Changing Pages with Routing

Module Introduction

- **Routing** allows us to change the URL despite continuing to use one page
- The page may appear so different that it appears totally separate from the homepage to the user

Why Do We Need A Router?

- Allows us to create "different" pages to better organize and display our data

Setting Up and Loading Routes

- Routes are responsible for navigation and overall structure of the application
- Since routes are such a core component, it's wise to register them inside **AppModule**
  - Above **@NgModule**, create an array of **Routes**
    - Route fields
      - **Path** → Don't include the URL base or **/**
      - **Component** → Typically a component designated for loading
  - Example → **const appRoutes: Routes = [{ path: 'home', component: 'HomeComponent' }]**
- To register routes, we must also import **RouterModule.forRoot(<route-array>)** inside the **imports** array of **AppModule**
- To load a routed component, we insert **<router-outlet>** into the appropriate HTML location

Navigating with Router Links

- One approach is to assign the proper paths to the **href** of navigation-based buttons
  - This reloads the entire page/app as opposed to fetching the target component → **Bad**
  - This is because a new request goes to the server

- Instead, we should use the **routerLink** directive
  - This parses a string, telling Angular that it's handling a link
  - Can also be used with **property-binding**, passing a string/array to specify all path segments
  - Catches the click element, prevents the default request (no reloading), and analyzes the path to see if it's in our defined route list

Understanding Navigation Paths

- Adding a **/** at the beginning of a **routerLink path** denotes that you're looking for a path starting from root → **Absolute Path**
- Not including a **/** at the beginning of a **routerLink path** tells Angular to add the new routing information to the end of the current path → **Relative Path**

Styling Active Router Links

- To get a visual indication of our current route (such as darkening the nav-button's background), we apply the **routerLinkActive** directive to the element containing the **routerLink** attributed element → **li** > a[routerLink]
  - Assign **active** to it to signal that it's the current path
- Be conscious of the **/ (empty-path)**
  - This will still have the styling even if you're on another page because the **/** will always be active
  - To remedy this, apply **[routerLinkActiveOptions]="{ exact: true }"** as a sibling attribute to **routerLinkActive="active"** from the parent element
    - This tells Angular to only apply the styling class if the full path matches the given path

Navigating Programmatically

- Suppose we have a button in a template that, when clicked, executes a method in the component that navigates us to another page
- We must **inject** the component (via the **constructor**) with **Router**
- To navigate with **Router**, we call its **navigate** method

    - This method takes an array of strings containing the desired path
    - Example → **this.router.navigate(['/servers'])**

Using Relative Paths in Programmatic Navigation

- Remember, not including **/** in a path entails that you're using a **relative path**
- To tell Angular our current location, we must include a navigation action as a second parameter
    - First, inject an **ActivatedRoute** into the component (via the **constructor**)
        - Keeps meta information about the current route
    - In this case, **{ relativeTo: this.<activated-route> }**

Passing Parameters to Routes

- We can specify parameters in a route by adding **:<param>** inside of the route's path
- We can fetch this value by using the parameter's name
- Example → **{ path: 'users/:id', component: UserComponent }**
    - Be conscious that any route one layer above **users** (in the example above) will have an interpreted value of **id**

Fetching Route Parameters

- If a route contains a **parameter** (say an **id**), then we can refer to that **parameter** in the route's designated **component**
- To access the **parameter**, we must first access the currently-loaded route by **injecting ActivateRoute** (via the **constructor**)
    - This contains the route's metadata, including variables and their values
    - Within the **ActivatedRoute** we find the **parameter** with **this.<activated-route>.snapshot.params['<parameter-name>']**

Fetching Route Parameters Reactively

- Not every route parameter can be fetched with **this.<activated-route>.snapshot.params['<parameter-name>']**

- Another method involves using **routerLink** with an array of string
  - Example → **[routerLink]="['<path>, <param-1>, <param-2>]"**
  - Better since we can more easily structure our path
  - Changes the URL, but doesn't change the page's content
    - This is because we're already on the component and doesn't instantiate it again
- Another, better method involves using the **params observable** in **ngOnInit**
  - **Observables** let us easily work with asynchronous tasks
  - Example → . . .
    this.<ActivatedRoute>.params.subscribe(
    (params: Params) => this.<variable>.<param> = params['<param>'];
    );
- **Subscribing** should be our general approach, only using **snapshot** if we know the current component will never need to be reloaded

An Important Note about Route Observables

- Angular **cleans** the subscription every time the component is **destroyed**
- It may be good to implement **ngOnDestroy**, unsubscribing from the subscription there
  - Create a new variable of type **Subscription**
  - Assign the subscription call to the **Subscription** variable
  - Call **this.<subscription-variable>.unsubscribe()** in **ngOnDestroy**

Passing Query Parameters and Fragments

- **Query Paramameters** are denoted in the URL by **?**'s
- **Hash Fragments** are denoted with **H**'s
- Example → **. . ./Anna?mode=editing#loading**
- How to pass this information via **routerLinks**?
  - Create a **[queryParams]** property in the same element you have the **[routerLink]**

        - Assignment this property to an object containing the parameter name and designated value
        - Example → **[queryParams] = "{ allowEdit: '1' }"**
    - Create a **fragment** attribute in the same element you have the **[routerLink]**
        - The value is whatever you wish to pass
        - Example → **fragment = "loading"**
- How to pass this information **programmatically**?
    - When clicking a button to navigate to a page, pass some sort of **id** as an argument
    - Within the component's **navigate** method, pass the **id** during your construction of the URL
    - For **queryParams** and **fragments**, create a new object containing:
        - **queryParms: { allowEdit: '1' }**
        - **fragment: 'loading'**

Retrieving Query Parameters and Fragments

- **Inject ActivatedRoute** within the **constructor**
- Like before, users can access parameters with **snapshots** or
    - **Snapshot**
        - **this.route.snapshot.queryParams**
        - **this.route.snapshot.fragment**
    - **Subscriptions**
        - **this.route.queryParams.subscribe()**
        - **this.route.fragment.subscribe()**

Practicing and Some Common Gotchas

- We can dynamically instruct links with **[routerLink]** and **objects** in the template if we define **objects** using ***ngFor**

Setting Up Child (Nested) Routes

- In the "parent" routes of the URL, add a **children** attribute
    - An array of routes in which the path doesn't contain the "parent" route

    - Example → **. . .**
    - **{ path: 'servers', component: ServersComponent. children: [**
      { path: ':id', component: ServerComponent }
      ] }
- All child routes displayed by adding **<router-outlet>** to the template of the parent route

Configuring the Handling of Query Parameters

- If you're navigating via a button, you may lose your initial URL qualities by moving again
- To preserve those, appened an object containing **queryParamsHandling** to the **navigate** function
    - **'merge'** → Merge old query params with new query params
    - **'unmerge'** → Dump the old query params, only keeping the new ones
    - **'preserve'** → Keep all old query params, not adding new ones

Redirecting with Wildcard Routes

- Attempting to access nonexistant routes may lead to trouble
- To resolve this, create a new **path** in **AppModule**
    - Assign its **path** field to the **wildcard** → ******
    - This catches every path that isn't defined
    - Must be the last path defiend; else, it may be caught too early and supersede legitimate routes
- To redirect, add **redirectTo: <path>** instead of a **component**
    - Be careful of assigning **''** to path as that is always caught as the base path
    - Add **pathMatch: 'full'** to force the checking of a full path

Outsourcing the Route Configuration

- If you have more than 2-3 routes, insert them into a new module titled **app-routing.module.ts**
- Add the **@ngModule** decorator to the class
    - Includes an object with two fields

        - **imports: [RouterModule.forRoot(<route-array-name>)]**
        - **exports: [RouterModule]**
        - Remove the default **declarations** field all route components have already been declared in **AppModule**
- Remove the **RouterModule** from **AppModule**'s **imports** array
- Add **AppRoutingModule** to **AppModule**'s **imports** array

An Introduction to Guards

- **Route Guards** pertain to code occurring when the component is entered or departed from

Protecting Routes with canActivate

- **canActivate** allows us to run certain code within a specific period of time
- Create a service (**authGuard**) at the **app** level → Preferably drop **Service** from the selector
- Implement the **CanActivate** interface
    - Forces you to include the **canActivate** method
        - **canActivate** parameters
            - **ActivatedRouteSnapshot** → The currently activated route
            - **RouterStateSnapshot** → A tree of ActivatedRouteSnapshots
        - **canActivate** return types
            - **Observable<boolean>** → Good for asynchronous calls
            - **Promise<boolean>** → Good for asynchronous calls
            - **boolean** → Good for synchronous calls
- So much promise shenanigans → Consult the code for details
- Guard components by adding the **canActivate: [<guard-name>]** attribute and value to its specific route
    - Guarded routes are only accessible if **canActivate** returns true
- Additionally, add all guards to the **providers** array of **AppModule**

Protecting Child (Nested) Routes with canActivateChild

- Similar to **CanActivate**, add **CanActivateChild** to the guard service file

- Takes **ActivatedRouteSnapshot** and **RouterStateSnapshot** as parameters, and returns **Observable<boolean>, Promise<boolean>, or boolean**
- Call **this.canActivate(<route>, <state>);** within the body
- In the routing module, replace the parent path's **canDeactivate: [AuthGuard]** with **canActivateChild: [AuthGuard]**
    - Only protects the child route, leaving access to the parent route

Controlling Navigation with canDeactivate

- There may be times when users accidentally navigate away from a page they wish to remain on
    - For instance, what if they're inputting data when a misclick fires them home
- There's a lot happening here
- Similar to **canActivate**, add **canDeactivate: [CanDeactivateGuard]** inside the desired route of **AppRoutingModule**

Passing Static Data to a Route

- Routes may receive data **statically** or **dynamically**
- **Static Data**
    - In **AppRoutingModule**, we can add the **data** attribute to each route
    - The **data** attribute takes any properties we want as **key/value** pairs
    - We can retrieve this data using **ActivatedRoute** (via **injection**) and **this.<activated-route>.snapshot.data[<key>]**
- **Dynamic Data**
    - Perform the first two steps above
    - From there, subscribe to the **ActivatedRoute** instance and access information via **data[<key>]**

Resolving Dynamic Data with the Resolve Guard

- **Resolvers** allow us to run specified code before the assigned route is rendered
    - Essentially **preloads** data before rendering the component

- Creating a pseudo-resolver (not **ng g r**) entails creating a class, exporting it, implementing **Resolve**, and invoking the **resolve** function
  - The **resolve** function has two parameters: **ActivatedRouteSnapshot** and **RouterStateSnapshot**
  - The **resolve** function returns **Observable<type> | Promise<type> | <type>**
  - The **resolve** function's body returns code
- The **resolver** class should be added to the **providers** array of **AppModule**
- The route receiving the resolver should possess a **resolve: { server: ServerResolver }** property
- Within the receiving component, the **resolver**'s data is accessed via **this.<activated-route>.data**

Understanding Location Strategies

- In the case of a 404 error, Angular must return the **index.html** file
  - This is because all URLs are parsed by the server first, not Angular
    - Can lead to big trouble
  - We want Angular to take over and parse the route
- To remedy this, we include **{ useHash: true }** inside **RouterModule.forRoot(. . .)** of **AppRoutingModules**
  - This adds a **#** to our URL right after the base webpage section
  - This informs the web server that it should only care about the URL section before the **#**
  - The second part can be parsed by Angular
- Not as clean → There are better options