Angular Router Module and Guards

CS569 – Web Application Development II

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Routing

Routing means splitting the application into different areas usually based on rules that are derived from the current URL in the browser.

Defining routes in our application is useful because we can:

- Separate different areas of the app
- Maintain the state in the app
- Protect areas of the app based on certain rules

Client-Side vs Server-Side Routing

The server must have one route "/" to serve your SPA application. All other routes should be for API to persist/retrieve data. Additional "*" unknown route is required to redirect user to Angular app.

Client-side routing will have difference routes to display various component trees to the user. Angular changes the URL with pushState API to support (refresh, bookmark and shareability to your routes).

With client-side routing we're not necessarily making a request to the server on every URL change.

Why Changing the URL?

Because our app is client-side, it's not technically required that we change the URL when we change pages (no HTTP requests). But what would be the consequences of using the same URL for all pages?

- You wouldn't be able to **refresh** the page and keep your location within the app
- You wouldn't be able to **bookmark** a page and come back to it later
- You wouldn't be able to **share the URL** of that page with others

Components of Angular routing

There are three main components that we use to configure routing in Angular:

Routes describes the routes our application supports. Will be passed to `RouterModule` and imported in our `NgModule`.

RouterOutlet is a placeholder component that gets expanded to each route's content.

RouterLink directive is used to link to routes so our browser won't refresh when we change routes.

RouterModule

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

const MY_ROUTES: Routes = [
        { path: '', redirectTo: 'home', pathMatch: 'full' },
        { path: 'home', component: HomeComponent },
        { path: 'about', component: AboutusComponent },
        { path: 'aboutus', redirectTo : 'about' },
        { path: '**', redirectTo : 'home' }
];
@NgModule({ . . .
        imports: [ BrowserModule, RouterModule.forRoot(MY_ROUTES) ]
                  . . . })
export class AppModule { }
```

RouterLink [routerLink]

If we tried creating links that refer to the routes directly using pure HTML, it will result to links when clicked they trigger a GET request and cause a **page reload**

```
<a href="/home">Home</a>
```

To solve this problem, we will use the **routerLink** directive:

```
<a [routerLink]="['home']">Home</a>
```

Router Outlet `<router-outlet>`

When we change routes, we want to keep our outer layout template and only substitute the inner section of the page with the route's component.

In order to describe to Angular **where** in our page we want to render the contents for each route, we use the **router-outlet** directive.

We are going to use our AppComponent as a layout.

Example of Layout Component

```
@Component({
    selector: 'AppComponent',
    template: ` <div>
        <nav>
          <ul>
             <li><a [routerLink]="['home']">Home</a></li>
             <li><a [routerLink]="['about']">About</a></li>
             <li><a [routerLink]="['contact']">Contact us</a></li>
          </ul>
        </nav>
        <router-outlet></router-outlet>
        </div> `
})
class RoutesDemoApp {}
```

> Using [routerLink] will instruct Angular to take ownership of the click event and then initiate a route switch to the right place, based on the route definition.

Base Tag `<base href="/">`

This tag is traditionally used to tell the browser where to look for images and other resources declared using **relative paths**.

Angular Router also relies on this tag to determine how to construct its routing information.

If we have a route with a path of `/hello` and our base element declares `<base href="/app">`, the application will use `/app/hello` as the concrete path.

Routes and Component Lifecycle

Angular Router is efficient when it comes to components lifecycle.

It only loads the component once it's been asked for, once created, it does not destroy it and recreate a new one the next time you ask for the same component, instead, **it will reuse the same instance** that you already have in memory (**snapshot**).

Route Parameters – Mandatory Params

We can specify that a route takes a parameter by putting a colon in front of the path segment like this `/route/:param`

```
const routes: Routes = [{ path: 'articles/:id', component: ArticlesComponent } ];
```
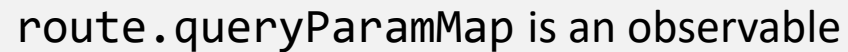
In order to read route parameters, we need to first import `ActivatedRoute` Service and we inject it into the constructor of our component.

Example

```
constructor(private route: ActivatedRoute) {
    route.paramMap
        .pipe(
            // Mimicking HTTP call with of() Observable
            flatMap((params: ParamMap) => of(params.get('id')))
         )
        .subscribe( id => { this.id = id });
}
```

Query Parameters – Optional Query Params

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

route.queryParamMap is an observable

Observable Subscription

You **must subscribe** to an observable in order to listen to its events.

When subscribing to the observable we generate a Subscription, after we finish from the component, this subscription will stay alive and will cause **memory leak**. You should always destroy your observable subscription.

You have to `unsubscribe()` from the Observable in the `ngOnDestroy` method/lifecycle hook

Example

```
import { Component, OnDestroy } from '@angular/core';
import { ActivatedRoute } from "@angular/router";
import { Subscription } from "rxjs";

@Component({ selector: 'component', template: `...` })
export class Component implements OnDestroy{
    private subscription: Subscription;
    id: string;
    constructor(private route: ActivatedRoute) {
        this.subscription = route.params.subscribe(
                        (param: any) => this.id = param['id'];
        );
    }
    ngOnDestroy() {
        this.subscription.unsubscribe();
    }
}
```

Imperative Routing

You can also navigate to a route imperatively (in your code), you need to inject the **Router** service then you may call **navigate()** like this:

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

constructor(private router: Router) {}

this.router.navigate(['home'])
```

> *Remember:* Angular Services don't need to be provided. Angular knows how to create an instance of its services.

Passing Data between Routes

Imperative navigation using the **router.navigateByUrl()** method

```
@Component({
    template: `<a (click)="navigateWithState()">Go</a>`,
})
export class AppComponent {
    constructor(public router: Router) {}
    navigateWithState() {
      this.router.navigateByUrl('/details', { state: { hello: 'world' } });
    }
}
```

Declarative navigation using the **routerLink** directive

```
@Component({
    template: `<a [routerLink]="['details']" [state]="{ hello: 'world' }">Go</a>`
})
export class AppComponent {}
```

Reading the State

Read it from the NavigationExtras property of the NavigationStart event. This one is useful in top level components, because you cannot listen to the NavigationStart event from inside the component that you are navigating to.

```typescript
@Component({
    selector: 'app-root',
    template: `<pre>{{ msg }}</pre>`
})
export class AppComponent implements OnInit {
    private msg: string = '';
    constructor(public router: Router) {
        this.msg = this.router.getCurrentNavigation().extras.state.hello;
    }
```

Router Link Best Practices

```
<a [routerLink]="['users', 'update', id.value]">Update</a>
// users/update/1
<a [routerLink]="['users', 'update']" [queryParams]="{id: id.value}">Update</a>
// users/update?id=1
```

```
this.router.navigate(['users', 'update'], { queryParams: { id: id.value} })
// users/update?id=1
```

Nested Routes

Nested routes is the concept of containing routes within other routes. With nested routes we're able to encapsulate the functionality of parent routes and have that functionality apply to the child routes.

We can have multiple, nested `router-outlet`. So each area of our application can have their own child components, that also have their own `router-outlet`.

```
const MY_ROUTES: Routes = [
        { path: 'parent', component: ParentComponent,
          children: [
                { path: 'child', component: ChildComponent }
           ]
        }];
```

It has an outlet, so its children are rendered in it

Router Hooks

There are times that we may want to do some actions when changing routes (for example authentication).

Let's say we have a login route and a protected route. We want to only allow the app to go to the protected route if the correct username and password were provided on the login page. In order to do that, we need to **hook into the lifecycle of the router** and ask to be notified when the protected route is being activated. We then can call an authentication service and ask whether or not the user provided the right credentials.

Guards

Guards are useful **Services** which allow you to control access to and from a Route/Component.

canActivate called when you are serving into the route

canDeactivate called when leaving the route.

Guard classes have to be registered in **providers[]**, why?

```
const ROUTES: Routes = [
        { path: 'my-path', component: MyComponent,
          canActivate: [Guard1, Guard2..],
          canDeactivate: [Guard3, Guard4..] }
];
```

Example CanActivate

```typescript
import { CanActivate,
         RouterStateSnapshot,
         ActivatedRouteSnapshot } from "@angular/router";
import { Observable } from "rxjs";

export class MyGuard implements CanActivate {
    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
    Observable<boolean> | boolean {

            // Your logic goes here
            // return true to continue
            // otherwise, you will have to redirect to another route
            return confirm('Are you sure?');
        }
}
```