

Angular Router

SD 555 – Web Application Development III

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

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

Angular Router

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

- Routes** describe a map between URLs and components.

- Router Outlet** is a placeholder where components are mounted and unmounted.

- The routerLink** directive is used instead of href attribute to link to routes so our browser won't refresh when we change routes.

Angular Router Setup

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

```
const appRoutes: Routes = [  
  { path: '', redirectTo: 'home', pathMatch: 'full' },  
  { path: 'home', component: HomeComponent, title: 'Homepage' },  
  { path: '**', redirectTo : 'home' }  
];
```

Sets up providers necessary to enable Router functionality for the application.

```
bootstrapApplication(AppComponent, {  
  providers: [provideRouter(appRoutes)]  
});
```

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.

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>
```

Styling Active Router Links

To make our link have an extra CSS style when its route is being activated we use **routerLinkActive** Directive:

```
@Component({
  imports: [RouterLink, RouterLinkActive],
  template: `
    <a [routerLink]="['users']"
      routerLinkActive="active"
      [routerLinkActiveOptions]="{exact:true}">Home</a> `
})
export class AppComponent {}
```

Will add an attribute `class="active"` to the anchor component once its route is activated

Example of Layout Component

```
@Component({
  selector: 'AppComponent',
  imports: [RouterOutlet, RouterLink],
  template: `
    <nav>
      <ul>
        <li><a [routerLink]="['home']">Home</a></li>
        <li><a [routerLink]="['products']" [queryParams]="{ page: 1 }">About</a></li>
        <li><a [routerLink]="['contact']">Contact us</a></li>
      </ul>
    </nav>
    <router-outlet />`
})
class AppComponent {}
```

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.

Bind Route Parameters to Inputs

The binding only works for routed components. This replaces the need for **ActivatedRoute** service.

The binding also works with resolved data and query parameters.

```
provideRouter(routes, withComponentInputBinding())
```

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:

```
#router = inject(Router);
```

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

```
this.#router.navigate(['users'], { queryParams: { page: 2 } }) // users?page=2
```

View Transitions API

The View Transitions API enables smooth transitions when changing the DOM. The feature uses the browser's native capabilities for creating animated transitions between routes.

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter, withViewTransitions } from '@angular/router';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withViewTransitions()),
    provideAnimationsAsync()
  ]
};
```

Defer loading of the animations module: **provideAnimationsAsync()** allows you to lazily load the animation module and shaves 60KBs from your initial bundle.

Router Scroll Position Restoration

You may configure the router to remember and restore scroll position as the user navigates around an application. New navigation events will reset the scroll position, and pressing the back button will restore the previous position.

To turn on restoration in the router configuration:

```
provideRouter(appRoutes,  
  withInMemoryScrolling({scrollPositionRestoration: 'enabled'})  
)
```

Dynamic Page Title

Use the Title service to set your page title dynamically as follows:

```
@Component({
  selector: 'app-root',
  template: `...`
})
export class AppComponent {
  #title = inject(Title);

  constructor() {
    this.#title.setTitle('Theo');
  }
}
```

Lazy Loading Component

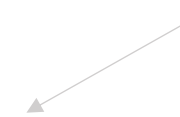
```
export const routes: Routes = [  
  {  
    path: '/login',  
    component: LoginComponent  
  },  
  {  
    path: '/signup',  
    loadComponent: () => import('./pages/signup.component').then(c => c.SignupComponent);  
  },  
];
```

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

Lazy Loading Several Standalone Components

We can accomplish this by adding a route with **loadChildren** in the root routing file but importing the routes directly, without any intermediary NgModule.

```
export const routes: Routes = [  
  {  
    path: 'employees',  
    loadChildren: () => import('./employees.routes').then(r => r.routes);  
  },  
];
```

```
export const routes: Routes = [  
  { path: 'list', component: EmployeeListComponent },  
  { path: 'details/:id', component: EmployeeDetailsComponent },  
  { path: 'create', component: CreateEmployeeComponent },  
  { path: 'edit', component: EditEmployeeComponent },  
];
```

Preloading Strategy

You have the option to either preload all the lazy loading routes or implement a custom preloading strategy to selectively preload specific routes.

To preload all the lazy loading routes:

```
export const appConfig: ApplicationConfig = {  
  providers: [provideRouter(routes, withPreloading(PreloadAllModules))]  
};
```

Deferrable Views

The deferrable views **lazily load the list of comments** and all their transitive dependencies via a compile-time transformation. Angular finds all components, directives, and pipes used inside of a `@defer` block, generates dynamic imports, and manages the process of loading and switching between states.

```
@defer (on viewport) {  
  <comment-list/>  
} @loading {  
  Loading...  
} @error {  
  Loading failed :(  
} @placeholder {  
  <!-- A placeholder content to show until the comments load -->  
    
}
```

Deferrable Views Triggers

`on idle` — lazily load the block when the browser is not doing any heavy lifting.

`on immediate` — start lazily loading automatically, without blocking the browser.

`on timer(<time>)` — delay loading with a timer.

`on viewport / on viewport(<ref>)` — the reference is for an anchor element,
lazily load the component and render it when the anchor element is visible.

`on interaction / on interaction(<ref>)` — initiate lazy loading when the user
interacts with a particular element.

`on hover / on hover(<ref>)` — triggers lazy loading when you hovers an element.

`when <expr>` — specify your own condition via a `boolean` expression.

Prefetch Deferrable Views

Deferrable views also provide the ability to prefetch the dependencies ahead of rendering them. Adding prefetching is as simple as adding a **prefetch** statement to the defer block and supports all the same triggers.

```
@defer (on viewport; prefetch on idle) {  
  <comment-list />  
}
```

Providing Dependencies Only to Certain Routes

It is possible to add a **providers** array to a route definition. It just has to be not marked as **providedIn: 'root'**.

```
{  
  path: 'employees',  
  providers: [EmployeeService],  
  loadChildren: () => import('./employees.routes').then(r => r.routes);  
},
```

This will make the **EmployeeService** only provided in the routes that reside inside **employees.routes.ts**, meaning only components that are routed in that configuration file will have access to this particular instance of **EmployeeService**.

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 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.

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


Guard Example

For a reusable guard, generate one with: `ng generate guard MyGuard`

or pass one directly

```
{  
  path: 'admin',  
  canActivate: [(route, state) => inject(LoginService).isLoggedIn()]  
}
```

`canActivate` is used to prevent unauthorized users from accessing certain routes, it returns a `boolean` indicating whether to proceed or not.

canMatch Guard

The `CanMatch` guard controls whether we can use the route and whether we can download the code. In addition, when one of the defined guards returns `false`, the route is skipped, and other routes are processed instead. It can be used to Load different components based on the user role.

```
const routes: Routes = [  
  {  
    path: 'profile',  
    canMatch: [() => inject(AuthService).isRole('admin')],  
    loadComponent: () => import('./admin.component').then(c => c.AdminComponent)  
  },  
  {  
    path: 'profile',  
    loadComponent: () => import('./user.component') .then(c => c.UserComponent)  
  }  
];
```

redirectTo

if you'd like to redirect to a route that depends on some runtime state, **redirectTo** may accept a function that returns a string.

```
const routes: Routes = [  
  { path: "first-component", component: FirstComponent },  
  {  
    path: "old-user-page",  
    redirectTo: ({ params, queryParams }) => {  
      const userIdParam = queryParams['userId'];  
      if (userIdParam !== undefined) {  
        return `/user/${userIdParam}`;  
      } else {  
        return `/not-found`;  
      }  
    },  
  },  
  { path: "user/:userId", component: OtherComponent },  
];
```