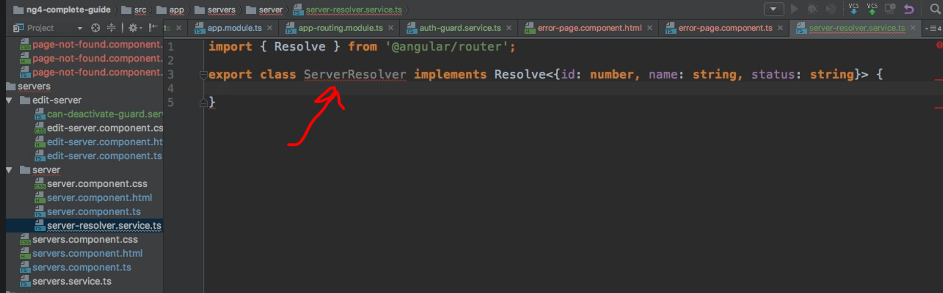
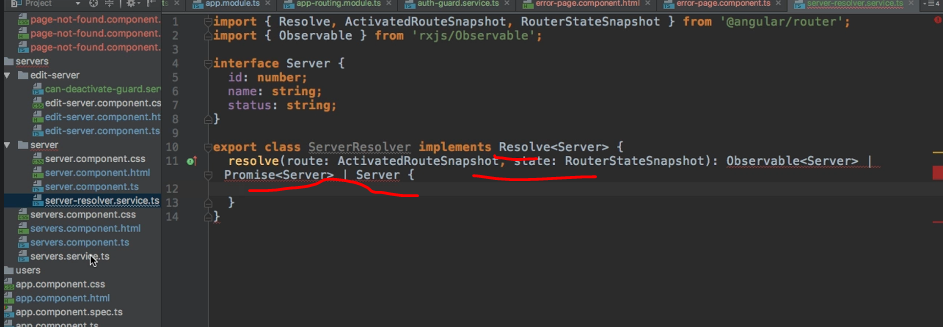
**151. Resolving Dynamic Data with the resolve Guard**

* -: In the last lecture, we learned how to pass static data.
* ***Now let's say we have some dynamic data we have to fetch before a route can be displayed or can be rendered.***
* **Use Case:**
* For example, here on the servers, let's say these servers already have been loaded.
* But once we click a server here, and maybe I should log in to demonstrate this.
* I want to load the individual server from some backend.
* So we are going to simulate that this takes a couple of milliseconds or seconds.
* So how could this work? If we have such a use case, we need a resolver.
* *There's always a service just like CanActivate or CanDeactivate, which will allow us to run some code before a route is rendered.*
* **Resolver Vs canActivate:**
* Now the difference to CanActivate is that the resolver will not decide whether this route should be rendered or not, whether the component should be loaded or not.
* *The resolver will always render the component in the end, but it will do some preloading, you could say.*
* It will fetch some data the component will then need later on.
* Of course, the alternative is to render the component or the target page instantly, and in the OnInit method of this page you could then fetch the data and display some spinner whilst you're doing so.
* So that is an alternative.
* But if you want to load it before actually displaying the route this is how you would add such a resolver.
* So let's say for the single server here, I will add it in the Server, File, my server-resolver.
* And again, this will be technically just be a service.
* So I'll export my class here which I'll name ServerResolver.



* And now this has to implement the resolve interface provided by @angular/router.
* Resolve is a generic type and it should wrap whichever item or data field you will get here, will fetch here in the end.
* So we will fetch a server here and therefore we should define the type here.
* Now, you could outsource this into an interface or a model in general.
* So we will know that a server will have an id which is of type number, that it will have a name which is a string, and a status, which is a string.
* So this is simply a type definition of, well, the thing this resolver will give us in the end to what it will resolve in the end.
* **Resolve Interface:**
* Now the resolve interface requires us to implement the resolve method.
* *And this resolve method here now takes two arguments the route, so the activated route snapshot which you already know, which you need to import from at angular core, and it will also provide us the state snapshot.*
* So the RouterStateSnapshot, which also needs to be imported from @angular/router.
* These are the two information pieces the resolve method gets by angular, and in the end, this then also has to return either an observable, which you need to import from rxjs/Observable.
* This observable will then also return this type here.
* So we can copy this, but since we're creating a lot of overheads here, we can also quickly define an interface here, and it would of course be better to outsource this in its own file.
* But let's quickly do it here.
* So the interface for the server which has an ID of type number, and then the name which is a string, and the status, which is a string.
* And now this allows us to simply use server here for as a shortcut for this type.
* So the observable should in the end, give us back such a server, or we get a promise which in the end will return such a server, or well, just such a server synchronously.
* So this is what we will get back here and resolve.
* So now we need to implement logic to get this back.
* Now we have our server's service here.

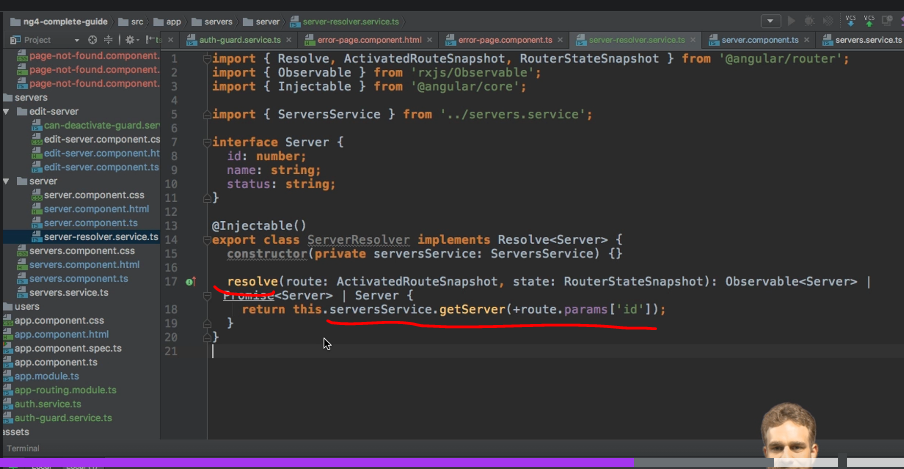


* And there we have the get server method which will give us back a server.

Text

Description automatically generated

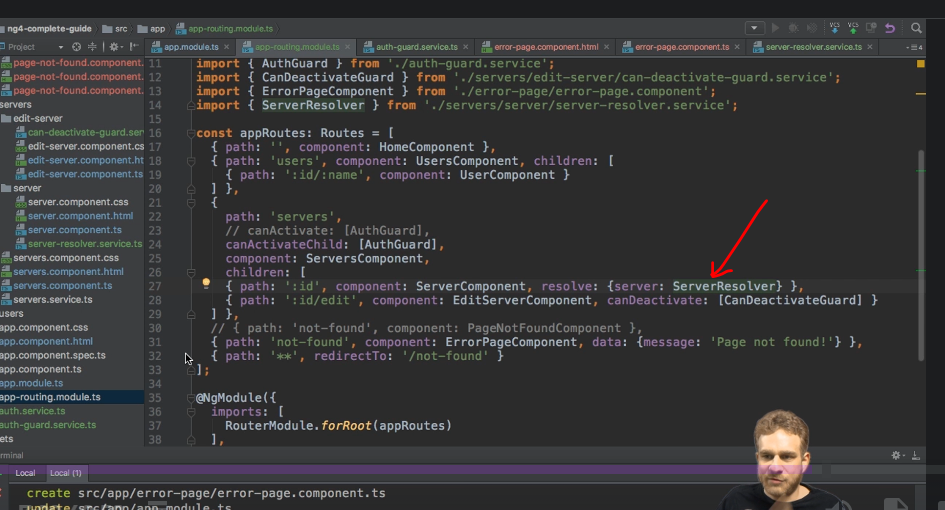
* Now this gritty is some synchronous code, it will run instantly here.
* So there is nothing wrong with this code, we will resolve this instantly.
* And as you saw in your resolver, that's fine.
* One return possibility is to instantly return the data.
* So the easiest thing we can do here is reach out to our server service and forward as we need to inject it, so we should provide our server serversService like this.
* That's a real hard word for me.
* Servers ServersService like this.
* And of course, if you want to inject a service into another service, you have to add, @Injectable to it, make sure to import this from at @angular/core, also, make sure to import your own ServersService.
* And with this, the easiest way to use this resolve function is to reach out to your service, and there we call getServer.
* And now we need to know the idea of the server we have to fetch.
* Because that is what we do in this server component here in the end, right? There in on in it, we're getting this server right now.
* So now we want to outsource this because let's say we want to do it before it loads.
* The good thing is, we do get the route here.
* It only needs this snapshot, but this service here will actually run on each time we re-render the route.
* So the snapshot is all we need.
* Unlike the component itself, again, this is executed each time.
* So no need to set up an observable or something like that.
* So here we can access our route, and the route params, and then the id, and make sure to cast it with a plus sign in front of it to an integer.



* This is the easiest way of using this resolver which will do the loading of our data in advance.
* And now this would also work if this were to return an observable or a promise, so with asynchronous code, for example, an HTP request.
* So this is our resolver.
* With this resolver in place, we of course now have to add it.



* So first thing is in app module, we should add it to our providers, our server resolver like this.
* Make sure to also add the import at the top.
* And with that added now, we want to add it to our routing module.
* **Different Approach for resolvers:**
* Here for the route where we want to use it, so let's say for this component here, we add another property, the resolve property.
* This takes a JavaScript object, and here we map all the resolvers.
* This is different to the AuthGuards.
* There we used arrays, but for resolve, a different approach is taken, and you will see why in a second.
* So here we have key value pairs of the resolvers we want to use.
* So for the server, and this name of the property is totally up to you.

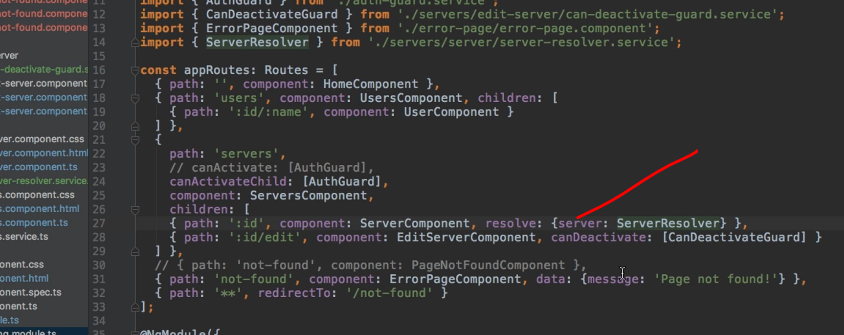


* We will use our ServerResolver.
* Like this.
* Make sure to import it here at the top.
* This will now map the data.
* This resolver gives us back, and remember, it gives us back some data with this resolve method it had to implement.

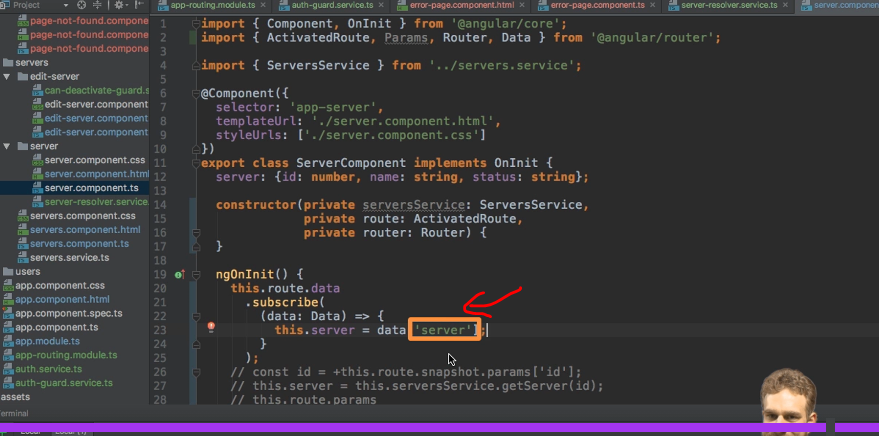
Text

Description automatically generated

* This method will be called by angular when this route is loaded.
* So the data it gives us back here will be stored in this server object we will then have available, or in this server property we will have available in this to be loaded component.



* So in this to be loaded component, in our server component, right now we get the server like this, using the params.
* Well, I will comment this out, because now we use a resolver for this.
* That's the goal of the resolver.
* And it makes even more sense if this is some asynchronous task, of course.
* So here we can easily get our server by binding the data observable.
* So just like static data, which you passed with this data object here, with this data property you could add to a route, the data returned by your resolver will also go into this data baggage and this data object you have in your to be loaded component.
* So here we can listen to any changes, and I'm setting up an observable here, or I'm using the observable here I should say, because the server can change while we already are on the page, as we have this side menu.
* This is why we also had to set up our route patterns here dynamically.
* And here we know we will get back our data of type data as learned before, make sure to import data from angular router.
* And now we know that we can simply assign our server to server property in this component by binding to data server.



* And now this name here is important.
* This server here, this name here, of course has to match the name you use here when you assign your resolver to some property.

Text

Description automatically generated

* So if you change the name here, you have to change it when you retrieve the data.
* Well, with this in place, if we save this with angular server still running, if we log in and go to Servers and we click on a server, you see this still works as intended, it still loads the server like before but now not using the params here in the server component but instead using a resolver before this, again, of course especially important when using asynchronous data.