Routing

Introduction to Angular 6

Router NgModule

- provides a service that lets you define a navigation path among the different application states and view hierarchies in your app.
- It is modeled on the familiar browser navigation conventions:
  - Enter a URL in the address bar and the browser navigates to a corresponding page.
  - Click links on the page and the browser navigates to a new page.
  - Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

Router NgModule

- The router maps URL-like paths to views instead of pages.
- When a user performs an action that would load a new page in the browser
  - the router intercepts the browser's behavior, and shows or hides view hierarchies.
- If the router determines that the current application state requires particular functionality
  - and the module that defines it hasn't been loaded, the router can lazy-load the module on demand.

Navigation rules

- The router interprets a link URL according to your app's view navigation rules and data state.
  - You can navigate to new views in response to some other stimulus from any source.
  - The router logs activity in the browser's history, so the back and forward buttons work as well.
- To define navigation rules, you associate navigation paths with your components.
- A path uses a URL-like syntax that integrates with your program data
  - You can then apply program logic to choose which views to show or to hide, in response to user input and your own access rules.

LAB

Setting up for routing

- Set up index.html to provide enough context to Angular on how to set up its navigation
  - using the base tag within the head element in index.html
- Define a separate routes module file instead of defining it in the same app.module.ts.
  - This is generally good practice even if you only have a few routes initially.
- Better to define a separate module and routes for each feature
  - This would also allow us to lazy load feature modules and certain routes instead of loading all our code up front.

Setting up for routing

- While importing the RouterModule
  - we mark it for the root module by calling the forRoot method on it with the routes we are defining.
- Each route involves a configuration that defines the path for the route
  - as well as the component to be loaded when the route is loaded

Loading and navigating

- Mark out where Angular is to load the components when a certain route or path is matched
  - using the RouterOutlet directive that is made available as part of the RouterModule.
- Replace the href links with an Angular directive routerLink.
  - This ensures that all navigation happens within Angular.
- The Angular directive, routerLinkActive, adds the argument passed to it as a CSS class when the current link in the browser matches the routerLink directive.
  - It is a simple way of adding a class when the current link is selected.

Handling initial load and catchall

- Match the empty path and ask Angular to redirect us to the login route.
- For any path, we can redirect to another already defined path as well instead of reloading a component
- The default pathMatch is prefix
  - This would check if a URL starts with the given path
- The pathMatch key when set to full
  - ensures that only if the remaining path matches the empty string do we redirect to the login route.
  - Both the ordering of routes as well as the pathMatch value is important.
- Our catch-all route is added by matching the path **.
  - On matching this route, we have the option of loading a component or redirecting again to another route.

Routing requirements

- A path can include a variable in the URL
  - This can change based on which item needs to be loaded.
- The ActivatedRoute is a context-specific service that holds the information about the currently activated route, and knows how to parse and retrieve information from it.
- Possible to inject an instance of the Angular Router into our constructor, which gives us the capabilities to navigate within our application

Routes with parameters

- There are routes where we might want additional params that may or may not be optional.
  - E.g. the current page number, the page size
- Use the snapshot from the ActivatedRoute to read our parameters in the ngOnInit
  - This is OK if the component is loaded only once, and we navigate from it to another component and route.
- If there is a chance that the same component might need to be loaded with different parameters
  - we can treat the parameters and query parameters as an observable, just like service calls and HTTP requests.
- This way, the subscription will get triggered each time the URL changes, allowing us to reload the data rather than relying on the snapshot.

Route guards

- Route guards are a way of protecting the loading or unloading of a route based on your own conditions.
  - Provide flexibility in the kinds of checks you want to add before a route opens or closes.
- We can create an authentication guard which will kick in before we open a protected route.
  - The guard will then decide whether we can continue on to the route, or if we need to redirect to a different route

canActivate

- canActivate method.
  - This can return either a boolean or an Observable <boolean>.
  - If it resolves to true, then the route will activate, otherwise it won't.
- canActivate can also return an observable or a promise
  - allowing server calls to decide whether or not to proceed.
  - Angular will wait for the service to return before making a decision on whether or not the route should activate.
- Another alternative is to preserve the URL we were trying to open.
  - Once the user successfully logs in, we can then redirect to the saved URL rather than the default.

canDeactivate

- most commonly used to prevent the user from losing data by navigating away unintentionally from a form page
  - or to autosave the data when the user navigates away from a page
  - also possible for logging and analytics.
- canDeactivate guard is slightly different from the canActivate guard.
  - Deactivation is in context of an existing component
  - state of that component (for e.g. form state and route state) is usually important in deciding whether or not the router can deactivate the component and route.

Preloading data

- There might be cases where we want to make the service call to fetch its data before the component loads.
- Similarly, we might want to check if the data exists before even opening up the component.
- In these cases, it might make sense for us to try to prefetch the data before the component itself.
- In Angular, we do this using a Resolver which implements the Resolve interface