**Section 11: Changing Pages with Routing**

**Section 11: Lecture 112//Module Introduction**

1. We already learnt a lot about angular, this far we really built a single page application. We changed directives and so on to change the potions of this page – but in the end we always were on the same page. If we look at the URL on the top it was always localhost: 4200.
2. Now, what if we have page where we want to display several pages – like e have /users or we have /accounts page for this we indeed need multiple pages i.e. multiple index HTML files – but really we don’t need because angular ships with its own router which allows you to change the URL – in the URL bar and still only use one page but then exchange major parts or lot of parts of that page.
3. To the user it looks like a new page was loaded may because only the headed is the same. But behind the scenes it is still JavaScript changing lot of parts in your DOM/in your page, making it look like new page was rendered but you are still in the angular world i.e. in your single page application
4. Let’s take a closer look in this module.

**Section 11: Lecture 113// Why do we need a Router?**

1. Attached to the section we have an example project. We can notice in that project that we cannot switch the pages.
2. To work in this module, we need angular router. We must know which routs our project application has.
3. We want to split all the given tabs in the application into multiple pages i.e. multiple page looking components as angular creates a single page application.
4. Here we have users, servers, and Home on the same page – so that is something we can improve. It would be nice to have separate URLs for these 3 tabs and for this we need angular router.
5. Angular router knows which route our front end application has. Notice pattern from a full stack application where you also register some routes. Now, we can do the same thing in angular.

**Section 11: Lecture 114//Understanding the example project**

1. In our app, we got three sections:

* Home
* Servers
  + View and Edit Servers
  + A Service is used to load and update Servers
* Users
  + View Users

This app will be improved by adding routing but feel free to play around with it - besides routing, everything should be working fine.

**Section 11: Lecture 115//Setting up and loading routes**

1. So, here I am in the source code responsible for the app we saw in the last lecture. Now we can see that we have the 3 major components and then some sub-components of these components.
2. In the app.component.html I am loading them all i.e. my home component, server’s component, and users’ component. For now, we will ignore the sub components.
3. Now, where do we register such routes? – these routes are responsible for our whole app.
4. Now, if we enter /users in the URL then we want to load the users’ component as our main pagelet. Now, this is the core part of our app, so, where should we register it? - well, app.module.ts sounds like a good place, because here is where we configure our app. Here only we add our components and so on.
5. Now, here we will add a new constant called appRoutes; now, this constant should be of a type the routes type which must be imported from @angular/router. This constant should hold an array because our application has multiple routes.
6. We add all the routes and want to setup this array for now. Each route for now is just a JavaScript object in this array.
7. Now, the question is that how does such a rout should be configured in an angular app? - It must follow a pattern and a structure for angular to be able to use it. And this structure always needs a path.
8. Just give the name of the path in the object in the constant and define what should happen when this path is reached. The action typically is the component. Now, we will inform angular here that when the certain path here is reached then this component should be reached. And this component would then be the page which gets loaded.
9. We must make sure we are configuring the components in our app such that they look like pages.
10. Now, just by adding that constant angular won’t know what to do. How would angular know that you want to use this constant? – so, we somehow must register these routes in our app and we do this by adding a new import here i.e. we must add the router module.
11. Using RouteModule.forRoot(appRoutes) we would be able to register our routing functionality in our app. And now angular knows our routes.
12. The missing piece is to render the currently selected component.
13. app.module.ts
14. import { BrowserModule } from '@angular/platform-browser';
15. import { NgModule } from '@angular/core';
16. import { FormsModule } from '@angular/forms';
17. import { HttpModule } from '@angular/http';
18. import { AppComponent } from './app.component';
19. import { HomeComponent } from './home/home.component';
20. import { UsersComponent } from './users/users.component';
21. import { ServersComponent } from './servers/servers.component';
22. import { UserComponent } from './users/user/user.component';
23. import { EditServerComponent } from './servers/edit-server/edit-server.component';
24. import { ServerComponent } from './servers/server/server.component';
25. import { ServersService } from './servers/servers.service';
26. import { Routes, RouterModule } from '@angular/router';
28. const appRoutes: Routes = [
29. { path: '', component: HomeComponent },
30. { path: 'users', component: UserComponent },
31. { path: 'servers', component: ServerComponent }
32. ];
33. @NgModule({
34. declarations: [
35. AppComponent,
36. HomeComponent,
37. UsersComponent,
38. ServersComponent,
39. UserComponent,
40. EditServerComponent,
41. ServerComponent
42. ],
43. imports: [
44. BrowserModule,
45. FormsModule,
46. HttpModule,
47. RouterModule.forRoot(appRoutes)
48. ],
49. providers: [ServersService],
50. bootstrap: [AppComponent]
51. })
52. export class AppModule { }

14. app.component.html:

<div class="container">

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<ul class="nav nav-tabs">

<li role="presentation" class="active"><a href="#">Home</a></li>

<li role="presentation"><a href="#">Servers</a></li>

<li role="presentation"><a href="#">Users</a></li>

</ul>

</div>

</div>

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<router-outlet></router-outlet>

</div>

</div>

</div>

**Section 11: Lecture 116// Navigating with Router Links**

1. As we can see that right now, we can change pages or make routing work by typing the URLs not by just clicking the buttons or menu items. It would also be nice to make some links work in our application here.
2. So, lets add some links to our application. Now, when we give link paths for routing in the app.component.html we see that now when we click on the menu item buttons the page reloads.
3. Every link we click the page reloads. This however is not the best behavior because restarts our app on every click to the button. That means our whole application state will be lost. It might not be the behavior we want to offer to the user.
4. So, this not how we should implement navigation. How should we implement then? - there is a special directive angular gives us i.e. routerLink.
5. Another way of using routerLink is by using Property Binding.
6. app.component.html:
7. <div class="container">
8. <div class="row">
9. <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
10. <ul class="nav nav-tabs">
11. <li role="presentation" class="active"><a routerLink="/">Home</a></li>
12. <li role="presentation"><a routerLink="/servers">Servers</a></li>
13. <li role="presentation"><a [routerLink]="['/users']">Users</a></li>
14. </ul>
15. </div>
16. </div>
17. <div class="row">
18. <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
19. <router-outlet></router-outlet>
20. </div>
21. </div>
22. </div>

**Section 11: Lecture 117//Understanding Navigation Paths**

1. In the last lecture we added this navigation, we also mentioned that we can change the way in which we use the paths here.
2. Now, in the servers.component.html we added this router link which was earlier only present at app.component.html.
3. servers.component.html:
4. <div class="row">
5. <div class="col-xs-12 col-sm-4">
6. <div class="list-group">
7. <a
8. href="#"
9. class="list-group-item"
10. \*ngFor="let server of servers">
11. {{ server.name }}
12. </a>
13. </div>
14. </div>
15. <div class="col-xs-12 col-sm-4">
16. <a routerLink="../server">Reload Page</a>
17. <app-edit-server></app-edit-server>
18. <hr>
19. <app-server></app-server>
20. </div>
21. </div>

**Section 11: Lecture 118//Styling active router links**

1. One thing is that we are not getting any indication of what the currently active route is, this is purely CSS thing.
2. Now, home page here is set to active by default. But we can set it dynamically to know which route is currently executing. Angular gives us a directive for this i.e. routerLinkActive directive.

**Section 11: Lecture 119//Navigating Programmatically**

1. Till now we have learnt how to add routs and how to load them. Either by navigation bar or by clicking some links. What now if we want to load a route programmatically.
2. Say, we finished some operation or the user clicked some button or the user clicked some button and then we want to trigger the navigation from our typescript code.
3. Let’s say in our home component here we add a new button to it and on this button I simply want to load the server.
4. Now, when the button is clicked and the click method is called we need to tell this to angular router that please navigate somewhere else.
5. [home.component.html](http://home.component.html):
6. <h4>Welcome to Server Manager 4.0</h4>
7. <p>Manage your Servers and Users.</p>
8. <button class ="btn btn-primay" (click) = "onLoadServers()">Load Servers</button>
10. [Home.component.ts](http://Home.component.ts):
11. import { Component, OnInit } from '@angular/core';
12. import { Router } from '@angular/router';
13. @Component({
14. selector: 'app-home',
15. templateUrl: './home.component.html',
16. styleUrls: ['./home.component.css']
17. })
18. export class HomeComponent implements OnInit {
19. constructor(private router: Router) { }
20. ngOnInit() {
21. }
22. onLoadServers(){
23. //complex calculation
24. this.router.navigate(['/servers']);
25. }
26. }

**Section 11: Lecture 120 //Using Relative Paths in Programmatic navigation**

1. In the last lecture we learnt how to navigate programmatically, now, we will look at how to use relative paths there.
2. Now, let’s say in our servers.component.htm, we add a button for the reload of the page. We will also inject the router in the services.component.ts
3. Let’s have the reload button this time instead of the reload link as we did in the last lecture. Here I will have my click listener on the reload page maybe.
4. Now, we are already injecting the service in the servers.component.ts, we will also import router in the similar way. So, here we added the router in the Constructor.
5. When we click this reload page then the request will never be sent to server and the page will not show any signs of being reloaded but internally it still does reload.
6. Unlike the router link the navigate method does not know on which route you are currently on.
7. The router link always knows in which component it sits and in which component is the template is, and therefore it knows where the currently loaded route is.
8. Now, to tell the navigate method where we are at present in the routing i.e. second perimeter in the navigate method which would be a JavaScript object. We will come to this later as there are more things we can add. One configuration is **relativeTo** property – meaning- relative to which route this link will be loaded, by default this should always be the link to main. Here we `must give a route though, we don’t have to give a string here.
9. The route which we intend to add in the navigate method can be also injected in the constructor, which is of the type ActivatedRoute.
10. ActivatedRoute injects the currently active routes for the component we loaded. Route is a complex object which keeps a lot of meta information about the currently active route. Now we can set this value for this route for the relative to property.
11. With this extra piece of information angular knows what the currently active route is.
12. So, here we learnt how to use the relative paths in the navigate method. With the second perameter we would be able to define the relative path to which we want to navigate the application.
13. servers.component.ts:
14. import { Component, OnInit } from '@angular/core';
15. import { ServersService } from './servers.service';
16. import { Router, ActivatedRoute } from '@angular/router';
17. @Component({
18. selector: 'app-servers',
19. templateUrl: './servers.component.html',
20. styleUrls: ['./servers.component.css']
21. })
22. export class ServersComponent implements OnInit {
23. private servers: {id: number, name: string, status: string}[] = [];
24. constructor(private serversService: ServersService, private router: Router, private route: ActivatedRoute) { }
25. ngOnInit() {
26. this.servers = this.serversService.getServers();
27. }
28. onReload(){
29. this.router.navigate(['servers'], {relativeTo:this.route});
30. }
31. }

**Section 11: Lecture 121//Passing Parameters to Routes**

1. We are now adding some other routes to app.module.ts; now; now let’s say besides our users route here we should be able to load the single. Now we would need to give path for the user inside the constant we declared inside the app.module.ts.
2. Now, here we will provide path as

{ path: 'users/:id', component: UserComponent },

1. Colon simply tells angular that this the dynamic part of the path
2. app.module.ts:
3. import { BrowserModule } from '@angular/platform-browser';
4. import { NgModule } from '@angular/core';
5. import { FormsModule } from '@angular/forms';
6. import { HttpModule } from '@angular/http';
7. import { AppComponent } from './app.component';
8. import { HomeComponent } from './home/home.component';
9. import { ServersComponent } from './servers/servers.component';
10. import { UserComponent } from './users/user/user.component';
11. import { UsersComponent } from './users/users.component';
12. import { EditServerComponent } from './servers/edit-server/edit-server.component';
13. import { ServerComponent } from './servers/server/server.component';
14. import { ServersService } from './servers/servers.service';
15. import { Routes, RouterModule } from '@angular/router';
17. const appRoutes: Routes = [
18. { path: '', component: HomeComponent },
19. { path: 'users', component: UsersComponent },
20. { path: 'users/:id', component: UserComponent },
21. { path: 'servers', component: ServerComponent }
22. ];
23. @NgModule({
24. declarations: [
25. AppComponent,
26. HomeComponent,
27. UsersComponent,
28. ServersComponent,
29. UserComponent,
30. EditServerComponent,
31. ServerComponent
32. ],
33. imports: [
34. BrowserModule,
35. FormsModule,
36. HttpModule,
37. RouterModule.forRoot(appRoutes)
38. ],
39. providers: [ServersService],
40. bootstrap: [AppComponent]
41. })
42. export class AppModule { }

**Section 11: Lecture 122//Fetching Route Parameters**

1. In the last lecture we created our route with dynamic path segment, now we want to have access of the data which user sent or which is encoded in the URL I say.
2. So, we noted that we will load the user component here and we know that there will be some data in the URL as well. How could we get access to it - now it is the typescript file in which we can get access of it.
3. Now, we need to inject the same things through the constructor that we injected earlier i.e. the Active route i.e. ActivatedRoute.
4. Now, we know that that the currently loaded route is the JavaScript object with the metadata about the currently loaded route. One of the important information is the currently active user.
5. We can see in the app.component.user file that we have defined a user in the file and for now it should have the following structure and it’s not used right now.
6. Now, after getting the parameters from the URL, we can output those using the string interpolation
7. app.module.ts:
8. import { BrowserModule } from '@angular/platform-browser';
9. import { NgModule } from '@angular/core';
10. import { FormsModule } from '@angular/forms';
11. import { HttpModule } from '@angular/http';
12. import { AppComponent } from './app.component';
13. import { HomeComponent } from './home/home.component';
14. import { ServersComponent } from './servers/servers.component';
15. import { UserComponent } from './users/user/user.component';
16. import { UsersComponent } from './users/users.component';
17. import { EditServerComponent } from './servers/edit-server/edit-server.component';
18. import { ServerComponent } from './servers/server/server.component';
19. import { ServersService } from './servers/servers.service';
20. import { Routes, RouterModule } from '@angular/router';
22. const appRoutes: Routes = [
23. { path: '', component: HomeComponent },
24. { path: 'users', component: UsersComponent },
25. { path: 'users/:id/:name', component: UserComponent },
26. { path: 'servers', component: ServerComponent }
27. ];
28. @NgModule({
29. declarations: [
30. AppComponent,
31. HomeComponent,
32. UsersComponent,
33. ServersComponent,
34. UserComponent,
35. EditServerComponent,
36. ServerComponent
37. ],
38. imports: [
39. BrowserModule,
40. FormsModule,
41. HttpModule,
42. RouterModule.forRoot(appRoutes)
43. ],
44. providers: [ServersService],
45. bootstrap: [AppComponent]
46. })
47. export class AppModule { }

8. user.component.ts:

import { Component, OnInit } from '@angular/core';

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

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css']

})

export class UserComponent implements OnInit {

user: {id: number, name: string};

constructor(private route: ActivatedRoute) { }

ngOnInit() {

this.user = { id: this.route.snapshot.params['id'],

name:this.route.snapshot.params['name'] };

}

}

1. user.component.html:
2. <p>User with {{ user.id }}loaded.</p>
3. <p>User name is {{ user.name }}</p>

**Section 11: Lecture 123//Fetching Route Parameters Reactively**

1. Now, in our user component we saw that we passed the user ID and URL, lets quickly add a router link in the user.component.html.
2. Now we will add an array in the router link as shown below which will act as /user/10/Anna
3. Once we click load Anna here; routing will take us to this link - /user/10/Anna; here we load our data using the snapshot object of the route.
4. Now, if we load a new route what will happen - Angular will look into our app.module.ts – finds the fitting route here – loads the component – initializes the component and gives us the data by accessing the snapshot here.
5. This all only happens if we haven’t been on this component before but if we click this link which is on the user component 🡪 the URL still changes but we are on the component already. Here angular cleverly doesn’t instantiate this component – that would only cost the performance by re-rendering the component we already are on.
6. By default, angular will not destroy the current component and re-create the same component again if we are on that component already.
7. Its fine to use the snapshot for the first initialization but to be able to react to the subsequent changes we need a different approach. In our user.component.ts
8. Now, we can use our route object i.e. the params property of the route object itself, we didn’t use that before we have the snapshot in between.
9. Params here is an observable; observable is something on which we will come to right after this section.
10. Basically, **observables** are the features added by a third party package and not by angular but its heavily used by angular which allows you to work easily with **Asynchronous tasks**.
11. Observable is used to subscribe some event which might happen in future to then execute this code when it happens.
12. Now, as the name describes observable is something which we can observe. So, we will call subscribe method on it and this is still called on params. Whenever the parameters change in this case the observable will be executed.
13. Now, we must pass arguments in the subscribe method – subscribe can take 3 functions as arguments – the 1st one is the most important here, this would be fired when new data will be passed through the observable. To put in the easier words whenever the parameters change in this use case then this first argument which should be a function here is executed.
14. We will use the ES6 arrow function which will take prams as an argument and we can set this as the type param.
15. Now, inside the function, which is the first parameter in the subscribe function we can fetch the user object and pass id from the params.
16. So, this will now update our user object whenever the parameter changes.
17. This code will not be executed when ngOnInit runs through and the subscription will be set up; now only if then parameters change then the inside part of the subscribe function will run.
18. If you know that the component you are working on might not reloaded from within the component, then we might not need this.
19. user.component.ts:
20. import { Component, OnInit } from '@angular/core';
21. import { ActivatedRoute, Params } from '@angular/router';
22. import { paramKey } from 'blocking-proxy/built/lib/webdriver\_commands';
23. @Component({
24. selector: 'app-user',
25. templateUrl: './user.component.html',
26. styleUrls: ['./user.component.css']
27. })
28. export class UserComponent implements OnInit {
29. user: {id: number, name: string};
30. constructor(private route: ActivatedRoute) { }
31. ngOnInit() {
32. this.user = { id: this.route.snapshot.params['id'],
33. name:this.route.snapshot.params['name'] };
34. this.route.params
35. .subscribe(
36. (params: Params) => {
37. this.user.id = params['id'];
38. this.user.name = params['name'];
39. }
40. );
41. }
42. }

20. user.component.html:

] <p>User with {{ user.id }}loaded.</p>

<p>User name is {{ user.name }}</p>

<hr>

<a [routerLink]="['/users', 10, 'Anna']">Load Anna (10)</a>

**Section 11: Lecture 124 //An important Note about Route Observables**

1. In the last lecture we learnt that how we can subscribe to our params to update them or to react to any changes and update our page if we already are on that page then we set up this subscription using observables and you don’t have to change anything here.
2. Here something is there which needs our attention though, the fact that we don’t need to add anything to this component is that angular does something for us here in the background which is super important for us. It cleans up the subscription that we created whenever this component is destroyed because if wouldn’t do it what are we doing here - we are subscribing to the parameter changes and then leave this component and later we come back. So, once we come back to this component then the same component will be created again and the subscription will remain in some memory. As, the component is destroyed the subscription won’t.
3. It will be there as the angular handles the destroying of the subscription for us but theoretically we want to implement on destroy lifecycle hook from @Core
4. And then we can store this subscription in some property. rxjs is the package offering all this observables functionality and as we missioned it is not shipping with angular. Now we can bound the subscription with this property and later when the component is destroyed then we can unsubscribe the subscription using that property.
5. It’s important to know that we don’t have to do this and angular will do it for us, but even if we does this manually then also it will not heart and will not add anything bad to our app.
6. User.component.ts
7. import { Component, OnInit, OnDestroy } from '@angular/core';
8. import { ActivatedRoute, Params } from '@angular/router';
9. import { paramKey } from 'blocking-proxy/built/lib/webdriver\_commands';
10. import { Subscription } from 'rxjs/Subscription';
11. @Component({
12. selector: 'app-user',
13. templateUrl: './user.component.html',
14. styleUrls: ['./user.component.css']
15. })
16. export class UserComponent implements OnInit, OnDestroy {
17. user: {id: number, name: string};
18. paramsSubsription: Subscription;
19. constructor(private route: ActivatedRoute) { }
20. ngOnInit() {
21. this.user = { id: this.route.snapshot.params['id'],
22. name:this.route.snapshot.params['name'] };
23. this.paramsSubsription = this.route.params
24. .subscribe(
25. (params: Params) => {
26. this.user.id = params['id'];
27. this.user.name = params['name'];
28. }
29. );
30. }
31. ngOnDestroy(){
32. this.paramsSubsription.unsubscribe();
33. }
34. }

**Section 11: Lecture 125//Passing Query Parameters and Fragments**

1. In the last lecture we learnt how we can retrieve our route params that awesome and this is super important, we will probably use it a lot.
2. Now, there are more things that we can add to our url; you might have query parameters separated by a question mark

Ex: localhost:4200/users/10/Anna?mode=editing

1. We can have multiple query parameters separated by the add i.e. & signs, question is that ow you can pass them using angular links/router links and how can you then retrieve them. You might also have # fragments
2. We can us the # sign to jump to the places in our app or maybe you just want to use it to retrieve some extra information from the URL. Let’s have a closer look at how we can pass the extra information in our links both when using our router link method and navigate method, and how we can then retrieve the information.
3. Let’s start by passing it first; lets say in app.module.ts we want to add more routes; lets say that would be the route to allow us to add certain server.
4. 'servers/:id/edit'
5. Here we have added edit in the end to describe what will happen if we access this component.
6. To be able to load this route in my servers component; now you say we also want to have some query parameter which defines whether we are allow to edit the parameter or not so and we will not add the ? in the routerLink array. But we will bind this to the [queryParams] property.
7. queryParams is not a new directive but it is the bind able property of the routerLink directive and here we must pass the JavaScript object and here in the queryParams we can add the name value pairs in the object in this.
8. Like queryParams property we also have the fragment property. This is named fragment as we ca only have 1 fragment
9. Now, here we will make the id dynamic as we were hard codding it earlier.
10. Now, in the [home.component.ts](http://home.component.ts) we can add parameter 2 which would be an object in the navigate method i.e. the queryParams
11. Like the queryParams we can also add the third parameter that would be the fragment
12. [Home.component.ts](http://Home.component.ts):
13. import { Component, OnInit } from '@angular/core';
14. import { Router } from '@angular/router';
15. @Component({
16. selector: 'app-home',
17. templateUrl: './home.component.html',
18. styleUrls: ['./home.component.css']
19. })
20. export class HomeComponent implements OnInit {
21. constructor(private router: Router) { }
22. ngOnInit() {
23. }
24. onLoadServers(id: number){
25. //complex calculation
26. this.router.navigate(['/servers',id, 'edit'], {queryParams: {allowEdit: '1'}, fragment:'loading'});
27. }
28. }
29. servers.component.html:
30. <div class="row">
31. <div class="col-xs-12 col-sm-4">
32. <div class="list-group">
33. <a
34. [routerLink]="['/server', 5, 'edit']"
35. [queryParams]="{allowEdit: '1'}"
36. fragment="loading"
37. href="#"
38. class="list-group-item"
39. \*ngFor="let server of servers">
40. {{ server.name }}
41. </a>
42. </div>
43. </div>
44. <div class="col-xs-12 col-sm-4">
45. <!-- <a routerLink="../server">Reload Page</a> -->
46. <button class="btn btn-primary" (click)="onReload()">Reload Page</button>
47. <app-edit-server></app-edit-server>
48. <hr>
49. <app-server></app-server>
50. </div>
51. </div>

**Section 11: Lecture 126 //Retrieving Query Parameters and Fragments**

1. In the last lecture we learnt how to pass the query parameters and the fragments, now let’s make sure that we can also retrieve them.
2. So, we are loading this edit server component in the end that is probably where we want to retrieve that information. We can do this very easily by going there i.e. esdit-server.component.ts. Till now everything in this file is done by our serversService – this service is getting the server and updating it in this component.
3. The interesting part in this edit-server.component.ts file is how will access the query parameter and the fragment - we need our activated route for this, so let’s inject it just like we did it before.
4. Now, we can retrieve it in our ngOnInit() method - now, we have 2 ways to retrieve the parameter and the fragment i.e.
5. Getting it by using the snapshot object and access queryParams here or our fragment.
6. The second method is – use the route and just like we have params as observable, we also have queryParams also as observable – to which we can subscribe like shown in the code and we have fragment as observable we can subscribe to.

So, remember we don’t need to unsubscribe here manually because angular here will do it for us, but this is how you get access to these extra features and then you can make sure you don’t miss the data.

1. edit-server.component.ts:
2. import { Component, OnInit } from '@angular/core';
3. import { ServersService } from '../servers.service';
4. import { ActivatedRoute } from '@angular/router';
5. @Component({
6. selector: 'app-edit-server',
7. templateUrl: './edit-server.component.html',
8. styleUrls: ['./edit-server.component.css']
9. })
10. export class EditServerComponent implements OnInit {
11. server: {id: number, name: string, status: string};
12. serverName = '';
13. serverStatus = '';
14. constructor(private serversService: ServersService, private route: ActivatedRoute) { }
15. ngOnInit() {
16. console.log(this.route.snapshot.queryParams);
17. console.log(this.route.snapshot.fragment);
18. this.route.queryParams.subscribe();
19. this.route.fragment.subscribe();
20. this.server = this.serversService.getServer(1);
21. this.serverName = this.server.name;
22. this.serverStatus = this.server.status;
23. }
24. onUpdateServer() {
25. this.serversService.updateServer(this.server.id, {name: this.serverName, status: this.serverStatus});
26. }
27. }

s

**Section 11: Lecture 127//Practicing some common Gotchas**

1. So, now we have learnt a lot about navigating and passing the parameters, lets improve our app a little bit by going to users.component.html where we have list in the end to single users and lets ass the routerLink here, which has an array as an argument. Here we want to target [routerLink]=”[‘/users’, user.id, user.name]” - this is how we will dynamically create such kind of links.
2. We saw this with the navigate method a couple of lectures earlier, let’s do the same thing in the servers.
3. For edit we need to add another route- so in the app.module.ts we will add another route in the constant. Now we will inject the Activated route in the server.component.ts in the constructor.
4. Now, in the ngOninit we must get this activated route from the route.snapshot object.
5. Now, if we want to react to any changes thereafter, we need to subscribe to our params observable here. By doing this we can get a new server here whenever our params should change.
6. On our servers component we are still loading our single servers component, there for we are loading it even if we don’t have the id available, now we will comment it, but soon we will learn how we can nest another router in there – we have some kind of child routing.
7. The id that we have in our URL is the string ID, so the server will be searched with the ID which is string - however, we have defined IDs as numbers in our app. To convert the id returned inside the observable into number we just need to add + in front of the line - so, here we converted the params to the number too.
8. server.component.ts:
9. import { Component, OnInit } from '@angular/core';
10. import { ServersService } from '../servers.service';
11. import { ActivatedRoute, Params } from '@angular/router';
12. @Component({
13. selector: 'app-server',
14. templateUrl: './server.component.html',
15. styleUrls: ['./server.component.css']
16. })
17. export class ServerComponent implements OnInit {
18. server: {id: number, name: string, status: string};
19. constructor(private serversService: ServersService, private route: ActivatedRoute) { }
20. ngOnInit() {
21. const id = +this.route.snapshot.params['id'];
22. this.server = this.serversService.getServer(1);
23. this.route.params
24. .subscribe((params: Params)=>{
25. this.server = this.serversService.getServer(+params['id']);
26. });
27. }
28. }

9. servers.component.ts:

import { Component, OnInit } from '@angular/core';

import { ServersService } from './servers.service';

import { Router, ActivatedRoute } from '@angular/router';

@Component({

selector: 'app-servers',

templateUrl: './servers.component.html',

styleUrls: ['./servers.component.css']

})

export class ServersComponent implements OnInit {

private servers: {id: number, name: string, status: string}[] = [];

constructor(private serversService: ServersService, private router: Router, private route: ActivatedRoute) { }

ngOnInit() {

//this.servers = this.serversService.getServers();

}

onReload(){

this.router.navigate(['servers'], {relativeTo:this.route});

}

}

10. app.module.ts:

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

import { HomeComponent } from './home/home.component';

import { ServersComponent } from './servers/servers.component';

import { UserComponent } from './users/user/user.component';

import { UsersComponent } from './users/users.component';

import { EditServerComponent } from './servers/edit-server/edit-server.component';

import { ServerComponent } from './servers/server/server.component';

import { ServersService } from './servers/servers.service';

import { Routes, RouterModule } from '@angular/router';

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent },

{ path: 'users/:id/:name', component: UserComponent },

{ path: 'servers', component: ServerComponent },

{ path: 'servers/:id', component: ServerComponent },

{ path: 'servers/:id/edit', component: EditServerComponent }

];

@NgModule({

declarations: [

AppComponent,

HomeComponent,

UsersComponent,

ServersComponent,

UserComponent,

EditServerComponent,

ServerComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule,

RouterModule.forRoot(appRoutes)

],

providers: [ServersService],

bootstrap: [AppComponent]

})

export class AppModule { }

11. servers.component.html

<div class="row">

<div class="col-xs-12 col-sm-4">

<div class="list-group">

<a

[routerLink]="['/server', server.id, 'edit']"

[queryParams]="{allowEdit: '1'}"

fragment="loading"

href="#"

class="list-group-item"

\*ngFor="let server of servers">

{{ server.name }}

</a>

</div>

</div>

<div class="col-xs-12 col-sm-4">

<!-- <a routerLink="../server">Reload Page</a> -->

<button class="btn btn-primary" (click)="onReload()">Reload Page</button>

<app-edit-server></app-edit-server>

<hr>

<!-- <app-server></app-server> -->

</div>

</div>

**Section 11: Lecture 128// Setting up Child (Nested) Routes**

1. Here issue is that when we click user/server/home we load a brand new page, so, we need some nested routing.
2. In the app module also we see there is some duplication in the URLs; half of our routes starts with servers and rest half start with users in constant where we have given path – this results into duplication.
3. So, here we need to nest them and create some child routes which start with servers/users – lets add such child routes.
4. Now, we will go to ServersComponent route and add another property in the object i.e. children; children take another array of routes. So, I can take my to other servers routes and add them in the children array. Now, we need to get rid of the servers in the starting of the path for the other 2 servers routes.
5. This now, already better because we have grouped the parent and children routes together. It gives us more than this visual grouping though.
6. Now the question is that we have ServersComponent in the parent route then where will the serverComponent and EditServerComponent load?
7. The child route needs a separate outlet because they cannot override the servers component, but they needs to be loaded nested into this servers component – and that is the actual behavior we want.
8. Now, quickly in the servers.component.html where we load the app-server page or app-edit-server page- we will simply comment all that code and instead add a router outlet here.
9. <router-outlet></ router-outlet > //this now adds a new hook which will be used for all the child routes of the route being loaded on the servers component. This is the way we can add the child routes now, let’s make the changes for the user routes now.
10. With <router-outlet></ router-outlet >, all the server and the user related child routes will be loaded.
11. This is how we implement the child routing and the nested routes.
12. app.module.ts:

app.module.ts  
import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

import { HomeComponent } from './home/home.component';

import { ServersComponent } from './servers/servers.component';

import { UserComponent } from './users/user/user.component';

import { UsersComponent } from './users/users.component';

import { EditServerComponent } from './servers/edit-server/edit-server.component';

import { ServerComponent } from './servers/server/server.component';

import { ServersService } from './servers/servers.service';

import { Routes, RouterModule } from '@angular/router';

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent,children:[

{ path: ':id/:name', component: UserComponent }

] },

{ path: 'servers', component: ServerComponent, children:[

{ path: ':id', component: ServerComponent },

{ path: ':id/edit', component: EditServerComponent }

]},

];

@NgModule({

declarations: [

AppComponent,

HomeComponent,

UsersComponent,

ServersComponent,

UserComponent,

EditServerComponent,

ServerComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule,

RouterModule.forRoot(appRoutes)

],

providers: [ServersService],

bootstrap: [AppComponent]

})

export class AppModule { }

1. users.component.html:
2. <div class="row">
3. <div class="col-xs-12 col-sm-4">
4. <div class="list-group">
5. <a
6. href="#"
7. class="list-group-item"
8. \*ngFor="let user of users">
9. {{ user.name }}
10. </a>
11. </div>
12. </div>
13. <div class="col-xs-12 col-sm-4">
14. <!-- <app-user></app-user> -->
15. <router-outlet></router-outlet>
16. </div>
17. </div>

14. servers.component.html:

<div class="row">

<div class="col-xs-12 col-sm-4">

<div class="list-group">

<a

[routerLink]="['/server', server.id, 'edit']"

[queryParams]="{allowEdit: '1'}"

fragment="loading"

href="#"

class="list-group-item"

\*ngFor="let server of servers">

{{ server.name }}

</a>

</div>

</div>

<div class="col-xs-12 col-sm-4">

<!-- <a routerLink="../server">Reload Page</a> -->

<!-- <button class="btn btn-primary" (click)="onReload()">Reload Page</button>

<app-edit-server></app-edit-server>

<hr> -->

<!-- <app-server></app-server> -->

<router-outlet></router-outlet>

</div>

</div>

**Section 11: Lecture 129//Using query parameters – Practice**

1. In the last lecture we used the child routing, now let’s use this app a little more, in our servers.component.html we have our links loading the individual servers, so, the single server component with this allow edit query parameter.
2. Now, in the server component we should add some button - where we say edit server
3. Now, to get the access to the navigate method in a method, we need to inject the router in the constructor.
4. In the navigate method we will pass the array for the path. In this case we will only pass the edit in single quotes.
5. Now, we want to store the information that we have clicked and gone on certain URL, somewhere and we also want to keep track if we want to allow to edit or not - this we can keep track by adding another property called allowEdit - which we will keep false initially.
6. Now, this we will set in the subscribe method inside the observable i.e. the value of the allowEdit property.
7. We will learn how to preserve our queyParams as we navigate in the app in the next lecture.
8. edit-server.component.html:
9. <h4 \*ngIf = "!allowEdit">You're not allowed to edit!</h4>
10. <div \*ngIf = "allowEdit">
11. <div class="form-group">
12. <label for="name">Server Name</label>
13. <input
14. type="text"
15. id="name"
16. class="form-control"
17. [(ngModel)]="serverName">
18. </div>
19. <div class="form-group">
20. <label for="status">Server Status</label>
21. <select
22. id="status"
23. class="form-control"
24. [(ngModel)]="serverStatus">
25. <option value="online">Online</option>
26. <option value="offline">Offline</option>
27. </select>
28. </div>
29. <button
30. class="btn btn-primary"
31. (click)="onUpdateServer()">Update Server</button>
32. </div>

10. edit-server.component.ts:

import { Component, OnInit } from '@angular/core';

import { ServersService } from '../servers.service';

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

@Component({

selector: 'app-edit-server',

templateUrl: './edit-server.component.html',

styleUrls: ['./edit-server.component.css']

})

export class EditServerComponent implements OnInit {

server: {id: number, name: string, status: string};

serverName = '';

serverStatus = '';

allowEdit = false;

constructor(private serversService: ServersService, private route: ActivatedRoute) { }

ngOnInit() {

console.log(this.route.snapshot.queryParams);

console.log(this.route.snapshot.fragment);

this.route.queryParams

.subscribe(

(queryParams) => {

this.allowEdit = queryParams['allowEdit']=== '1' ? true : false;

}

);

this.route.fragment.subscribe();

this.server = this.serversService.getServer(1);

this.serverName = this.server.name;

this.serverStatus = this.server.status;

}

onUpdateServer() {

this.serversService.updateServer(this.server.id, {name: this.serverName, status: this.serverStatus});

}

}

11. server.component.ts:

import { Component, OnInit } from '@angular/core';

import { ServersService } from '../servers.service';

import { ActivatedRoute, Params, Router } from '@angular/router';

import { Route } from '@angular/compiler/src/core';

@Component({

selector: 'app-server',

templateUrl: './server.component.html',

styleUrls: ['./server.component.css']

})

export class ServerComponent implements OnInit {

server: {id: number, name: string, status: string};

constructor(private serversService: ServersService, private route: ActivatedRoute, private router:Router) { }

ngOnInit() {

const id = +this.route.snapshot.params['id'];

this.server = this.serversService.getServer(1);

this.route.params

.subscribe((params: Params)=>{

this.server = this.serversService.getServer(+params['id']);

});

}

onEdit(){

this.router.navigate(['edit'], {relativeTo: this.route });

}

}

**Section 11: Lecture 130 //Configuring the Handling of Query Parameters**

1. So, in the last lecture we tried to improve our application a bit more, the issue the query parameter is gone once we navigate away from our single server component – the edit-server component for example.
2. Bu, we want to preserve them because we set the information at the time we preserved the single server.
3. Now, we want to preserve this information once we navigate one set further to the edit-server component.
4. To, preserve such information we’ve got the simple way of doing so. In the server component when we navigate we can add another property to the JavaScript object which is the second parameter in the navigate method.
5. Here we will add queryParamshandlingProperty which takes the value as a string i.e. preserver here as we want to **preserve** the query params not drop them.
6. Now, if we override and add the new ones then we would have needed the value as **merge** for the queryParamshandlingProperty.
7. Server.component.ts:
8. import { Component, OnInit } from '@angular/core';
9. import { ServersService } from '../servers.service';
10. import { ActivatedRoute, Params, Router } from '@angular/router';
11. import { Route } from '@angular/compiler/src/core';
12. @Component({
13. selector: 'app-server',
14. templateUrl: './server.component.html',
15. styleUrls: ['./server.component.css']
16. })
17. export class ServerComponent implements OnInit {
18. server: {id: number, name: string, status: string};
19. constructor(private serversService: ServersService, private route: ActivatedRoute, private router:Router) { }
20. ngOnInit() {
21. const id = +this.route.snapshot.params['id'];
22. this.server = this.serversService.getServer(1);
23. this.route.params
24. .subscribe((params: Params)=>{
25. this.server = this.serversService.getServer(+params['id']);
26. });
27. }
28. onEdit(){
29. this.router.navigate(['edit'], {relativeTo: this.route, queryParamsHandling: 'preserve' });
30. }
31. }

**Section 11: Lecture 131//Redirecting and Wildcard Routes**

1. We learnt a lot about routing, but we don’t know if we enter something in the URL say localhost:4200/something - what would happen?
2. We will get an error because we don’t have a route – /something. So here we will handle setting up the routs for which we don’t have setup and redirecting requests.
3. So, let’s start with redirecting; lets add a new component in our application through CLI. - ng g c page-not-found
4. For all the not defined paths or the random paths we need to provide ‘\*\*’ – double asterisk which is also called **wildcard route** for the redirection.
5. Make sure this is the last one in the constant of the routs because if it were above somewhere the routs below it will never execute.
6. app.module.ts:
7. import { BrowserModule } from '@angular/platform-browser';
8. import { NgModule } from '@angular/core';
9. import { FormsModule } from '@angular/forms';
10. import { HttpModule } from '@angular/http';
11. import { AppComponent } from './app.component';
12. import { HomeComponent } from './home/home.component';
13. import { ServersComponent } from './servers/servers.component';
14. import { UserComponent } from './users/user/user.component';
15. import { UsersComponent } from './users/users.component';
16. import { EditServerComponent } from './servers/edit-server/edit-server.component';
17. import { ServerComponent } from './servers/server/server.component';
18. import { ServersService } from './servers/servers.service';
19. import { Routes, RouterModule } from '@angular/router';
20. import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
22. const appRoutes: Routes = [
23. { path: '', component: HomeComponent },
24. { path: 'users', component: UsersComponent,children:[
25. { path: ':id/:name', component: UserComponent }
26. ] },
27. { path: 'servers', component: ServerComponent, children:[
28. { path: ':id', component: ServerComponent },
29. { path: ':id/edit', component: EditServerComponent }
30. ]},
31. {path: 'not-found', component: PageNotFoundComponent},
32. {path: 'something', redirectTo: '/not-found'}
33. ];
34. @NgModule({
35. declarations: [
36. AppComponent,
37. HomeComponent,
38. UsersComponent,
39. ServersComponent,
40. UserComponent,
41. EditServerComponent,
42. ServerComponent,
43. PageNotFoundComponent
44. ],
45. imports: [
46. BrowserModule,
47. FormsModule,
48. HttpModule,
49. RouterModule.forRoot(appRoutes)
50. ],
51. providers: [ServersService],
52. bootstrap: [AppComponent]
53. })
54. export class AppModule { }

7. page-not-found.component.html:

<h3>This page was not Found!</h3>

**Section 11: Lecture 132//Important: Redirection Path Matching**

In our example, we didn't encounter any issues when we tried to redirect the user. But that's not always the case when adding redirections.

By default, Angular matches paths by prefix. That means, that the following route will match both /recipes  and just /

{ path: '', redirectTo: '/somewhere-else' }

Actually, Angular will give you an error here, because that's a common gotcha: This route will now ALWAYS redirect you! Why?

Since the default matching strategy is "prefix" , Angular checks if the path you entered in the URL does start with the path specified in the route. Of course every path starts with ''  (Important: That's no whitespace, it's simply "nothing").

To fix this behavior, you need to change the matching strategy to"full" :

{ path: '', redirectTo: '/somewhere-else', pathMatch: 'full' }

Now, you only get redirected, if the full path is ''  (so only if you got NO other content in your path in this example).

**Section 11: Lecture 133//Outsourcing the routeConfiguration**

1. We have seen that routing takes some significant space in our app.module.ts; typically if we have more than 2 or 3 routes we don’t add them in the app module, instead we add a new file which is for the application as the whole.
2. We will name this file as app-routing.module.ts, now from here onwards we will hold a second module. We have a whole module section about angular modules later in the course and there we will learn about defining different modules and using them together.
3. This is a super easy use case here so we can add it now and here we need to add the decorator @NgModule, which shall be imported from the @angular/core.
4. We can basically build our angular app from multiple modules. Here this simple module will handle all the routing related tasks in our application. We will cut all the app routes constants and move to this new module.
5. Now, as we moved routing part in the app-routing.module.ts, now we need to remove below line from the app.modul.ts

RouterModule.forRoot(appRoutes)

1. Now, we will make sure that we import the RouterModule from the @angular/router
2. Now, we need to configure some things in the @NgModule decorator. Here we will add imports again. And in the imports inside the decorator we will add RouterModule.forRoot(appRoutes)
3. We need to use module for routing as the separate thing, but we need to define it in the app.module.ts. We will simply use app-routing.module.ts to outsource our routes, that’s the reason we need to add the AppRoutingModule to our main module – and for this we need to add exports array in the @NgModule decorator along with imports array. We will dive deeper into the modules later in this course.
4. Exports just tells that if we want to add this module to the imports of the other module - what should be accessible to that module which imports this module. One thing which we want to make accessible RouterModule. So, now in the app.module.ts we can now import our own AppRoutingModule.
5. So, in the imports of the app.module.ts we will add AppRoutingModule
6. app.module.ts:
7. import { BrowserModule } from '@angular/platform-browser';
8. import { NgModule } from '@angular/core';
9. import { FormsModule } from '@angular/forms';
10. import { HttpModule } from '@angular/http';
11. import { AppComponent } from './app.component';
12. import { HomeComponent } from './home/home.component';
13. import { ServersComponent } from './servers/servers.component';
14. import { UserComponent } from './users/user/user.component';
15. import { UsersComponent } from './users/users.component';
16. import { EditServerComponent } from './servers/edit-server/edit-server.component';
17. import { ServerComponent } from './servers/server/server.component';
18. import { ServersService } from './servers/servers.service';
19. import { Routes, RouterModule } from '@angular/router';
20. import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
21. import { AppRoutingModule } from './app-routing.module';
23. @NgModule({
24. declarations: [
25. AppComponent,
26. HomeComponent,
27. UsersComponent,
28. ServersComponent,
29. UserComponent,
30. EditServerComponent,
31. ServerComponent,
32. PageNotFoundComponent
33. ],
34. imports: [
35. BrowserModule,
36. FormsModule,
37. HttpModule,
38. AppRoutingModule
39. //RouterModule.forRoot(appRoutes)
40. ],
41. providers: [ServersService],
42. bootstrap: [AppComponent]
43. })
44. export class AppModule { }

12. app-routing.module.ts

import { NgModule } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

import { HomeComponent } from "./home/home.component";

import { UsersComponent } from "./users/users.component";

import { UserComponent } from "./users/user/user.component";

import { ServerComponent } from "./servers/server/server.component";

import { EditServerComponent } from "./servers/edit-server/edit-server.component";

import { PageNotFoundComponent } from "./page-not-found/page-not-found.component";

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent,children:[

{ path: ':id/:name', component: UserComponent }

] },

{ path: 'servers', component: ServerComponent, children:[

{ path: ':id', component: ServerComponent },

{ path: ':id/edit', component: EditServerComponent }

]},

{path: 'not-found', component: PageNotFoundComponent},

{path: '\*\*', redirectTo: '/not-found'}

];

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

],

exports:[

RouterModule

]

})

export class AppRoutingModule {

}

**Section 11: Lecture 134//An Introduction to Guards**

1. Throughout the section we learnt about routing, we know how to setup routes and learnt how to load the routes both through routerLink or Programmatically, how to pass params, query params and how to use data; how to load different components and much more.
2. One important feature is left in this module which I want to cover – **Route Guards**.
3. So basically functionality, logic, code which is executed before a route is loaded or once you want to leave a route.
4. Let’s start with the simple use case – let’s say you only want to give access to your server component i.e. your single server component or the edit-server.component.ts
5. If a user is locked in now we’re not going to add a full login functionality for this.
6. We will later have the authentication section in this course but we will fake it now, still we want to check it before any of our sub routes here i.e. servers 3, servers 1 and then also the added routes are accessed now manually checking this in the onInit method in our server and edit-server component because that would be very cumbersome because you wouldn’t have to add it in both components and if you add more components belonging to the service feature you would have to add this over and over again.
7. So, therefore we want to use a feature built into the angular router running some code before the component is loaded.
8. We will use the can activate guard and we will do so in the next lecture.

**Section 11: Lecture 135//Protecting Routes with canActivate**

1. As mentioned in the last lecture we want to maybe protect some of our routes and we can use a feature offered by angular which allows us to run some code at a point of time defined by us.
2. We will add a new file in our root folder and we will name it auth-guard.service.ts, because the code we run there is stored in such a service.
3. We will name it auth-guard though because the feature of the Angular router is called guards, it guards certain actions - like navigating to a route or around a route or away from it but in the end it is the normal service and therefore as such I will export a class which will name AuthGuardService and there we will implement wait interface i.e. canActivate interface and a canActivate(){} method in this class.
4. The canActivate method now will receive 2 arguments i.e. the ActivatedRouteSnapshot and the RouterStateSnapshot.
5. These 2 parameters will be injected by angular for us in the method. canActivate also returns something – it either returns an observable and this observable will wrap a Boolean in the end which will resolve to a true or false; alternatively, this router returns a promise – which will also resolve to a Boolean value or it returns just a Boolean.
6. canActivate can run Asynchronously or synchronously returning an observable or promise, because you might have some guards which execute some code which runs completely on the client. Therefore, it runs synchronously or you might have some code which takes couple of seconds to finish because you use a timeout in there or you reach out to a server, so, it runs asynchronously and both is possible with canActivate. We’ll see an example in a second.
7. So, here we want to be able to log in or out, for this let’s say we have another service say – auth.service.ts - which is the fake service here. This might reach out to server and allow login or logout and check our current authentication state.
8. We will track state here too with the loggedIn property which is set to false initially and then I’ll add a login method which will set loggedIn to true and the logout method which will set loggedIn to false again to fake out the behaviour.
9. And we will add a method which allows us to check the state i.e. authenticate. In this method we will return a promise. This promice will always take a function as an argument with the resolve and reject method we can execute.
10. In this promise we will execute set timeout method to wait let’s say 800 mili-seconds and then execute another method in which we will resolve the promise i.e. again just to fake that this takes a couple of seconds.
11. So, with this off service added, I want to use it in our AuthGuard service as this is also a service, we need to add @Injectable to be able to reachout to the other serevice to inject this service into this guard. And then I will add a constructor in the AuthGuard, where I reach out to my fake off service here and type of service .
12. Now, we must check if the user is logged in or not. So we will call here isAuthenticate method of the AuthService, which again returns the promise.
13. Then here I would need to be able to handle that when the promise in the AuthService resolves. Now, if it is authenticated the we will return true and otherwise I want to navigate away because I don’t want to allow the uiser access to the route you wanted to go to originally.
14. So, if the authentication fails we will navigate the user away to allow him to go somewhere else, so we will inject the angular router for this in the constructor. And then in this we will navigate to the root page say. So, in this case our can navigate method will navigate away.
15. Now, we will return this promise in the end.
16. Now, we need to define which route should be protected by this guard in the app-routing.module.ts.
17. So, here we will add canActivate property to servers route. canActivate takes the array of code basically containing all the guards that needs to be applied and all those will also be applied to the child routes.
18. canActivate array will take here authGuard as the value in the array.
19. Before trying anything lets go to app module and define the 2 new services we created.
20. app-routing.module.ts:
21. import { NgModule } from "@angular/core";
22. import { Routes, RouterModule } from "@angular/router";
23. import { HomeComponent } from "./home/home.component";
24. import { UsersComponent } from "./users/users.component";
25. import { UserComponent } from "./users/user/user.component";
26. import { ServerComponent } from "./servers/server/server.component";
27. import { EditServerComponent } from "./servers/edit-server/edit-server.component";
28. import { PageNotFoundComponent } from "./page-not-found/page-not-found.component";
29. import { AuthGuard } from "./auth-guard.service";
30. const appRoutes: Routes = [
31. { path: '', component: HomeComponent },
32. { path: 'users', component: UsersComponent,children:[
33. { path: ':id/:name', component: UserComponent }
34. ] },
35. { path: 'servers',canActivate: [AuthGuard], component: ServerComponent, children:[
36. { path: ':id', component: ServerComponent },
37. { path: ':id/edit', component: EditServerComponent }
38. ]},
39. {path: 'not-found', component: PageNotFoundComponent},
40. {path: '\*\*', redirectTo: '/not-found'}
41. ];
43. @NgModule({
44. imports: [
45. RouterModule.forRoot(appRoutes)
46. ],
47. exports:[
48. RouterModule
49. ]
50. })
51. export class AppRoutingModule {
52. }

21. app.module.ts

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

import { HomeComponent } from './home/home.component';

import { ServersComponent } from './servers/servers.component';

import { UserComponent } from './users/user/user.component';

import { UsersComponent } from './users/users.component';

import { EditServerComponent } from './servers/edit-server/edit-server.component';

import { ServerComponent } from './servers/server/server.component';

import { ServersService } from './servers/servers.service';

import { Routes, RouterModule } from '@angular/router';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

import { AppRoutingModule } from './app-routing.module';

import { AuthGuard } from './auth-guard.service';

import { AuthService } from './auth.service';

@NgModule({

declarations: [

AppComponent,

HomeComponent,

UsersComponent,

ServersComponent,

UserComponent,

EditServerComponent,

ServerComponent,

PageNotFoundComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule,

AppRoutingModule

//RouterModule.forRoot(appRoutes)

],

providers: [ServersService, AuthGuard, AuthService],

bootstrap: [AppComponent]

})

export class AppModule { }

1. auth.service.ts:
2. import { logging } from "protractor";
3. export class AuthService{
4. loggedIn = false;
5. isAuthenticated(){
6. const promise = new Promise(
7. (resolve, reject) => {
8. setTimeout(()=>{
9. resolve(this.loggedIn)
10. }, 800);
11. }
12. );
13. return promise;
14. }
15. loggedInMethod(){
16. this.loggedIn = true;
17. }
18. loggedOutMethod(){
19. this.loggedIn = false;
20. }
21. }
23. auth-guard.service.ts:
24. import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from "@angular/router";
25. import { Observable } from "rxjs/Observable";
26. import { Injectable } from "@angular/core";
27. import { AuthService } from "./auth.service";
28. @Injectable()
29. export class AuthGuard implements CanActivate{
30. constructor(private authService: AuthService, private router: Router){}
31. canActivate(route: ActivatedRouteSnapshot,
32. state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> |boolean{
33. // return this.authService.isAuthenticated()
34. return this.authService.isAuthenticated()
35. .then(
36. (Authenticated: boolean)=>{
37. if (Authenticated){
38. return true;
39. }else{
40. this.router.navigate(['/']);
41. }
42. }
43. );
44. }
45. }

**Section 11: Lecture 136//Protecting Child (Nested) Routes with canActivatedChild**

1. In the last lecture we added the canActivate guard and it was working fine but it was working for our whole servers path here.
2. Now, we can remove it from there and add it to only childs so that the child id protected and not our root path but that is not the easiest way because we add more child items we have to add canActivate for each of them.
3. There is another guard we can use, it’s pretty similar to can activate; it’s called canActivateChild - lets implement this interface too and we need to import it from @angular/router.
4. Now, this interface requires a canActivateChild method in this class which basically takes the same fold as canActivate method.
5. Well, since this is exactly the same form and we want to run the same logic we can simply call this.canActivate(); - now, here we will add 2 arguments i.e. route and state since canActivate excepts these arguments.
6. What is the advantage of adding this then, well since we added the canActivate child interface; so, now we can use a different hook in our routes instead.
7. Instead of using only canActivate we can use a different one and we can split this over multiple lines and comment canActivate method and instead of that we will add canActivateChild
8. canActivateChild also takes the array of services which acts as guards, which implement the right interfaces and here we can still add the off guard and now is able to do both protect a single route since we have canActivate implemented or all child routes since we have canActivateChild implemented too.
9. app-routing.module.ts
10. import { NgModule } from "@angular/core";
11. import { Routes, RouterModule } from "@angular/router";
12. import { HomeComponent } from "./home/home.component";
13. import { UsersComponent } from "./users/users.component";
14. import { UserComponent } from "./users/user/user.component";
15. import { ServerComponent } from "./servers/server/server.component";
16. import { EditServerComponent } from "./servers/edit-server/edit-server.component";
17. import { PageNotFoundComponent } from "./page-not-found/page-not-found.component";
18. import { AuthGuard } from "./auth-guard.service";
19. const appRoutes: Routes = [
20. { path: '', component: HomeComponent },
21. { path: 'users', component: UsersComponent,children:[
22. { path: ':id/:name', component: UserComponent }
23. ] },
24. { path: 'servers',
25. //canActivate: [AuthGuard],
26. canActivateChild:[AuthGuard] ,
27. component: ServerComponent, children:[
28. { path: ':id', component: ServerComponent },
29. { path: ':id/edit', component: EditServerComponent }
30. ]},
31. {path: 'not-found', component: PageNotFoundComponent},
32. {path: '\*\*', redirectTo: '/not-found'}
33. ];
35. @NgModule({
36. imports: [
37. RouterModule.forRoot(appRoutes)
38. ],
39. exports:[
40. RouterModule
41. ]
42. })
43. export class AppRoutingModule {
44. }
45. auth-guard.service.ts:
46. import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router, CanActivateChild } from "@angular/router";
47. import { Observable } from "rxjs/Observable";
48. import { Injectable } from "@angular/core";
49. import { AuthService } from "./auth.service";
50. @Injectable()
51. export class AuthGuard implements CanActivate, CanActivateChild{
52. constructor(private authService: AuthService, private router: Router){}
53. canActivate(route: ActivatedRouteSnapshot,
54. state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> |boolean{
55. // return this.authService.isAuthenticated()
56. return this.authService.isAuthenticated()
57. .then(
58. (Authenticated: boolean)=>{
59. if (Authenticated){
60. return true;
61. }else{
62. this.router.navigate(['/']);
63. }
64. }
65. );
66. }
67. canActivateChild(route: ActivatedRouteSnapshot,
68. state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> |boolean{
69. return this.canActivate(route, state);
70. }
71. }

**Section 11: Lecture 137//Using Fake Auth Service**

1. Now, let’s finish this canActivate behavior by allowing the user to log in or quickly do this by going – by allowing the user to log in.
2. In the home component we will add two new buttons – i.e. 1 login button and one logout button and we will add some morel and here we will add click methods i.e. logout and login.
3. We will keep a simple UI here as it’s not about having a nice UI here. We will simply inject our AuthService here.
4. [home.component.html](http://home.component.html):
5. <h4>Welcome to Server Manager 4.0</h4>
6. <p>Manage your Servers and Users.</p>
7. <button class ="btn btn-primay" (click) = "onLoadServers()">Load Servers 1</button>
8. <button class="btn btn-default" (click) = "onLogin()">Login</button>
9. <button class="btn btn-default" (click)="onLogout()">Logout</button>
10. )
11. [home.component.ts](http://home.component.ts):
12. import { Component, OnInit } from '@angular/core';
13. import { Router } from '@angular/router';
14. import { AuthService } from '../auth.service';
15. @Component({
16. selector: 'app-home',
17. templateUrl: './home.component.html',
18. styleUrls: ['./home.component.css']
19. })
20. export class HomeComponent implements OnInit {
21. constructor(private router: Router, private authService: AuthService ) { }
22. ngOnInit() {
23. }
24. onLoadServers(id: number){
25. //complex calculation
26. this.router.navigate(['/servers',id, 'edit'], {queryParams: {allowEdit: '1'}, fragment:'loading'});
27. }
28. onLogin(){
29. this.authService.login();
30. }
31. onLogout(){
32. this.authService.logout();
33. }
34. }

6. auth.service.ts:

import { logging } from "protractor";

export class AuthService{

loggedIn = false;

isAuthenticated(){

const promise = new Promise(

(resolve, reject) => {

setTimeout(()=>{

resolve(this.loggedIn)

}, 800);

}

);

return promise;

}

login(){

this.loggedIn = true;

}

logout(){

this.loggedIn = false;

}

}

**Section 11: Lecture 138//Controlling Navigation with canDeactivate**

1. In the last lectures we learnt how to use canActivate control access to a route, now I want to focus on to the control of wether you are allowed to leave a rote or not. We can control this too and we might want to control this if we are logged in once we do edit a server.
2. Now, we are allowed to edit the dev server, and here if I want to change something I want to ask the user if he accidently clicks the back or somewhere if you want to leave off if you maybe forgot to click update server first.
3. So, this convenient method of keeping the user from accidently navigating away. How can we implement this.
4. Lets go to the edit serve component for this here and lets add a new property i.e. changesSaved - we will keep this property as false initially and later we will changes this property whenever we click update server. After the changes are saved we want to navigate away. So, we will inject the router here and we will add the route to navigate away in OnUpadteServer
5. So, here we will navigate away one level up to the last loaded server and for this we will add relativeTo configuration in the object as the second parameter of the navigate method and then will navigate relative to currently active route.
6. Now, let’s make sure that whenever the user tries to accidently navigate a way that we prevent him from doing so; we will atleast ask if he really wants to leave now.
7. So, we somehow nee to execute this code in edit-server.component here. Here we will check the property changesSaved to check if the update method was clicked or not.
8. However, a guard always needs to be a service because we need to provide it just like we provided the off guard. So, somehow this doesn’t fit – we need to access to code our component and have a service and that sounds like a complicated setup but isn’t, but its actually very simple .
9. In our edit-server folder we will add a new file and we will name it can-deactivate-guard.service.ts. And this also will be a service again as all guards are and in here we first of all now want to export **an interface simply is a Contract which can be imported by some other class**. Interface forces this class to provide some logic.
10. So, here I will name it the scan component deactivate and this interface will require one thing form the component that implements it.
11. This component should have a can deactivate method. So, here we will add here the return type
12. Since it is a interface, it will not contain the actual logic, it will only contain the information that how, it should look like.
13. So this method should take no arguments but in the end it should return the observable. Now, let’s come to the important thing.
14. So, that is our interface – nice to have it - but that alone won’t do the trick, now let’s come to a important thing – the meet of the class of service his is say the service itself.
15. So, we will name it CanDeactivateGuard and it will implement an interface provided by angular routers called **CanDeactivate.** This is the interface provided by the angular router.
16. CanDeactivate is the generic type so we must define the type – in this type we will wrap our own interface
17. So, it warps the interface which forces some class to implement the canDeactivate method – this might sound complicated but this is the setup which will make sure that we connect a component to canDeactivate guard here.
18. So, for now let’s go with that piece of information; now this class/ this guard here also needs to implement the can deactivate method here - this is the canDeactivate method that will be implemented by angular router once we try to leave a route. Therefore this will have the component- which we are currently on as the argument and this component needs to be of the type CanComponentDeactivate – which means that this needs to be a component which has CanComponentDeactivate interface implemented. We also will receive the current route as an argument i.e. that will be the ActivatedRouteSnapshot.
19. It will also have the current state and the next state i.e. where do we want to go in the end when we want to leave the route. So the nextState is optional argument though which is of the type RouterStateSanpshot. And this will also return a Promise, Observable or a Boolean
20. That is how the can deactivate method in our can deactivate guard looks like. Now here in the canDeactivate method we will return the component on which we are currently on and this is the reason I need to implement this interface in this component.
21. Now, angular router can activate canDeactivate in our service and can rely on the fact that the component we are currently on has the canDeactivate method too because this is where we are allowed to check the logic checking whether we are allowed to leave a component or not – because we need this connection between our guard an component.
22. Now, with that information we can go back to our app routing module and on the edit page in the constant here we want to add this guard.
23. Now the Angular will run this whenever we leave this path here. For this to work we need to provide canDeactivate in the providers of the app.module.ts
24. We are almost there but one important piece is missing, remember that we discussed that CanDeactivateGuard will call canDeactivate on our current component, now, for this to weok in our edit-server.component.ts we will implement CanComponentDeactivate, which is our own Interface which we exported in CanDeactivateGuard service file. This interface now forces us to implement the canDeactivate method on our component – which is important because we tried to call this method on our components here so in the edit servers component here I will add canDeactivate method.
25. Now in this canDeactivate method implemented on this component above we will write the actual logic that we are allowed to leave or not. This logic will run when we deactivate guar is checked by the angular router.
26. So, here we will first check whether we can check this or not.
27. Now, there is one more guard we want to dive into, that we will do in next lecture.