 구성을 그대로 사용할 것이다. 아래와 같이 프로젝트를 생성한다.

BASH

```
$ ng new children-routing -t -s -S
$ cd children-routing
```

2개의 부모 컴포넌트와 부모 컴포넌트의 `<router-outlet>` 영역에 표시할 2개의 자식 컴포넌트를 생성한다.

BASH

```
# 부모 컴포넌트
$ ng generate component user
# 자식 컴포넌트
$ ng generate component user/user-detail --flat
# 부모 컴포넌트
$ ng generate component customer
# 자식 컴포넌트
$ ng generate component customer/customer-detail --flat
```

루트 컴포넌트에 2개의 부모 컴포넌트(UserComponent, CustomerComponent)를 위한 내비게이션과 `<router-outlet>` 영역을 작성한다.

TYPESCRIPT

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

@Component({
  selector: 'app-root',
  template: `
    <nav>
      <a routerLink="/">User</a>
      <a routerLink="/customer">Customer</a>
    </nav>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {}
```

루트 URL(localhost:4200)로 접근하면 UserComponent가 `<router-outlet>` 영역에 표시되고 URL 패스가 '/customer'인 요청이 오면 CustomerComponent가 `<router-outlet>` 영역에 표시될 것이다. 이를 위해 루트 모듈에 라우트 구성을 추가하자.

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 { UserComponent } from './user/user.component';
import { UserDetailComponent } from './user/user-detail.component';
import { CustomerComponent } from './customer/customer.component';
import { CustomerDetailComponent } from './customer/customer-detail.componen
t';

// 라우트 구성
const routes: Routes = [
  { path: '', redirectTo: '/user',  pathMatch: 'full' },
```

```
    {
      path: 'user',
      component: UserComponent,
      // 자식 라우트
      children: [
        { path: ':id', component: UserDetailComponent }
      ]
    },
    {
      path: 'customer',
      component: CustomerComponent,
      // 자식 라우트
      children: [
        { path: ':id', component: CustomerDetailComponent }
      ]
    }
];

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

지금 상태에서는 2개의 부모 컴포넌트(UserComponent, CustomerComponent)를 위한 내비게이션과 `<router-outlet>` 영역은 루트 컴포넌트에 존재하지만 자식 컴포넌트를 위한 내비게이션과 `<router-outlet>` 영역이 존재하지 않는다. 자식 컴포넌트를 위한 네비게이션과 `<router-outlet>` 영역을 각각의 부모 컴포넌트에 추가하도록 하자.

먼저 UserComponent를 작성한다.

TYPESCRIPT

```typescript
// user/user.component.ts
import { Component, OnInit } from '@angular/core';

interface User {
  id: number;
  name: string;
}

@Component({
  selector: 'app-user',
  template: `
    <p>User List</p>
    <!-- 자식 컴포넌트를 위한 내비게이션 -->
    <ul>
      <li *ngFor="let user of users">
        <a [routerLink]="['/user', user.id]"
        [routerLinkActiveOptions]="{ exact: true }"
        routerLinkActive="active">{{ user.name }}</a>
      </li>
    </ul>
    <!-- 자식 컴포넌트를 위한 영역 -->
    <router-outlet></router-outlet>
  `,
  styles: [`
    a:hover, a.active { color: red; }
  `]
})
export class UserComponent implements OnInit {
  users: User[];

  ngOnInit() {
    this.users = [
      { id: 1, name: 'User-1' },
      { id: 2, name: 'User-2' },
    ];
  }
}
```

부모 컴포넌트에 자식 컴포넌트를 위한 네비게이션과 `<router-outlet>` 영역을 추가하였다. 자식 컴포넌트를 위한 `<router-outlet>` 영역에는 루트 모듈에서 지정한 라우트 구성의 children 프로퍼티의 설정에 따라 UserDetailComponent가 표시될 것이다.

TYPESCRIPT

```
// app.module.ts
...
  {
    path: 'user',
    component: UserComponent,
    // 자식 라우트
    children: [
      { path: ':id', component: UserDetailComponent }
    ]
  },
...
```

자식 컴포넌트 UserDetailComponent는 부모 컴포넌트의 routerLink 디렉티브에 의해 설정된 라우트 파라미터가 전달된다. 자식 컴포넌트 UserDetailComponent는 부모 컴포넌트가 전달한 라우트 파라미터를 취득하여 이를 가지고 서버로부터 상세한 정보를 추가 획득할 수 있다.

자식 컴포넌트 UserDetailComponent는 아래와 같다.

TYPESCRIPT

```
// user/user-detail.component.ts
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-user-detail',
  template: '<p>User Detail: ID {{ id }}</p>'
})
export class UserDetailComponent implements OnInit {

  id: number;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
```

```
    // 라우트 파라미터 취득
    this.route.paramMap
      .subscribe(params => this.id = +params.get('id'));
  }
}
```

CustomerComponent와 CustomerDetailComponent은 아래와 같다.

TYPESCRIPT

```
// customer/customer.component.ts
import { Component, OnInit } from '@angular/core';

interface Customer {
  id: number;
  name: string;
}

@Component({
  selector: 'app-customer',
  template: `
    <p>Customer List</p>
    <!-- 자식 컴포넌트를 위한 내비게이션 -->
    <ul>
      <li *ngFor="let customer of customers">
        <a [routerLink]="['/customer', customer.id]"
        [routerLinkActiveOptions]="{ exact: true }"
        routerLinkActive="active">{{ customer.name }}</a>
      </li>
    </ul>
    <!-- 자식 컴포넌트를 위한 영역 -->
    <router-outlet></router-outlet>
  `
})
export class CustomerComponent implements OnInit {

  customers: Customer[];

  ngOnInit() {
    this.customers = [
```

```
        { id: 1, name: 'Customer-1' },
        { id: 2, name: 'Customer-2' },
      ]
    }
  }
```

TYPESCRIPT

```
// customer/customer-detail.component.ts
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-customer-detail',
  template: '<p>Customer Detail: ID {{ id }}</p>'
})
export class CustomerDetailComponent implements OnInit {

  id: number;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    // 라우트 파라미터 취득
    this.route.paramMap
      .subscribe(params => this.id = +params.get('id'));
  }
}
```

#3. 모듈의 분리와 모듈별 라우트 구성

구성 요소를 모듈 단위로 구성하는 것과 동일하게 라우트도 모듈 단위로 구성할 수 있다.

루트 모듈 또는 AppRoutingModule에서는 전체 라우트 정보를 담고 있는 라우트 구성을 RouterModule의 forRoot 메소드의 인자로 전달하였다.

TYPESCRIPT

```
const routes: Routes = [ ... ];

@NgModule({
  ...
  imports: [ RouterModule.forRoot(routes) ],
```

```
  ...
})
```

모듈 단위로 라우팅 구성을 분리하는 경우, 분리한 모듈에 RouterModule의 forChild 메소드의 인자로 라우트 구성을 등록한다.

TYPESCRIPT

```
const routes: Routes = [ ... ];

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

앞서 살펴본 “자식 라우트”의 예제를 보면 프로젝트에는 루트 모듈만 존재한다. 기능별로 모듈을 분리하고 라우트 또한 모듈 단위로 분리하여 보자.

기능 모듈을 생성하기 위해 아래의 명령어를 실행한다.

BASH

```
$ ng generate module user
$ ng generate module customer
```

각 기능 모듈별로 라우트를 분리하기 위해 기능 모듈별로 라우트 모듈을 생성한다.

BASH

```
$ ng generate module user/user-routing --flat
$ ng generate module customer/customer-routing --flat
```

앞서 생성한 기능 모듈 UserModule에 라우트 모듈 UserRoutingModule과 소속 컴포넌트를 등록한다.

TYPESCRIPT

```typescript
// user/user.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { UserRoutingModule } from './user-routing.module';
import { UserComponent } from './user.component';
import { UserDetailComponent } from './user-detail.component';

@NgModule({
  imports: [
    CommonModule,
    /* 라우트 모듈의 등록 */
    UserRoutingModule
  ],
  declarations: [
    UserComponent,
    UserDetailComponent
  ]
})
export class UserModule { }
```

이제 기능 모듈 UserModule에 등록된 라우트 모듈 UserRoutingModule에 라우트를 구성한다. 해당 기능 모듈에 소속된 컴포넌트의 라우트 구성만을 추가하면 된다.

TYPESCRIPT

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

import { UserComponent } from './user.component';
import { UserDetailComponent } from './user-detail.component';

/* 기능 모듈 단위 라우팅 구성 */
const routes: Routes = [{
  path: 'user',
  component: UserComponent,
  children: [
    { path: ':id', component: UserDetailComponent }
```

```
    ]
  }];

  @NgModule({
    /* 기능 모듈 단위 라우터 등록  */
    imports: [ RouterModule.forChild(routes) ],
    exports: [ RouterModule ]
  })
  export class UserRoutingModule { }
```

이와 같은 방법으로 기능 모듈 CustomerModule과 라우트 모듈 CustomerRoutingModule을 작성한다.

TYPESCRIPT

```
// customer/customer.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { CustomerRoutingModule } from './customer-routing.module';
import { CustomerComponent } from './customer.component';
import { CustomerDetailComponent } from './customer-detail.component';

@NgModule({
  imports: [
    CommonModule,
    /* 라우트 모듈의 등록 */
    CustomerRoutingModule
  ],
  declarations: [
    CustomerComponent,
    CustomerDetailComponent
  ]
})
export class CustomerModule { }
```

TYPESCRIPT

```
// customer/customer-routing.module.ts
import { NgModule } from '@angular/core';
```

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

import { CustomerComponent } from './customer.component';
import { CustomerDetailComponent } from './customer-detail.component';

/* 기능 모듈 단위 라우팅 구성 */
const routes: Routes = [{
  path: 'customer',
  component: CustomerComponent,
  children: [
    { path: ':id', component: CustomerDetailComponent }
  ]
}];

@NgModule({
  /* 기능 모듈 단위 라우터 등록  */
  imports: [ RouterModule.forChild(routes) ],
  exports: [ RouterModule ]
})
export class CustomerRoutingModule { }
```

이제 루트 모듈 내부에 작성된 라우트 구성을 분리하기 위해 AppRoutingModule을 생성한다.

BASH

```
$ ng generate module app-routing --flat
```

루트 모듈에 작성되어 있던 라우트 구성을 생성된 AppRoutingModule로 분리한다. 이때 분리된 모듈의 라우트 구성은 제외한다.

TYPESCRIPT

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

// 라우트 구성
const routes: Routes = [
  { path: '', redirectTo: '/user',  pathMatch: 'full' }
```

```
];

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

루트 모듈에서 라우트 구성을 제거하고 AppRoutingModule을 임포트한다.

TYPESCRIPT

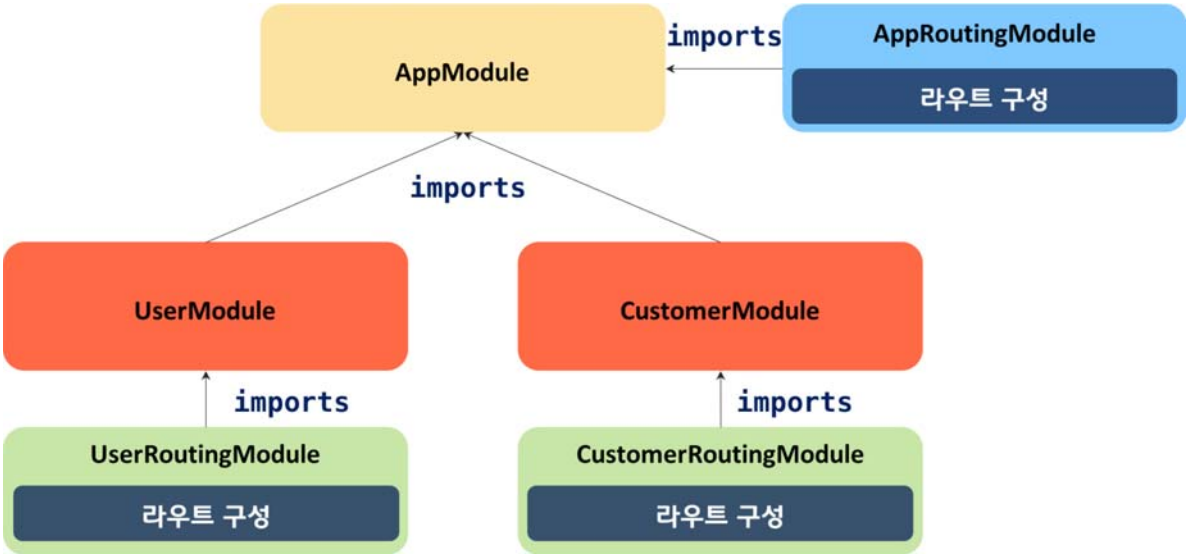
```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';

import { UserModule } from './user/user.module';
import { CustomerModule } from './customer/customer.module';

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

@NgModule({
  declarations: [ AppComponent ],
  imports: [
    BrowserModule,
    UserModule,
    CustomerModule,
    /* AppRoutingModule 등록 */
    AppRoutingModule
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

children-routing-seperate-module
By ungmo2

Run Project

루트 모듈은 AppRoutingModule을 등록하고 AppRoutingModule은 기능 모듈 UserModule과 CustomerModule을 등록하였다. 이때 AppRoutingModule은 루트 모듈을 위한 라우트 구성을 포함하며 AppRoutingModule에 등록된 2개의 기능 모듈은 각각의 기능 모듈을 위한 라우트 구성을 포함하였다.

이를 그림으로 표현하면 아래와 같다.



모듈의 분리와 모듈별 라우트 구성

4. 라우트 가드(Route Guard)

라우트 가드는 라우터를 통해 컴포넌트나 모듈을 활성화시킬 때 또는 컴포넌트에서 빠져나갈 때 권한 등을 체크하여 접근을 제어하는 방법이다. 예를 들어 사용자 인증을 하지 않은 사용자의 접근을 제어하거나 다른 뷰로 이동하기 이전에 저장하지 않은 사용자 입력 정보가 있다면 사용자에게 알릴 수 있다.

Angular는 가드를 위한 5개의 인터페이스를 제공한다.

4.1 CanActivate

CanActivate 가드는 라우트를 활성화할 수 있는지 결정한다. 주로 뷰로의 접근 권한을 체크하고 접근을 제어할 때 사용한다.

CanActivate 인터페이스를 구현하여 가드 클래스를 정의한다. 이때 CanActivate.canActivate 메소드는 접근 권한 체크 로직을 수행하고 true 또는 false를 반환한다.

TYPESCRIPT

```
// auth.guard.ts
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthService } from '../services/auth.service';

@Injectable()
```

```
export class AuthGuard implements CanActivate {

  constructor(private router: Router, private auth: AuthService) { }

  // 접근 권한 체크 로직을 수행하고 true 또는 false를 반환한다.
  canActivate() {
    // 토큰 유효성 확인
    if (!this.auth.isAuthenticated()) {
      // 토큰이 유효하지 않으면 로그인 페이지로 강제 이동
      this.router.navigate(['signin']);
      return false;
    }
    return true;
  }
}
```

가드는 모듈에 등록되어야 한다. 가드를 모듈에 등록하는 방법은 아래와 같다.

TYPESCRIPT

```
// 라우트 구성
...
  {
    path: 'user',
    component: UserComponent,
    canActivate: [ AuthGuard ] /* 가드에 의한 접근 제한 */
  },
...
```

라우트 구성에 canActivate 프로퍼티로 가드를 선언한다. 이때 UserComponent를 활성화하기에 앞서 가드가 실행되고 가드의 실행 결과에 따라 컴포넌트에의 접근을 제어한다. 즉, AuthGuard.canActivate 메소드의 실행 결과가 true일 경우에만 UserComponent를 활성화한다.

4.2 CanActivateChild

CanActivateChild 가드는 자식 라우트를 활성화할 수 있는지 결정한다. 주로 자식 컴포넌트로의 접근 권한을 체크하고 접근을 제어할 때 사용한다.

CanActivateChild 인터페이스를 구현하여 가드 클래스를 정의한다. 이때 CanActivateChild.canActivateChild 메소드는 접근 권한 체크 로직을 수행하고 true 또는 false를 반환한다.

TYPESCRIPT

```
// auth.guard.ts
import { Injectable } from '@angular/core';
import { Router, CanActivateChild } from '@angular/router';
import { AuthService } from '../services/auth.service';

@Injectable()
export class AuthChildGuard implements CanActivateChild {

  constructor(private router: Router, private auth: AuthService) { }

  // 접근 권한 체크 로직을 수행하고 true 또는 false를 반환한다.
  canActivateChild() {
    // 토큰 유효성 확인
    if (!this.auth.isAuthenticated()) {
      // 토큰이 유효하지 않으면 로그인 페이지로 강제 이동
      this.router.navigate(['signin']);
      return false;
    }
    return true;
  }
}
```

가드는 모듈에 등록되어야 한다. 가드를 모듈에 등록하는 방법은 아래와 같다.

TYPESCRIPT

```
// 라우트 구성
...
  {
    path: 'customer',
    component: CustomerComponent,
    canActivateChild: [ AuthChildGuard ], /* 가드에 의한 접근 제한 */
```

```
    children: [
      { path: ':id', component: CustomerDetailComponent }
    ]
  },
...
```

라우트 구성에 canActivateChild 프로퍼티로 가드를 선언한다. 이때 자식 컴포넌트 CustomerDetailComponent를 활성화하기에 앞서 가드가 실행되고 가드의 실행 결과에 따라 자식 컴포넌트에의 접근을 제어한다. 즉, AuthChildGuard.canActivateChild 메소드의 실행 결과가 true 일 경우에만 CustomerDetailComponent를 활성화한다.

4.3 CanLoad

CanLoad 가드는 모듈이 로드되기 이전에 모듈을 활성화할 수 있는지 결정한다. 애플리케이션이 처음 실행될 때 모든 모듈을 미리 컴파일하지 않고 호출 시점에 컴파일 하는 지연 로딩(Lazy Loading)을 사용하는 경우, CanLoad 가드는 접근 권한이 없는 모듈을 컴파일하지 않는다.

CanLoad 인터페이스를 구현하여 가드 클래스를 정의한다. 이때 CanLoad.canLoad 메소드는 접근 권한 체크 로직을 수행하고 true 또는 false를 반환한다.

4.4 Resolve

Resolve 가드는 각 라우트의 뷰가 렌더링되기 이전에 뷰가 렌더링되기 위해 반드시 필요한 데이터를 로딩할 때 사용한다.

Resolve 인터페이스를 구현하여 가드 클래스를 정의한다. 이때 Resolve.resolve 메소드는 true 또는 false가 아닌 뷰가 렌더링되기 위해 필요한 데이터를 반환한다.

4.5 CanDeactivate

CanDeactivate 가드는 뷰에서 빠져나갈 때 즉 컴포넌트가 비활성화될 때 사용한다.

CanDeactivate 인터페이스를 구현하여 가드 클래스를 정의한다. 이때 CanDeactivate.canDeactivate 메소드는 true 또는 false를 반환한다.

Reference

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

- CanActivate
- CanActivateChild
- CanLoad
- Resolve
- CanDeactivate