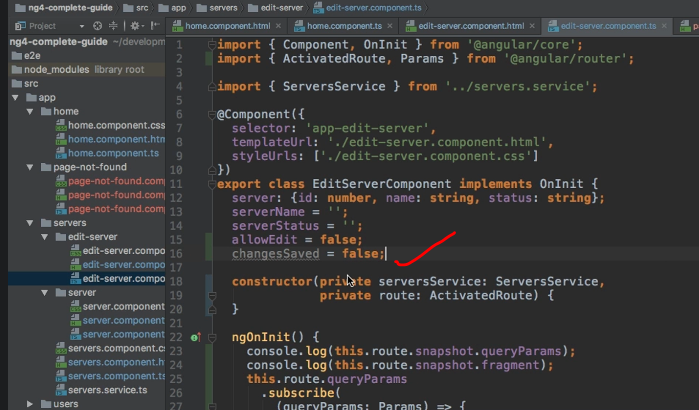
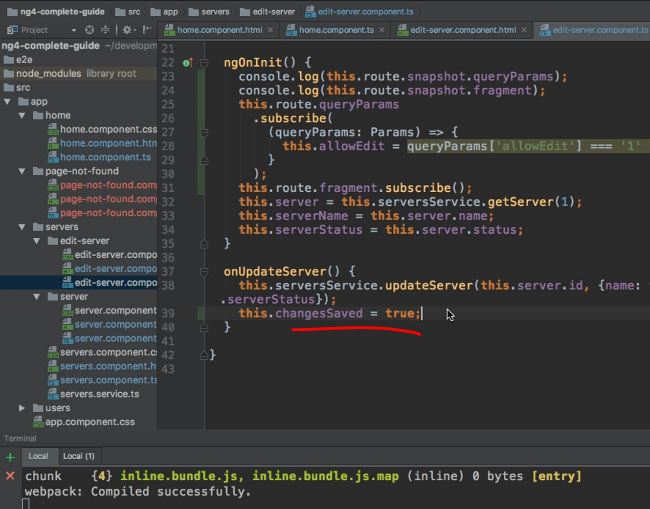
**149. Controlling Navigation with canDeactivate**

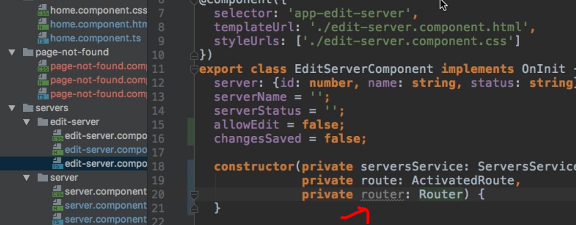
* Instructor: In the last lectures, we learned how to use CanActivate to control access to a route.
* *Now I wanna focus on the control of whether you are allowed to leave a route or not.*
* We can control this too, and we might want to control this if we are locked in, once we do edit a server.
* Now we, for example, are allowed to edit the Devserver.
* And here if we actually changed something, I want to ask the user if he accidentally clicks back or somewhere else, if you really want to leave, or if you maybe forgot to click update server first.
* ***So this convenience method of keeping the user from accidentally navigating away.***
* How can we implement this? Let's go to the edit-server.
* component for this.
* Here.
* And let's add a new property below allowEdit.
* I'll add a changesSaved property, which is false by default.
* And which I want to change, whenever we click onUpdateServer.
* Here of course, we did update the server.
* So I want to call changesSaved, and set this to true here.



* Because now the changes were indeed saved.



* After the changes were saved here, I want to navigate away.
* So I will inject the router here, of type Router.

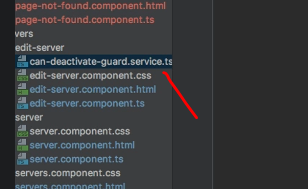


* Make sure to add the import at the top, as always, from the @angular/router package.
* And then here after we saved our changes, I want to navigate away.
* I want to go up one level to the last loaded server.
* And for this I will pass the relativeTo configuration, and want to navigate relativeTo the currently active route.

Text

Description automatically generated

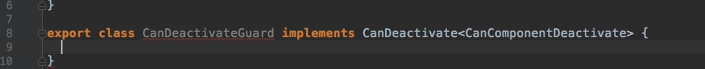
* So far so good.
* We're changing this changesSaved property here.
* Now let's make sure that whenever the user tries to accidentally navigate away, that we prevent him from doing so, or at least ask if he really wants to leave.
* Now we somehow need to execute this code in this component here, because we will need access to this changesSaved method, which informs us on whether this update button was clicked or not.
* *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 the code in our component and have a service.
* And that sounds like a complicated setup, but it isn't.
* It's actually really simple.
* In my edit-server folder here, I'll add a new file.



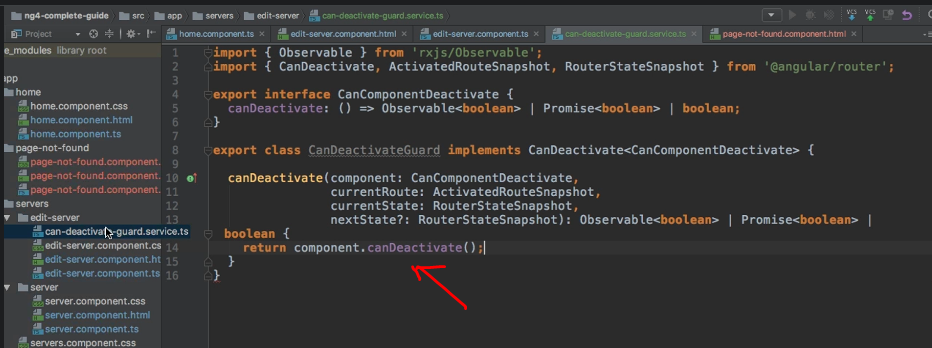
* I'll name it can-deactivate-guard.
* And as again, will also be a service as all guards are.
* And in here, I first of all now want to export an interface.
* An interface simply is a contract which can be imported by some our class.
* Let's say, which forces this class to provide some logic.
* So here I will name this CanComponentDeactivate.
* And this interface will require one thing from the component which implements it.
* This component should have a canDeactivate method.
* This method, and this is simply how I define the type of this canDeactivate method, since this is only an interface, it won't contain the actual logic.
* It will only contain information how it should look like.
* So this method should take no arguments, but in the end it should return an Observable, which therefore needs to be imported in this file here from rxjs/observable, which will resolve to a boolean in the end.
* Or a Promise, which will resolve to a boolean in the end.
* Or just a boolean.
* You might recognize this pattern here, from the canActivate guard.
* These guards share the same structure.



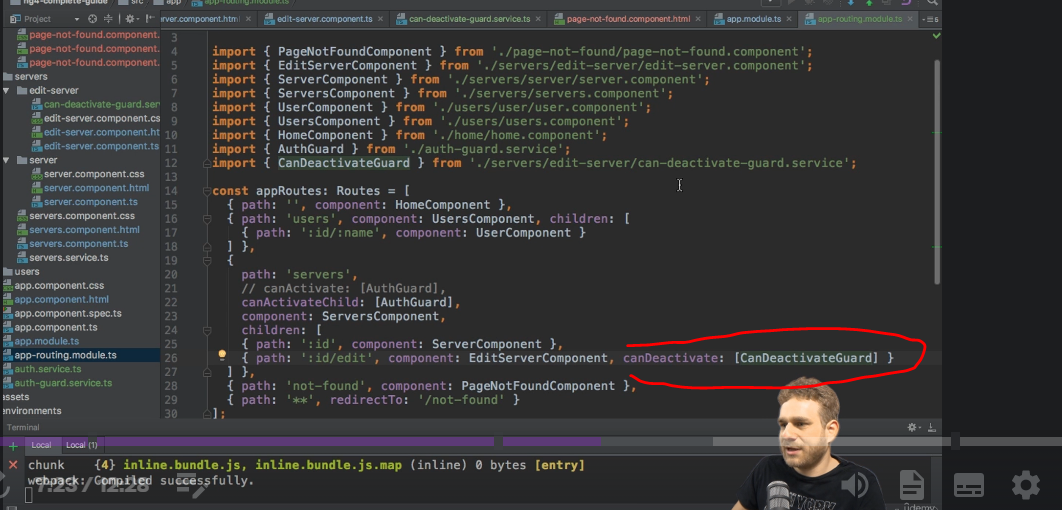
* So that is my interface.
* Nice to have it.
* But that alone won't do the trick.
* Now let's come to the important thing.
* The meat of this class, of the service here, I should say.
* The service itself.
* So I'll name it CanDeactivateGuard.
* And this guard will implement CanDeactivate.
* This is a interface provided by the @angular/router.
* So we need to import it from there.
* This actually is a generic type, and it will wrap our own interface.
* So it will wrap an interface which forces some component or some class to implement the CanDeactivate method.



* Might sound complicated, but this is the setup which will make sure that we later can easily connect a component to our CanDeactivateGuard here.
* So for now, let's just go with that piece of information.
* *And now this class here, this guard, will also need to have a CanDeactivate method.*
* ***This is the CanDeactivate method, which will be called by the @angular/router, once we try to leave a route.***
* Therefore, this will have the component, on which we're currently on as an argument.
* And this component needs to be of type CanComponentDeactivate, which means it needs to be a component, which has this interface here implemented.
* Therefore a component which has a CanDeactivate method.
* This will become super important in a second.
* We also will receive the currentRoute as an argument.
* So that is a ActivatedRouteSnapshot.
* Need to make sure to import this too.
* It will have the currentState, so that is a RouterStateSnapshot.
* Import this too.
* And the nextState, so where do you want to go? Because remember this will be called at the end, when we want to leave a route.
* So nextState is an optional argument though.
* This will also be a RouterStateSnapshot.
* And this will now also return an Observable, a Promise, or a boolean.
* So just like the CanActivateGuard.
* That is how the CanDeactivate method in our CanDeactivateGuard looks like.
* And here, I now want to call CanDeactivate on the component we are currently on.
* And this is why I need to implement this interface in this component, why I created this interface in the first place.
* *Because now, the @angular/router can execute CanDeactivate in our service, and can rely on the fact that the component we're currently on has the CanDeactivate method too.*
* *Because this is where we will actually implement the logic checking, whether we are allowed to leave or not.*
* Because we need this connection between our guard and component.
* So this is why we can safely call CanDeactivate here.



* And now, with that information, we can go back to our app-routing.
* module.
* *And on the edit path here, we want to add this Guard, so we can add canDeactivate as a property to this Route config.*
* It takes an array just like CanActivate.
* And here, we now will point to our CanDeactivateGuard.

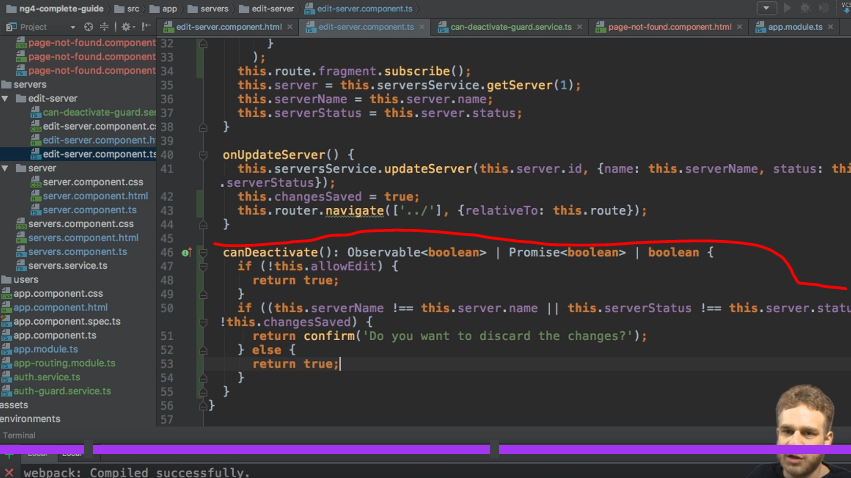


* Make sure to add this import, therefore.
* And now angular will run this guard, whenever we try to leave this path here, this component loaded at this path.
* For this to work, we of course need to go to our module, and provide our CanDeactivateGuard, just like before.

Text

Description automatically generated

* CanDeactivate.
* With this provided, make sure to add the import.
* We're almost there, but one important piece is missing.
* ***cancomponentDeactivate interface.***
* Remember that I told you that the CanDeactivateGuard will in the end call canDeactivate in our component? Well, for this to work on our edit-server.
* component, here, ***we need to implement our CAN component Deactivate interface.***
* Our own interface, we export it in our can-deactivate-guard.
* service file.
* This interface.
* This interface now forces us to implement the canDeactivate method in our component, which is import, because we try to call this method on our component here.
* So in the edit-server.
* component here, I will simply add a canDeactivate method now.
* We know how this method should look like because we defined it here.
* It should return an Observable, a Promise or a boolean.
* So let's make sure that it does.
* And now here, first of all, we need to import {Observable} from 'rxjs/Observable'.
* But with that added, with this import added, here, in canDeactivate, we now provide the actual logic deciding on whether we are allowed to leave or not.
* This logic will be run whenever the canDeactivateGuard is checked by the @angular/router.
* So here, I will quickly check if, we first of all allowed to edit this.
* Because if we're not allowed, like this, here I will return true anyways, because, "Hey we weren't allowed to edit it in the first place.
* So, sure, you may leave.
* " And otherwise, I will check if my serverName here is unlike the server.
* name we had at the beginning.
* ServerName here, this property, is simply bound via two-way binding to user inputs.
* So initially, it is set to the value we get from outside, from the server which was loaded.
* We bind it here.
* But as soon as we type, we will have some differences.
* So if we have differences on the serverName, or on the serverStatus, so here I ought to check if this is not the same as before server.
* status.
* So if one of the two was changed, and now I will wrap it in parentheses.



* So if one of the two was changed, and the changes were not saved, so changesSaved is false, in this case, I want to return a confirm dialogue, where I ask the user, "Do you want to discard the changes?" Otherwise, I will return true, because otherwise, either nothing was changed, or it was changed, but changesSaved is set to true.
* So we did save it, indeed.
* With this in place, let's go back.
* Let's go back to our servers.
* And first of all, log in, maybe.
* Let's go to the Devserver, which is the one we are allowed to edit.
* And here, if I now type something there and I didn't click update server, and I click somewhere else, we are asked if we want to leave.
* And if I click cancel, we stay on this page.

Graphical user interface, application

Description automatically generated

* We may now save our changes, and then we are navigated away without getting any issues at all.
* And the fact that we updated the wrong server, simply happens because here, we still are always loading the server with the id 1.
* So we should update this to get our real id.
* So maybe here, access our route.
* snapshot.
* params['id'].
* And pass this as an id, and thereafter, of course, also subscribe to route params to update the id if params change.
* That is something you can do as a exercise.
* And of course, make this of type number by converting it.
* But more importantly here, everything's working as it should.
* We are only allowed to change this if we did click update.
* Otherwise we are asked.
* Or of course, if here once we are asked, we click OK, we are allowed to discard our changes.
* So that is true or that is possible too.
* But now, with canDeactivate, we have this extra protection in place.
* So with this, we learned a lot about routing in this module.
* Now there is one more guard I want to dive into and we will do so in the next lecture.