**Routing**

Inside the *elements-routing.module.ts* add to the routes constant the url path and the component that needs to be loaded. Eg:

const routes: Routes = [

  { path: 'elements', component: ElementsHomeComponent }

];

The app module uses all the routing rules included in the module. So, when we navigate to this url, we see what the module specifies

The <router-outlet> of the app.component.html ‘reads’ any URL and searches if there is any module/component with that name and then renders it inside the outlet.

When creating anchor elements, we need to use the ***routerLink*** attribute, because otherwise the page refreshes and reloads all resources. The **route*rLink***navigates inside the router and decides which module to show inside the *outlet.*

The *routerLinkActive* specifies which class will be applied when the link is active.

**Routing Hierarchy**

In the *app.module.ts*, rearrange the imports as needed. The *AppRoutingModule* should go last to allow any other routing first.

**Using the router inside components/modules**

1. Inject the constructor with a parameter of type *Router.*
2. Use the property wherever needed.

**LAZY LOADING**

1. Select which modules should be lazy loaded
2. Remove any imports of these elements anywhere
3. In the *AppRoutingModule*, define a Route in the *‘routes’* array to specify when to load that module.

{ path: 'anyElement',

    loadChildren: () => import('./modules/elements/anyElement.module').then((*m*)=>*m*.anyElementModule)

  },

1. Modify the routing in the modules themselves, because the routing rules are cumulative. When navigating to the element root, the path should be empty

  { path: '', component: ElementsHomeComponent }

**GUARDS**

1. Generate a new guard by ng g guard <type>
2. If it is of *auth* type, select which kind from the console prompt.
3. Add the logic with which the guard will operate.

eg.

canActivate(): Observable<boolean> {

    return this.accountService.currentUser$.pipe(

      map(user => {

        if(user) return true;

        this.toastr.error("Thou shall not pass!")

      })

    )

  }

1. Add it to any route that needs to use it.
2. If many routes need to use the same guards, add a mock route in the routing module and specify there whether any and which guards should run and the array of the children routes that they should be applied to. eg.

{

    path:'',

    runGuardsAndResolvers: 'always',

    canActivate: [SomeGuard],

    children:[

      { path: 'route', component:SomeComponent},

      { path: 'otherRoute', component:SomeOtherComponent}

    ]

  },