JAVA FULL STACK DEVELOPMENT PROGRAM

Angular Pipe and Router

OUTLINE

- Angular router
  - Router Module
  - Configuring Routes
  - Passing data to routes
  - Guarding routes
  - Developing a SPA with multiple router outlets
  - Splitting an app into modules
  - Lazy-loading modules

- Pipes
  - Introduction of Pipes
  - Built-in pipes
  - Custom pipes

SINGLE PAGE APPLICATION

- SPA (Single Page Application) — In a single-page application, we change what the user sees by showing or hiding portions of the components, rather than going out to the server to get a new page.
    - For Angular, we only have one *index.html* file, with a CSS bundle and a JavaScript bundle
    - These 3 static files can be uploaded to any *static* content server like Apache, Nginx, Amazon S3 or Firebase Hosting
    - When we need to display different data, only part of the page it getting reloaded.
- In single application page, how can we dynamically load the view we need?

ROUTING

- The Router enables navigation by interpreting a browser URL as an instruction to change the view. When ever we change the URL, we will load different views.
- We can build the single page application(SPA) and still have dynamic URL with the help of Router.
- Routing allows you to:
  - Maintain the state of the application
  - Implement modular applications
  - Dynamically load the view
  - Implement the application based on the roles (certain roles have access to certain URLs)
  - Pass optional parameters to the View

ROUTER MODULE

- The Router is a separate module in Angular. It is in its own library package, *@angular/router*. The Router Module provides the necessary service providers and directives for navigating through application views.
- We can use the following command to generate a basic app routing module:
  - Ng generate module {{name}} --routing
- Once we generate the module, we need to import it in the module which we would like to perform routing
- If we create a module during application creation, it will set up Routes array for your routes and configures the imports and exports array for @NgModule decorator

ROUTER MODULE

- We need to define our routes in Routes array
- Each routes in this array is a JavaScript object that contains two properties:
    - Path: defines the URL path for the route
    - Component, defines the component Angular should use for the corresponding path.
- Now we have defined the router, we need to add it in the application.
- RouterOutlet: this is a directive, it acts as a placeholder that Angular dynamically fills based on the current router state.
- RouterLink: this is also a directive, while we apply this to an element in our template, it will make this element a link that initiates navigation to a route.
- **EXAMPLE**

WILD CARD

- A well functioned application should also handle the situation when user attempts to navigate to an un-exist URL.

- To add this functionality, we can setup a **wildcard** route. Angular will select this route any time the requested URL does not match any other outer path.

- **EXAMPLE**

REDIRECT

- We can also use redirect to navigate user back into our router defined URL.

- Angular routing also support redirect, we can configure the route by providing a **redirectTo** property

- When reaching the URL, angular will redirect to the pre-defined URL instead

- **EXAMPLE**

NESTED ROUTER

- As our application grows more complex, we may want to create routes that are relative to a component other than the root component
- These are called nested routes, also called child routes.
- Which also means we will add a second <router-outlet> to our application.
- To implement nested routes, we need to define the child property
- **EXAMPLE**

ROUTER ORDER

- The order of routes is important because the Router uses a first-match wins strategy when matching routes.

- We should place more specific routes above less specific routes.

- And we should always put the wild card route as the last one otherwise all the pages will match the wild card.

- **EXAMPLE**

ROUTE NAVIGATE

- Router link provides us a way to navigate in template, what if we need to navigate inside the component?
- To do that we need to use Route and navigate method
- **EXAMPLE**

URL INFORMATION

- When talking about URL, two things will always be mentioned:
  - Path Variable
  - Query Params
- We can also implement path variable and query params using routing

QUERY PARAMETER

- To use query parameter together with router link, we need to setup another property called [queryParams], and pass an object with key-value pairs that defines the parameters.

- In router.navigate, we can pass the second argument which contains queryParams

- **EXAMPLE**

PATH VARIABLE

- To setup path variable, we use the second form of [routerLink]. When assign value, we also give it a second data which will be used as path variable

- **EXAMPLE**

HOW TO GET DATA

- Now we can pass the information using query parameter and path variable, but how can we get these information from URL?

- To get these information we need to use ActivatedRoute **service**.

ACTIVEROUTE

- To use ActivatedRoute, we need to import it in our component

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

- Then inject it into the component using dependency injection

```
constructor(private _Activatedroute:ActivatedRoute)
```

- There are two ways in which you can use the ActivatedRoute to get the parameter value
  - Using Snapshot
  - Using observable

ACTIVEROUTE

- The ActivatedRoute service has a great deal of useful information including:
  - url — This property returns an array of Url Segment objects, each of which describes a single segment in the URL that matched the current route.
  - params — This property returns a Params object, which describes the URL parameters, indexed by name.
  - queryParams — This property returns a Params object, which describes the URL query parameters, indexed by name.
  - snapshot — The initial snapshot of this route.
  - data — An Observable that contains the data object provided for the route
  - component — The component of the route. It's a constant.
  - routeConfig — The route configuration used for the route that contains the origin path.
  - parent — An ActivatedRoute that contains the information from the parent route when using child routes.
  - firstChild — contains the first ActivatedRoute in the list of child routes.
  - children — contains all the child routes activated under the current route

ACTIVEROUTE

- Why observable
  - We usually retrieve the value of the parameter in the ngOninit life cycle hook, when the component initialized.
  - When the user navigates to the component again, the Angular does not create the new component but reuses the existing instance.
  - In such circumstances, the ngOnInit method of the component is not called again. Hence we need a way to get the value of the parameter.
  - By subscribing to the observable params property, we will retrieve the latest value of the parameter and update the component accordingly.

STATIC DATA

```
{path: 'product/:id', component: ProductDetailComponentParam ,
  data:[{isProd: true}]}
```

- Parent components will usually pass data to their children, but Angular also offers a mechanism to pass arbitrary data to components at the time of route configuration.

- For example, besides dynamic data like a product ID, you may need to pass a flag indicating whether the application is running in a production environment. This can be done by using the data property of your route configuration.

```
constructor(route: ActivatedRoute) {
    this.productID = route.snapshot.params['id'];

    this.isProdEnvironment = route.snapshot.data[0]['isProd'];
    console.log("this.isProdEnvironment = " + this.isProdEnvironment);
  }
```

GUARD ROUTES

- To control whether the user can navigate to or away from a given route, use route guards.
  - For example, we may want some routes to only be accessible once the user has logged in or accepted Terms & Conditions. We can use route guards to check these conditions and control access to routes.
    - Log in check
- Route guards can also control whether a user can leave a certain route.
  - For example, say the user has typed information into a form on the page, but has not submitted the form. If they were to leave the page, they would lose the information.
  - Another example is the back button in the browser: if user has already submitted the quiz, then we should prevent him from using the back button to go back to the review page again.

GUARD ROUTES

- Uses of Guards
  - To Confirm navigational operation
  - Asking whether to save before moving away from a view
  - Allow access to certain parts of the application to specific users
  - Validating the route parameters before navigating to the route
  - Fetching some data before you display the component.
- Route Guards
  - The Angular Router supports Five different guards, which you can use to protect the route
    - CanActivate
    - CanDeactivate
    - Resolve
    - CanLoad
    - CanActivateChild

GUARD ROUTES

- CanActivate
  - This guard decides if a route can be activated (or component gets used). This guard is useful in the circumstance where the user is not authorized to navigate to the target component.
- CanDeactivate
  - This Guard decides if the user can leave the component (navigate away from the current route). This route is useful in where the user might have some pending changes, which was not saved.
- Resolve
  - This guard delays the activation of the route until some tasks are completed. You can use the guard to pre-fetch the data from the backend API, before activating the route
- CanLoad
  - This guard is used to guard the routes that load feature modules dynamically
- CanActivateChild
  - This guard determines whether a child route can be activated.

GUARD ROUTES

- Build the Guard as Service
- You need to import the corresponding guard from the Angular Router Library using the Import statement.
- For Example to use CanActivate Guard import the CanActivate in the import the CanActivate in the import statement

```
import { Injectable } from '@angular/core';
import { Router, CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router
```

- Next, create the Guard class which implement the selected guard Interface as shown below.

```
@Injectable()
export class AuthGuardService implements CanActivate {
```

GUARD ROUTES

- Implement the Guard Method in the Service
  - The next step is to create the Guard Method. The name of the Guard method is same as the Guard it implements. For Example to implement the CanActivate guard, create a method CanActivate.

```
canActivate(route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): boolean {

}
```

  - The guard method must return either a True or a False value.
  - if it returns true, the navigation process continues. if it returns false, the navigation process stops and the user stays put.

GUARD ROUTES

- Register the Guard Service in the Root Module
  - As mentioned earlier, guards are nothing but services. Hence they need to be registered with the Providers array of the Module as shown below

```
  ],
  providers: [AuthGuardService],
  bootstrap: [AppComponent]
})
```

  - Update the Routes to use the guards

```
{
  path: 'items',
  component: ItemsComponent,
  canActivate: [AuthGuardService]
},
```

- You can add more than one guard as shown below

```
{ path: 'product', component: ProductComponent,
      canActivate : any[],
      canActivateChild: any[],
      canDeactivate: any[],
      canLoad: any[],
      resolve: any[]
}
```

GUARD ROUTES

- A route can have multiple guards and you can have guards at every level of a routing hierarchy.
    - CanDeactivate() and CanActivateChild() guards are always checked first. The checking starts from the deepest child route to the top.
    - CanActivate() is checked next and checking starts from the top to the deepest child route.
    - CanLoad() is invoked next, If the feature module is to be loaded asynchronously
    - Resolve() is invoked last.
- If any one of the guards returns false, then the entire navigation is canceled.

MULTIPLE ROUTERS

- A component has one primary route and zero or more auxiliary routes.
- Auxiliary routes allow you to use and navigate multiple routes.
- To define an auxiliary route you need a named router outlet where the component of the auxiliary route will be rendered.
- We can have multiple outlets in the same template:

```
<router-outlet></router-outlet>
<router-outlet name="bottom"></router-outlet>
```

MULTIPLE ROUTERS

- Why multiple routers?
  - Let's think we want to add to an single page application a chat area so the user can communicate with a customer service representative while keeping the current route active as well.
- Basically we want to add an independent chat route allowing the user to use both routes at the same time and switch from one route to another

```
{ path: 'items', component: ItemsComponent },
{ path: 'items', component: ItemDetailComponent, outlet: "bottom" },
```

MULTIPLE MODULES

- Angular modules allow you to split an application into more than one module, where each module implements certain functionality
- As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features.
- With feature modules, you can keep code related to a specific functionality or feature separate from other code.
- AppModule, BrowserModule, and RouterModule.
  - AppModule is the root module of an application, but BrowserModule and RouterModule are feature modules.
  - Note the main difference between them: the root module is bootstrapped, whereas feature modules are imported.

MULTIPLE MODULES

- In a feature module, the @NgModule decorator has to import CommonModule instead of BrowserModule
- This is pretty easy refactoring, compared to the process of removing functionality from a monolithic single-module app.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CustomerDashboardModule { }
```

LAZY-LOADING MODULE

- In large applications, you want to minimize the amount of code that needs to be downloaded to render the landing page of your application. The less code your app initially downloads, the faster the user will see it.
- This is especially important for mobile apps when they're used in a poor connection area.
- If your application has modules that are rarely used, you can make them downloadable on demand, or lazy-loaded.
- Lazy loaded routes need to be outside of the root app module, so you'll want to have your lazy loaded features into feature modules.

LAZY-LOADING MODULE

- To implement Lazy loading into your application follow these steps
  - We use the property loadChildren instead of component.
  - We pass a string instead of a symbol to avoid loading the module eagerly.
  - We define not only the path to the feature module but the name of the class as well.
- When we load our application for the first time, the AppModule along the AppComponent will be loaded in the browser and we should see the navigation system and the text "Eager Component".
- Until this point, the LazyModule has not being downloaded, only when we click the link "Lazy" the needed code will be downloaded and we will see the message "Lazy Component" in the browser.

```
const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
  { path: 'eager', component: EagerComponent },
  { path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
];
```

PIPES

- A pipe is a template element that allows you to transform a value into a desired output. A pipe is specified by adding the vertical bar (|) and the pipe name right after the value to be transformed.

- Syntax of Pipe

```
Expression | pipeOperator[:pipeArguments]
```

- Expression : is the expression, which you want to transform
  - | : is the Pipe Character
  - pipeOperator : name of the Pipe
  - pipeArguments: arguments to the Pipe

BUILD-IN PIPES

- There are some built-in pipes provided by Angular
- DatePipe
  - The Date pipe formats the date according to locale rules. The syntax of the date pipe is as shown below

```
date_expression | date[:format]
```

```
{{toDate | date:'medium'}}
```

- date_expression is a date object or a number
- date is the name of the pipe
- format is the date and time format string which indicates the format in which date/time components are displayed.

BUILD-IN PIPES

- UpperCasePipe & LowerCasePipe
  - As the name suggests, these pipes transform the string to Uppercase or lowercase.
- DecimalPipe / NumberPipe
  - The Decimal Pipe is used to Format a number as Text. This pipe will format the number according to locale rules.

CUSTOM PIPE

- It's easy to create custom pipes to use in your templates to modify interpolated values
- It Import Pipe and PipeTransform and Use @Pipe decorator.

```
import {Pipe, PipeTransform} from '@angular/core';
```

- In your customPipe class implement PipeTransform interface.
- The PipeTransform interface has only one method known as the transform. This interface takes the value being piped as the first argument. It takes the variable number of optional arguments of any type. It returns the final transformed data.

CUSTOM PIPE

- We decorate the class with a @pipe decorator.
  - @pipe decorator is what tells Angular 2 that the class is a Pipe.

```
@pipe({
    name: 'tempConverter'
})
```

- Before using our Angular 2 custom pipe, we need to tell our component, where to find it. This done by first by importing it and then including it in declarations array of the AppModule.

```
import {TempConverterPipe} from './temp-convertor.pipe';

@NgModule({
    declarations: [AppComponent,TempConverterPipe],
    imports: [BrowserModule,FormsModule,HttpModule],
    bootstrap: [AppComponent]
})
export class AppModule { }
```