Section 11: Changing Pages with Routing: Table of Contents

[126. Setting up and Loading Routes 2](#_Toc63521051)

[127. Navigation with Router Links 2](#_Toc63521052)

[128. Understanding Navigation Paths 3](#_Toc63521053)

[129. Styling Active Router Links 3](#_Toc63521054)

[130. Navigating Programmatically 3](#_Toc63521055)

[131. Using Relative Paths in Programmatic Navigation 4](#_Toc63521056)

[132. Passing Parameters to Routes 4](#_Toc63521057)

[133. Fetching Route Parameters 4](#_Toc63521058)

[134. Fetching Route Parameters Reactively 4](#_Toc63521059)

[135. An Important Note about Route Observables 5](#_Toc63521060)

[136. Passing Query Parameters and Fragments 6](#_Toc63521061)

[137. Retrieving Query Parameter and Fragments 6](#_Toc63521062)

[138. Practicing and some Common Gotchas 7](#_Toc63521063)

[139. Setting up Child (Nested) Routes 7](#_Toc63521064)

[140. Using Query Parameters Practice 8](#_Toc63521065)

[141. Configuring the Handling of Query Parameters 8](#_Toc63521066)

[142. Redirecting and Wildcard Routes 9](#_Toc63521067)

[143. Important Redirection Path Matching 9](#_Toc63521068)

[144. Outsourcing the Route Configuration 10](#_Toc63521069)

[146. Protecting Routes with canActivate 10](#_Toc63521070)

[147. Protecting Child (Nested) Routes with canActivateChild 12](#_Toc63521071)

[149. Controlling Navigation with canDeactivate 12](#_Toc63521072)

[150. Passing Static Data to a Route 13](#_Toc63521073)

[151. Resolving Dynamic Data with the resolve Guard 14](#_Toc63521074)

[152. Understanding Location Strategies 15](#_Toc63521075)

# 127. Setting up and Loading Routes

In order to enable Angular routers, you need to first import Routes and RouterModule from @angular/router. Our routes will be stores in an array of type routes and should be objects that include a path and tied component. Ex:

1. const appRoutes: Routes = [
2. { path: '', component: HomeComponent },
3. { path: 'users', component: UsersComponent },
4. ];

In order to register these routes, however, you need to include the following RouterModule.forRoot() method in the imports array of the @NgModule decorator:

1. @NgModule({
2. imports: [
3. RouterModule.forRoot(appRoutes)
4. ]
5. })

You can then call upon these routes in the template by simply using the router-outlet directive as done below:

1. <router-outlet></router-outlet>

# 128. Navigation with Router Links

Instead of using an href in an anchor tag for routing, we want to user the Angular routing directive, routerLink. routerLink can simply set a path by setting it equal to a string with a / and a path name, or you can property bind it to an array of strings to allow for child paths to be routed.

1. <a routerLink="/servers">Servers</a>
2. <a [routerLink]="['/users', 'secondaryPath]">Users</a>

routerLink works better than href because it does not perform a full reload of the page, meaning you won't lose the state of your application every time you click on a new path.

# 129. Understanding Navigation Paths

Router links behave much like a tiered folder system. The paths are dictated by either relative or absolute references. By beginning your path with /, you are instructing Angular to use an absolute path that is appended to the root domain. However, by beginning your path with nothing or ./, you are instructing Angular to use a relative path from the path your are currently on. Much like with folder navigation, using  ../ will instruct Angular to up a path level relative to the current path.

# 130. Styling Active Router Links

With Angular, you are able to add the routerLinkActive directive to wrapping elements like li , or directly to an anchor itself, with the expression entered being the class we wish to enact. In the case of our bootstrap tabs list item, we set the class, active. Since paths like the root path may interfere with styling since it is always active, you can include the [routerLinkActiveOptions] directive with the bound expression being an object with the exact property set to true. Ex:

1. <ul class="nav nav-tabs">
2. <li role="presentation"
3. routerLinkActive="active"
4. [routerLinkActiveOptions]="{exact: true}">
5. <a routerLink="/">Home</a>
6. </li>
7. </ul>

# 131. Navigating Programmatically

Routing can be accomplished programmatically as well, not just in the template code. To do so from a component typescript file, we first want to import and inject Router into the component file via the constructor. Once it's done, we can then use the router's navigate() method, which lets us insert a path in an array, similar as to when property binding the routerLink directive. Ex:

1. constructor(private router: Router) { }
3. onLoadServers() {
4. // complex calculation
5. this.router.navigate(['/servers']);
6. }

# 132. Using Relative Paths in Programmatic Navigation

Unlike the routerLink directive used in the template, the Router navigate() method does not know which path you are currently on, so it assumes the root by default. By injecting an instance of ActivatedRoute (imported from @angular/router), you can insert an object with the relativeTo property set as the ActivatedRoute instance into the navigate() method as a second argument. This lets you resolve the relative reference being used. Ex:

1. constructor(private router: Router,
2. private route: ActivatedRoute) { }
4. onReload() {
5. this.router.navigate(['servers'], {relativeTo: this.route});
6. }

# 133. Passing Parameters to Routes

You can pass parameters to our routes by adding the colon operator : followed by any name. Doing so adds a dynamic condition to the path, where you can enter anything after the users/ part of the url and the UserComponent will load as a result. Ex:

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

# 134. Fetching Route Parameters

In order to fetch parameters that have been passed via the colon operator in the app module, we need to inject an instance of the ActivatedRoute class again. With it, we can assign variables/arrays/objects to the instance parameters with the activatedRoute.snapshot.params() method as done below:

1. this.user = {
2. id: this.route.snapshot.params['id'],
3. }

# 135. Fetching Route Parameters Reactively

When you update a URL of a component from within the component, Angular cleverly does not re-instantiate the component by default, in order to keep performance up. Angular is not aware when the data affecting the url might have changed, so we need to subscribe to changes via the Params observable on the ActivatedRoute, which also need to be imported from @angular/router.

Observables are features added by a 3rd-party package, not Angular, that allow you to work with asynchronous tasks and execute code when something happens unexpectedly.

snapshot.params is fine to use for a first initialization of a component, but otherwise you want to use just params.subscribe() to react to subsequent changes in the data. Ex:

1. this.route.params.subscribe(
2. (params: Params) => {
3. this.user.id = params['id'];
4. }
5. );

If you don't need to update the component from within in, you don't need to subscribe and can simply just use snapshot.params.

# 136. An Important Note about Route Observables

As subscriptions are not tied to components in Angular, normally they will keep active in memory even after a component has been destroyed. Luckily, observables that Angular is familiar with are typically cleaned up, but it's important to know how to unsubscribe from observables, and it doesn't hurt to do it anyways. First, you need to create an instance of the Subscription class, which is imported from rjxs/Subscriptions. Then, you can set the instance equal to the activatedRoute.params observable that is utilizing the subscribe() method. Ex:

1. this.paramsSubscription = this.route.params.subscribe(...)

Then, you will simply want to use the unsubscribe() method of the Subscription instance in the ngOnDestroy() {...} lifecycle hook. Ex:

1. ngOnDestroy() {
2. this.paramsSubscription.unsubscribe();
3. }

# 137. Passing Query Parameters and Fragments

In order to pass a query parameter or fragment to a url path, there are properties of the routerLink directive that will enable proper creation of these things. Specifically, the [queryParams] and [fragment] properties. The [queryParams] property expects an object with key-value pairs, and the [fragment] property expects a single quotation string within double quotation marks (however, it can also be unbound and just given double quotation marks). Ex:

1. <a
2. [routerLink]="['/servers', 5, 'edit']"
3. [queryParams]="{allowEdit: '1'}"
4. [fragment]="'loading'"> <!-- Or: fragment= "loading" -->

Similarly, these can also be used dynamically with the Router.navigate() method as a second input argument object with the two properties separated by comma inside. Ex:

1. onLoadServers(id: number) {
2. this.router.navigate(['/servers', id, 'edit'],
3. {queryParams: {allowEdit: '1'}, fragment: 'loading'});
4. }

# 138. Retrieving Query Parameter and Fragments

To get access to the queryParams and fragment data on the receiving side, we need to first import and create an instance of the ActivatedRoute class. From there, you can access the data one of two ways. The first way is to access the snapshot of our activated route, and call either the queryParams or fragment properties. Ex:

1. console.log(this.route.snapshot.queryParams);
2. console.log(this.route.snapshot.fragment);

The other method, which should be used if we need to be attentive to updates of the data in the component is the subscribe method, as both queryParams and fragment are observables. Like with the params observable, both of these will automatically get cleaned up by Angular upon component destruction. Ex:

1. this.route.queryParams.subscribe(...);
2. this.route.fragment.subscribe(...);

# 139. Practicing and some Common Gotchas

Something to watch out for is when using the snapshot.params observable. Whenever we try to access the id stored in our path, it gives use errors. In this case, these errors are occurring because the params returns a string value while our servers service is expecting a number. You can easily turn the string into a number by adding the plus operator + in front of the id call. Ex:

1. const id = +this.route.snapshot.params['id'];
2. this.server = this.serversService.getServer(id);
3. this.route.params.subscribe(
4. (params: Params) => {
5. this.server = this.serversService.getServer(+params['id']);
6. }
7. );

# 140. Setting up Child (Nested) Routes

One problem is that we have no way to insert children components inside of their parents so that a new page isn't loaded for each sub-component. Child paths help to fix this, and can be created by adding a second key-value pair into our path objects. the key is children and the value entered is an array of the children path objects, with the parent path name removed from the children path string names. Ex:

1. { path: 'users', component: UsersComponent, children: [
2. { path: ':id/:name', component: UserComponent },
3. ] },

Angular will be able to tell that these children paths should not be included in the router-outlet directive used in the app component, but will be made available via the same directive in their parent paths. For example, the UserComponent will be accessible in the UsersComponent when the following line is entered in the template file:

<router-outlet></router-outlet>

# 141. Using Query Parameters Practice

When using queryParams, it is often useful to utilize ternary expressions in order to determine things like page access, etc. Here, we set allow edit to true if a selected server has an id equal to 3, and only show the EditServerComponent if allowEdit is true.

Parent template Ex:

1. <a
2. [routerLink]="['/servers', server.id]"
3. [queryParams]="{allowEdit: server.id === 3 ? '1' : '0'}">
4. </a>

Child Typescript Ex:

1. this.route.queryParams.subscribe(
2. (queryParams: Params) => {
3. this.allowEdit = queryParams['allowEdit'] === '1' ? true : false;
4. }
5. );

Child Template Ex:

1. <h4 \*ngIf="!allowEdit">You're not allowed to edit!</h4>
2. <div \*ngIf="allowEdit">
3. ...
4. </div>

# 142. Configuring the Handling of Query Parameters

Within the second argument javascript object that is entered into the navigate method, there is another property that we can set in order to make sure that query parameters are passed on from the current path to the new one. That property is the queryParamsHandling property, and there are two useful values to assign it with for the time being, 'preserve' and 'merge' . The preserve value passes on all of the old values, while the merge value with merge new and old query parameter, with the old value being overwritten.

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

# 143. Redirecting and Wildcard Routes

With the creation of a page-not-found component, it is possible to set up your paths so that all unknown paths are redirected to the new component. With the path defined for the page-not-found component, you can create a new path with path: '\*\*' and redirectTo: '/not-found' in place of a component declaration, for instance.  \*\* is is treated as a wild card, and redirectTo directs to another existing path, in this case that of the page-not-found-component. Ex:

1. { path: 'not-found', component: PageNotFoundComponent },
2. { path: '\*\*', redirectTo: '/not-found' }

The wild card redirect path **MUST ALWAYS** be the last path entered as paths are parsed from top to bottom.

# 144. 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).

# 145. Outsourcing the Route Configuration

Routing for a project can be outsourced to its own AppRoutingModule class (saved as app-routing.module.ts for instance).

A custom module will look similar to the app module in that you will have your imports, @NgModule decorator, and your class export, along with the appRoutes const created. NgModule needs to be imported from @angular/core, and RouterModule & Routes both need to be imported from @angular/router. All components called in a path need to be imported as well.

In the decorator, you'll simply have your RouterModule import, but you'll also need an export for the RouterModule after the paths have been loaded:

1. const appRoutes: Routes = [...];
3. @NgModule({
4. imports: [RouterModule.forRoot(appRoutes)],
5. exports: [RouterModule]
6. })
8. export class AppRoutingModule {}

# 147. Protecting Routes with canActivate

A guard is a service that prevents specific paths/components from being accessed without meeting some sort of baseline condition. In order to create a guard, we must make a service that imports the CanActivate interface and implements it in the class declaration. Additionally, a method by the name of canActivate() must also be created in the class. The following things must be imported, and the following service outline should to be used for the guard:

1. import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot
2. } from "@angular/router";
3. import { Injectable } from "@angular/core";
4. import { Observable } from 'rxjs/Observable';
6. @Injectable()
7. export class AuthGuard implements CanActivate {
9. canActivate(route: ActivatedRouteSnapshot,
10. state: RouterStateSnapshot): Observable<boolean> |
11. Promise<boolean> | boolean {
12. ...verification service here...
13. }
14. }

The following is an example of a service that could be used to authenticate. Here, a variable of type Promise object is used. A promise allows you to associate handlers with an asynchronous task's eventual success or failure. The promise is resolved after a timeout, and then returned so that it can be handled by something like a guard service, for example.

1. export class AuthService {
2. loggedIn = false;
4. isAuthenticated() {
5. const promise = new Promise((resolve, reject) => {
6. setTimeout(()=> {
7. resolve(this.loggedIn)
8. }, 800 )
9. })
10. return promise;
11. }
13. login() { this.loggedIn = true; }
15. logout() { this.loggedIn = false; }
16. }

In the following code, we can call an AuthService and do an authentication check, which returns a promise. We can then use the .then() handler to check if the authentication was a pass (true) or fail (false), and either return true and navigate the user to a new path otherwise. However, when we return something inside of a promise, we are still returned a promise. So we need to add a return before the service authentication call.

1. canActivate(route: ActivatedRouteSnapshot,
2. state: RouterStateSnapshot): Observable<boolean> |
3. Promise<boolean> | boolean {
4. return this.authService.isAuthenticated()
5. .then((authenticated: boolean) => {
6. if (authenticated) {
7. return true;
8. } else {
9. this.router.navigate(['/']);
10. }
11. });
12. }

To implement a guard with the canActivate() interface in a path, we need to edit the router module. In the path object inside our Routes array, we will add in the canActivate property after the path property, and assign it to an array with our guard service passed inside. This will apply the guard to the route, along with all of its associated child routes.

1. { path: 'servers', canActivate: [AuthGuard],
2. component: ServersComponent, children: [...] }

Keep in mind that in the app module @NgModule decorator, any guard and authentication services also need to be included server-wide in the providers array.

# 148. Protecting Child (Nested) Routes with canActivateChild

Similar to CanActivate, we can implement the CanActivateChild interface in a guard class and include a canActivateChild() method that simply returns the result from the canActivate() method. This lets us get the same functionality as canActivate, but instead only applying it to children of the component in question.

1. canActivateChild(route: ActivatedRouteSnapshot,
2. state: RouterStateSnapshot): Observable<boolean> |
3. Promise<boolean> | boolean {
4. return this.canActivate(route, state);
5. }

Same as with canActivate, we simply include it in our path object and pass the guard into the canActivateChild property:

1. { path: 'servers', canActivateChild: [AuthGuard],
2. component: ServersComponent, children: [...] }

# 150. Controlling Navigation with canDeactivate

Assume we want to be able to block someone from leaving a path, similar to granting someone access. We can do so with canDeactivate, but it requires us to somehow intermingle our guard with some component variables. To do so, we will need to create an interface and class in a new deactivate guard service with the following format:

1. export interface CanComponentDeactivate {
2. canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean
3. }
5. export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {
6. canDeactivate(component: CanComponentDeactivate, currentRoute:
7. ActivatedRouteSnapshot, currentState: RouterStateSnapshot,
8. nextState?: RouterStateSnapshot): Observable<boolean> |
9. Promise<boolean> | boolean {
10. return component.canDeactivate();
11. }
12. }

This allows us to implement the interface in a component and require the canDeactivate method in it.

In order to be able to use a deactivate guard, we have to add it to our path and to the app as a whole. In the app routing module, we are able to add our guard to a path object as follows:

1. { path: ':id/edit', component: EditServerComponent,
2. canDeactivate: [CanDeactivateGuard] }

However, we also can't forget to add the deactivate guard service to the app module providers array.

In order to be able to use the deactivate service, we need to first add the CanComponentDeactivate interface that we created to the implements for the component class that is going to use it. With it implemented, we are forced to create a canDeactivate() method in the component, where we store our logic. In implementing this method, it's good practice to check if we are allowed to be there or if unsaved changes have been made, and if so confirming if the user really wants to leave. Ex:

1. if (!this.allowEdit) {
2. return true;
3. }
4. if ((this.serverName !== this.server.name || this.serverStatus !==
5. this.server.status) && !this.changesSaved) {
6. return confirm('Do you want to discard the changes?');
7. }
8. else {
9. return true;
10. }

# 151. Passing Static Data to a Route

In a router path object, we can pass static data to a path/component with the data property, which takes an object with key-value pairs to retrieve data with. Ex:

1. { path: 'not-found', component: ErrorPageComponent,
2. data: {message: 'Page not found!'} },

To receive this data on the component side, we will inject an instance of the ActivatedRoute class and use the data[] property of the ActivatedRoute.snapshot property.

1. this.errorMessage = this.route.snapshot.data['message'];

Alternatively, in case of changes while on the page, you can subscribe to the data as an observable that receives a variable of type Data to be imported from @angular/route:

1. this.route.data.subscribe(
2. (data: Data) => {
3. this.errorMessage = data['message'];
4. }
5. )

# 152. Resolving Dynamic Data with the resolve Guard

If we have a use case where we want to load data from some backend, we need a resolver, which is a service like CanActivate or CanDeactivate which will allow us to run some code before a route is rendered. However, the resolver will not decide if anything should be loaded or not.

A resolver will always render the component in the end, but it does some pre-loading to fetch some data the component may need before the route is rendered.

To create a resolve guard, we need to make a new service. If you don't have a model for your data, you can define an interface as done below:

1. interface Server {
2. id: number;
3. name: string;
4. status: string;
5. }

The class below lays out the general structure of the Resolve service, which needs to wrap your data type. It also needs to have the resolve() method, which holds your process for fetching data.

1. @Injectable()
2. export class ServerResolver implements Resolve<Server> {
3. constructor(private serversService: ServersService) {}
5. resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
6. Observable<Server> | Promise<Server> | Server {
7. return this.serversService.getServer(+route.params['id']);
8. }
9. }

In the app routing module, the resolve guard can be used as seen below. The property that is set equal to your guard should resemble the data that is represents, in this case: server.

1. { path: ':id', component: ServerComponent,
2. resolve: {server: ServerResolver} },

In the component that would need to fetch the data grabbed by the resolver, you can simply subscribe to the data observable (as with static data) and utilize the custom 'server' data.

1. ngOnInit() {
2. this.route.data.subscribe((data: Data) => {
3. this.server = data['server'];
4. });
5. }

# 153. Understanding Location Strategies

One issue you may run into is how a server plays with the angular routing. When some paths are selected, the host server may take control and product a 404 error instead of deferring to the angular app's index.html file. One way to avoid this is the insert a hash tag  # in the route as done below in the routing module imports:

1. RouterModule.forRoot(appRoutes, {useHash: true})

With this, everything to the left of the hash is handled by the server, while everything to the right of the has is handled by the angular app.