**ANGULAR 8 – Part II**

**#SECTION 8 : COURSE PROECT - DIRECTIVES**

**#### 102. BUILDING AND USING A DROPDOWN DIRECTIVE**

Welcome back to the recipe book.

We already use ngIf and ngFor in the sections, so we're already using directives but now I want to enhance this project by adding our own directive which allows us to open the drop downs which don't work as of now if you click them.

They don't work because we didn't import Bootstrap's Javascript code and I don't want to import

this because Angular should be the only thing interacting with my DOM but now with the directive knowledge, we can create our own directive which will manipulate these buttons.

I'll add the directive in the “**shared folder**” and I will name it “**dropdown.directive.ts**”, so the file is named dropdown.directive.ts. Of course you could use the CLI to create it and you could even point to the shared folder in the command to automatically generate it in this shared folder.

I'll do it manually though, always a good practice and here, the name should be “**DropdownDirective**” like this and of course,

$ **export class DropdownDirective{…………………………..}**

we need to add a decorator to turn this into a directive. So it's the **@directive decorator** which needs to be imported from @angular/core and here

**$ @Directive ( { ……………………….} )**

there is one thing I need to set up on the selector, I want to define how I can add my directive and I will use an attribute selector, therefore I'll enclose the name in square brackets. The name now will be “**appDropdown**” to have a unique name which doesn't accidentally interfere with an officially existing attribute name or anything like that.

**$ @Directive ( {**

**$ selector: ‘[appDropdown]’**

**$ } )**

Now what should this directive do? I want to add some functionality to it which allows us to add a certain CSS class to the element it sits on once it is clicked, so once this element the directive sits on is clicked and removes the class once we click again let's say. So this is your challenge, try implementing such a method which basically listens to clicks and then toggles some property which determines whether some class is attached or not and also try implementing some code which dynamically attaches a CSS class or not.

Before you actually start though, let's find out which class we need to attach. If you go to the recipe detail component here, if you attach the open CSS class to the div which holds the button group and you go to your application and select a recipe ( **<div class=”btn-group open”>** ), you see now it looks like the dropdown was opened. So it's this “**open**” CSS class, the same for the usage in the header we need to attach. So that's your challenge, build a dropdown directive which attaches this class whenever we click on it or removes it once we click a second time. Let's build this directive together now.

So as explained, I want to toggle it upon clicking this. So to listen to a click, I should add @HostListener which is to be imported from @angular/core and I want to listen to a click event.

$ **@HostListener(‘click’)**

Then I want to execute toggle open which looks like or sounds like a fitting name because that is what I want to do, I want to toggle whether this should be opened or not.

$ **@HostListener(‘click’) toggleOpen( ){……………………}**

Well then, I will add an isOpen directive which I'll set to false initially( **isOpen=false;** ) and in my @HostListener, I'll simply set open or isOpen to what it is currently not ( **this.isOpen = !this.isOpen;** ). So if isOpen was true, it will now be set to false and the other way around.

So with that, I'm toggling that property. Now to dynamically attach or detach a CSS class depending on this, all I have to do is, I'll add @HostBinding which needs to be imported from @angular/core and allows us to bind to properties of the element the directive is placed on and here, I want to bind to the class property of that component, of that element, class simply is an array of all the classes and here, it's all about this “open” class. As you learned, that's the CSS class we need to attach dynamically or detach and now the rest will be handled by Angular since I bind to isOpen, this will not be attached initially, whenever isOpen switches to true, it will be attached and whenever it switches to false, it will be removed and that's our entire directive.

$ **export class DropdownDirective{**

**$ @HostBinding(‘class.open’) isOpen= false;**

**$**

**$ @HostListener(‘click’) toggleOpen(){**

**$ this.isOpen = ! this.isOpen;**

**$ }**

**$ }**

Now to be able to use it, I need to go to app module and add it here, add the dropdown directive and add

the import pointing to the shared folder and with that imported now, I can use it.

**$ @NgModule({**

**$ …………………….**

**$ …………………..**

**$ DropdownDirective**

**$ })**

I can use it in the recipe.detail.component, here I have to use it on the div or on the element where open should be attached to in the end, so in this case the div with the btn-group class and here, I'll add my own “**appDropdown”** directive, without square brackets, without assigning any value because I don't need to

**$ <div class=”btn-group” appDropdown>**

configure anything on this directive and now in the header, here the place where I need to add it is this list item with the class of dropdown, here

**$ <li class=”dropdown” appDropdown>**

this is what will actually need the open class to show something. With that in place and and ng serve still running, let's save this and view this in our application, see if this works the way it should.

Let's test it in the header, looks pretty good, we close it by clicking again and here on a recipe, that looks awesome too.

So that's working exactly the way I want it to work and with that in place, that already is everything I wanted to build, so a relatively short recipe book section here. Now we can move on to services, a very powerful tool and then we will add services in this app here to greatly enhance it and fix some things which we had to do kind of complicated thus far. We'll also add some new features.

**#### 103. CLOSING THE DROP DOWN FROM ANYWHERE**

If you want that a dropdown can also be closed by a click anywhere outside ( which also means that a click on one dropdown closes any other one, btw.) replace the code of the “dropdown.directive.ts” by this one (placing the listener not on the dropdown, but on the document);

**$ import { Directive, ElementRef, HostBiding, Hostlistener } from ‘@angular/core’**

**$ @Directive({**

**$ selector: ‘[appDropdown]’**

**$ })**

**$ export class DropdownDirective{**

**$ @HostBinding(‘class open’) isOpen = false;**

**$ @HostListener(‘document:click’,[‘$event’]) toggleOpen(event: Event){**

**$ this.isOpen = this.elRef.nativeElement.contains(event.target) ? !this.isOpen : false;**

**$ constructor(private elRef: ElementREf){ }**

**#SECTION 9 : USING SERVICES AND DEPENDENCY INJECTION**

**#### 104. MODULE INTRODUCTION**

Welcome to a new module in this course. This module is about services and dependency injection,

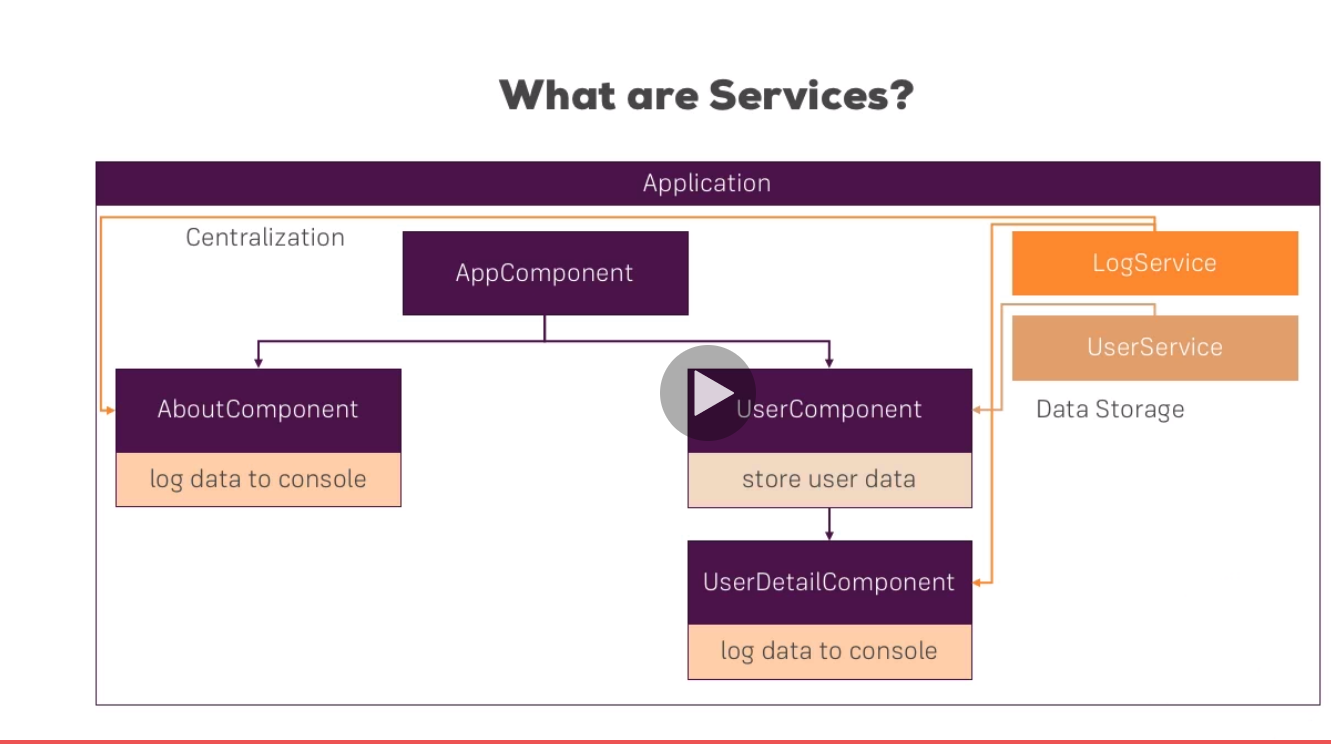
now what is that?

Let's start with services, what are services?

Let's consider this application, we get an AppComponent, an AboutComponent, a UserComponent and a UserDetailComponent.

Now we might have some methods in those components, let's say we want to log some data to the console in the AboutComponent ( log data to console) and in the UserDetailComponent ( log data to console) and somehow this log statement is pretty similar, so pretty much the same text, so we're basically repeating ourselves here, we're duplicating code in two different components.

Let's say in the UserComponent ( store user data), we also want to access some data, some array of users maybe and maybe we also don't know if we will later use that in some other part of our application. Both use cases, duplication of code and data storage, providing data are typical use cases for a service because a service is basically just another piece in your Angular app, another class you can add which acts as a central repository, as a central business unit you could say, something where you can store, where you can centralize your code in. So we could create a log service for example to centralize our log statement, we could create a user service to manage the data storage here. Let's see this in a real app so that it becomes more clear though.



**#### 105. WHY WOULD YOU NEED SERVICES**

Here's an example app ( folder name services-start-mysolution ) which you can find attached to this lecture.

Here we can create some accounts, you'll see we started with three accounts here at the bottom,

we can switch these status of these accounts and you can see that whenever I switch it or log something

to the console, you can also see that this gets logged ( in console )

if I create a new account and overall, what we get here is the logging and the data storage,

so in general tasks which might be centralized. We also communicate between these components, that's another area where services can be really helpful because let's have a look at our code. I got “**3 components**”, the “**app component**”, then the “**account**” and the “**new account**” component. The app component is where it all comes together.

There in the template, I embed the **<app-new-account>** and the **<app-account>** component and I use property and event binding to pass data around, for example I loop through all accounts which are right now stored in the app component here (app.compoment.ts) and I pass them via property binding to the app account component here(app.account.component.html) and there on the other hand, I have an event which gets fired when I click one of the three change status buttons, which is submitted here

**$ Output( ) statusChnaged = new EventEmitter<{id: number, newStatus: string}>();**

**$ onSEtTo…….{……this.statusChnaged.emit({id: this.id, newStatus: status})**

which will be catched by my app component where I then handle it again

$ (statusChnaged) = “onStatusChnaged($event)”

So I do have event binding to be informed in my app component whenever something changes in the account component, pretty much the same happens in there, new account component. There, I can also listen to a click when I create a new account and on such a click, I emit some data to the app component. So this is all fine but we already got quite a complex set up here to pass data around and that even though that it's still relatively simple, it's not a complex app here after all.

Now in this lecture, we're going to see how services can help us with that and what we have to watch out for. So let's dive into it and build our first service in the next lecture.

**#### 106. CREATING A LOGGING SERVICE**

As mentioned at the end of the last lecture, I want to start by building our first service here, a service

which takes over this logging task here,

$ **( console.log(‘A server status changed, new status: ‘ + accountStatus);**

so a service where we can log data.

Now how do we create a new service, how does this service look like in Angular? Creating it is super simple. First we have to choose a fitting place now since this service will do some generic task, I will place it in the app folder, you could also create a shared subfolder there, I will again place it right in the app main folder and I will simply name it logging and the file name should be “**logging.service.ts”** to follow this convention of being pretty clear about what's in a file.

So here, this file will hold a service so that's why I'll name it like this. There, I will export a TypeScript class and I will name this class “LoggingService” to again to be explicit about what this class does.

**$ export class LoggingService**

Now a component becomes a component because we attached this @component decorator, a directive becomes a directive because we attach @directive, you could think that we now simply add @service right but we don't do that. A service is just the normal TypeScript class, like this, this can already be used as a service.

So here, we could create a helper method, “logToConsole” and

$ **logToConsole(message )**

and simply pass the “message” but since we know that we simply want to log the change of account status here, I will simply say “log status change()”

$ **logToConsole(message ) //replace with**

**$ logStatusChnage( status: string )**

like this and take the new status which will be a string as an argument. Here, I can then simply copy the code I do have in one of my components right now,

**( console.log(‘A server status changed, new status: ‘ + accountStatus);**

this console log code here and copy it into my logging service here, of course account status now is just status here, this argument we're getting.

**$ export class LoggingService {**

**$ logStatusChange(status: string) {**

**$ console.log(‘A server status changed, new status: ‘ + accountStatus)**

**$ //change accountStatus to status**

So with that, remove the semi-colon, with that I centralized this code, now we need to use the service in our other components, so everywhere where we console log. Now you could think that we simply use it inside “new-account.component” by “importing” it at the top of our file, so here we could import the “loggingService” from and now let's move up to the main folder and then here from the “logging.service.ts” file

$ **import { LoggingService } from ‘../logging.service’;**

and then we could simply create our service like this,

$ **const service = new LoggingService( );**

**$ service.logStatusChange(accountStatus)**

new logging service and on our service here, we could call log status change and pass the “new account status”. And actually, if I get rid( comment out old console.log) of this code here, this will work, let it recompile and if I now create a new service here, you see I do log it to the console.

So this would work and still, this is wrong, this is not and this is super important, this is not how you

use a service in Angular. The reason for this will become more clear throughout this module but generally, Angular provides a much better way of getting access to your services and therefore, you

should not create the instances manually. So let's get rid of that import and of that instantiation and let's learn in the next video which tool Angular offers us to get access to our services.

**#### 107. INJECTING THE LOGGIN SERVICE INTO COMPONENTS**

In the last lecture, we created our service and I told you to not instantiate it on your own. I told you that Angular offers some great tool which will give us access to our services, it's Angular's dependency injector.

**What is a dependency injector?**

Well a dependency is something a class of ours will depend on, for example the “**new account**” component depends on the “**loggingService**” because we want to call a method in that service and the dependency injector simply injects this dependency, injects an “**instance of this class**” into our component automatically.

All we need to do is we need to inform Angular that we require such an instance. So how do we inform Angular that we require such an instance? We add a constructor to the class, to the component in this case where we want to use our service.

So there, I can bind it to a property by using this TypeScript shortcut of adding an “**accessor**” in front

of the name, of the argument to instantly create a property with the same name and bind the value to it.

So here, I will name this “**loggingService**”,

$ **constructor( private loggingService: LoggingService){ }**

this name is totally up to you. Now this is the important part here though, add a type assignment here,

this is not optional. You need to set the “**type**” and the type has to be the class you want to get injected,

“**LoggingService**” in this case, make sure to also add the “**import**” at the top.

Now let me close the body of this constructor and this simple task here informs Angular that we will

need an instance of this loggingService. Correctly you might ask, well how do we inform Angular, why does this matter if we write this in the constructor?

Think about who gives us the instance of this account component, this new account component here.

This is a TypeScript class in the end so somewhere, this needs to get instantiated so that something happens in our app.

**Well who is responsible for creating our components?**

Angular is of course because we are placing selectors in our templates and when Angular comes across

these selectors, it gives us instances of our components. Now since Angular is responsible for instantiating our components, Angular will need to construct them correctly.

So if we define in the constructor that we require some argument, Angular will recognize this and now

it tries to give us that “**argument**”, it tries to give us this type in this case. So it knows we want an instance of the loggingService class because we defined a type, this is why this is important.

Now that is almost enough but not quite, now Angular knows what we want but it doesn't know how to give us such an instance. We need to do one additional step, we need to provide a service. “**Provide**” simply means we tell Angular how to create it and that sounds very complicated and it is very simple,

all we have to do is add one extra property to the @component decorator, the “providers property” here.

**$ @Component({**

**$ selector: ‘app-new-account’,**

**$ ……………………………………………..**

**$ providers: [ LoggingService ]**

This also takes an array like other properties we added before and here we again just have to specify the type of what we want to be able to get provided you could say, so “**LoggingService**” again.

Now with that, Angular when analyzing the component, recognizes that it should be able to give us such

a loggingService and it will set itself up to be able to do so and when it then actually builds the component, constructs it, it sees that we want to have such an instance and it will know how to give us such an instance and now we can simply in our component, anywhere in this component, access our loggingService property which is created automatically since I use this TypeScript shortcut here and call log status change.

$ **this.loggingService.logStatusChnage(accountStatus)**

So now I'm not creating that instance manually, Angular does it for us and why is this better than creating it manually? You will see some other advantages later but this basically let's you stay in the Angular ecosystem and Angular knows how your app works. Again, some other advantages will come up later.

So with this, we now have the same code as before, so if we have a look at our application again and create a new account, we still see the log here depending on which status we chose here, so this still works but now, we're injecting the service. Now let's also do the same by copying that in the account.component here.

So here, I'll also add the constructor and of course here we also need to add the “**import**” because TypeScript needs to know where this loggingService comes from, not Angular, TypeScript and I will also add a “**provider**” of course because for this component too, I need to inform Angular how to create

this loggingService. So if we do this and then go back to the application here, reload, create a new service again, this works, now let's change the status and this also still works.

Now of course because I didn't replace this here,

**$ // console.log(‘A server status changed, new status: ‘ + status ) in the account.component**

**$ //change this to**

**$ this.loggingService.logStatusChnage(status)**

so let's call the loggingService now, and call log status change and pass the status. Now if we save this and try to change this, it still works. So now we get this same logging functionality outsourced, centralized in a service and therefore, our code here is a bit leaner and I guess you can imagine how in bigger applications, where you have duplicate code, a service can really help you to get more “dry”, to don't repeat yourself all the time but instead to really cleverly outsource your code into a service and have it there.

**#### 108. CREATING A DATA SERVICE**

We saw how to use that loggingService in the last lecture, now let me show you another typical use case for a service, to store and manage our data.

So for example our accounts, right now I store them here in the app component and then we have this chain of property and event binding to get data to the app component so that we there can update our accounts. Let's create a service for that, the account service maybe.

$ **account.service.ts** in the app folder

So I will create this and I will **export** my **AccountsService** class here,

$ **export class AccountsService { …………………………. }**

just like that and here I will now copy in my accounts

  accounts = [

    {

      name: 'Master Account',

      status: 'active'

    },

    {

      name: 'Testaccount',

      status: 'inactive'

    },

    {

      name: 'Hidden Account',

      status: 'unknown'

    }

  ];

, so I will remove them from my app component and paste them into my accounts.service and I will add an addAccount method here where I simply expect to get an account name and status and I will add an updateStatus method where I expect to get the ID of the account I want to update and then the new status here.

$ **addAccount( name: string, status: string )**

**$ updateStatus( id: number, status: string )**

Now the logic for these two methods is basically the same as in the app component, for a new account

or in the add account, I simply want to **push a new account onto my array** of accounts and here, I will simply quickly **create a new object** where I assign the **name** and the **status**, so the data we get passed into this method and for the status changed, well here I basically want to also just access my account with that ID here and set the status to the new status we get passed as an argument too.

$ **addAccount( name: string, status: string ){**

**$ this.accounts.push( { name: name, status: status} )**

**$ updateStatus( id: number, status: string ){**

**$ this.account[id].status = status; }**

So with that, this service looks all right, we got our accounts there and we got our methods to change something there, of course the app is broken though because for one, I don't need that code in the app component anymore.

  // this is added into the account.service.ts

  // onAccountAdded(newAccount: {name: string, status: string}) {

  //   this.accounts.push(newAccount);

  // }

  // onStatusChanged(updateInfo: {id: number, newStatus: string}) {

  //   this.accounts[updateInfo.id].status = updateInfo.newStatus;

  // }

If we have a look at the app component template, we still loop through all the accounts here,

 \*ngFor="let acc of accounts; let i = index"

so I should still have my accounts here in the app component but we remove them from here, they're now in the service.

So we should maybe re-add them here but now, this should simply be an array of accounts, so each account object has a name which is a string and a status which is a string, so I'm just defining the type here and we want to have an array of such objects, initially it should be an empty array, this is what I'm saying here.

export class AppComponent {

  accounts: { name: string, status: string }[] = [];

Now since we have our account service, we should inject it here as we learned it. So let's add our accounts service here, of type accounts service, the type is super important to inform Angular

about what we need here.

constructor( private accountsService: AccountsService )

Let's close this body of the constructor and make sure to add the import at the top pointing to your account service file. Now what did we learn? We also need to **add a provider**. So here at the **providers array**, we should add accounts service so that Angular knows how to create such an account service.

**$ @Component({**

**$ selector: ‘app-root’,**

**$ ……………..**

**$ ……………**

**$ providers: [AccountsServices]**

**$ })**

With that, we can now add for example onInit, this **implements OnInit** lifecycle hook as most initialization should not be done in the constructor but instead here and simply set this accounts, accounts equal to this account service which is available as a property because I use this shortcut with the accessor in front of the argument name and here, I can access my accounts field.

$  ngOnInit(){

    this.accounts = this.accountsService.accounts;

  }

Just as a little note, since accounts here is an array, it is a reference type, this is normal Javascript behavior. So this is a reference type, so by setting it equal here, we're actually getting access to the exact same array as stored in the service, just as a side note.

So with that, we do have access to the accounts and if we save this and have a look at our application,

this still seems to work. These buttons will not work and will actually give us errors because I remove

the event listeners but in general, this looks all right, we can see the three accounts which are now stored in our accounts service.

So with that, let's go back to our application and here, I now want to update the account component and the new account component, let's start with the new account. Here we no longer need to **emit this event** here

 onCreateAccount(accountName: string, accountStatus: string) {

    this.accountAdded.emit({

      name: accountName,

      status: accountStatus

    });

this.accountsService.addAccount(accountName, accountStatus);

because we're not listening to it anyways. So we can get rid of the output here, get rid of the imports up here, of eventEmitter and output, instead we should inject our service, our accounts service of type accounts service. And what did we learn? We should provide it, right, so let's add account service, account service and make sure to add the import at the top.

 @Output() accountAdded = new EventEmitter<{name: string, status: string}>();

  constructor(private loggingService: LoggingService,

              private accountsService: AccountsService ){};

Now with this, in here I can call accounts service, add account and pass the account name and the account status as this is what this method expects to get as arguments. So with that, let's go to our account component now, let's also get rid of the output here. ( remove from imports )

We still need the inputs because we're still receiving that data from outside and we can get rid of this event calling here, we removed the output anyways, get rid of the imports we don't need anymore and now here, I also want to get access to the accounts service.

// remove this.statusChanged.emit({id: this.id, newStatus: status});

So let's inject it here too, maybe in a new line to make it more readable, the accounts service of type accounts service, like this, make sure to add the import at the top and let's add it to the providers array here.

  constructor(private loggingService: LoggingService,

              private accountsService: AccountsService ){};

With that, I can now call it here, account service, call update status, pass my ID and the new status, again this is what this method expects. If we now save this and go back to our application, it looks all right but we quickly realized that if we click on add account, we don't get an error, we get the log but I don't see the account. If I click these buttons, I also see the log but I don't see the change here,

so something is not working correctly, it seems like it doesn't, right. Well let's have a look at what's going wrong here in the next lecture.

**#### 109. CREATING A DATA SERVICE**

In the last lecture, we set up our account service and hooked it up on our app and still, our app seems

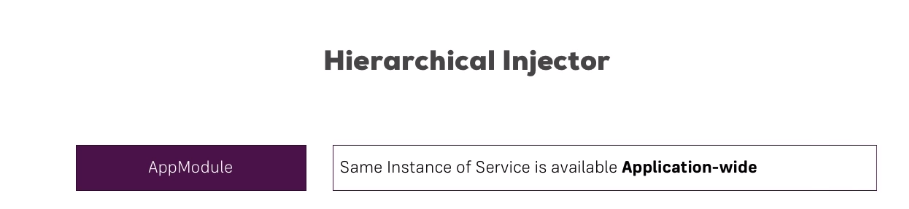
to be broken. If I click these buttons, the status has not changed, so somehow our code is not working.

Actually it's working correctly, we're just using services wrong here, in this specific case, we're not using the service in the way we should use it in our app.

What's going wrong?

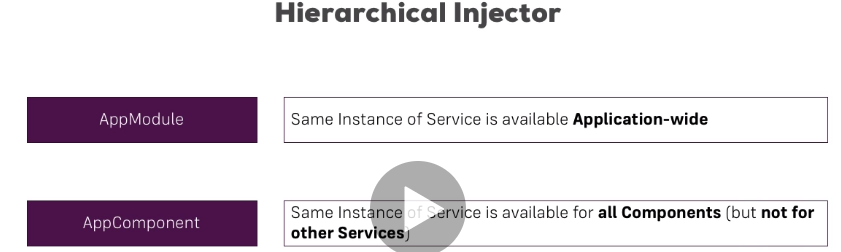
The Angular dependency injector actually is a hierarchical injector, that means that if we provide a service in some place of our app, let's say on one component, the Angular framework knows how to create an instance of that service for this component and important, all its child components and actually this component and all its child components and the child components of the child components will receive the same instance of the service.

There are other places where we can provide a service too, the highest possible level is the app module.



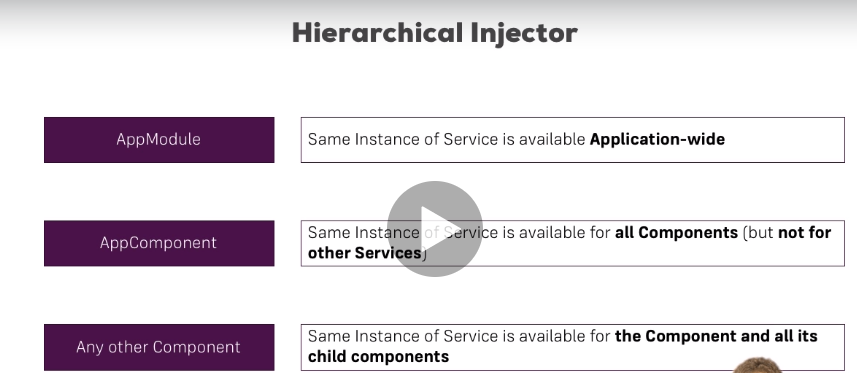
we haven't done this yet but we will see this soon. If we provide a service there, the app module does have a providers array, the same instance and that's important, the same instance of the class, of the service is available in our whole app, in all components, in all directives, in all other services where we maybe inject the service. Yes, we can inject services into services, I'll come back to this.

The next level for example would be the app component.



there the app component and all its child components do have the same instance of the service and this is true for any component, so even if we have a child of the app component, if we provide it on that child, all the children of this child will have the same instance and the child itself but not the app component. The instances don't propagate up, they only go down that tree of components.

The lowest level therefore is a single component with no child components.



If we provide a service there, this component will have its own instance of this service and well it

doesn't have any child components, so this instance will only be available for this component and this will actually even overwrite if we were to provide the same service on a higher level and that's exactly what we're doing. So let's take a closer look in the next lecture.

**#### 110. HOW MANY INSTANCES OF SERVICES IT SHOULD BE**

In the last lecture, we learned about that hierarchical injector and actually, this injection behavior leads to the behavior we see here. Now that's not a bug, it's intended, you might have an application where you want to have many different instances of the same service, where you absolutely don't want to have the same instance but sometimes, you want to have the same instance and we definitely do because right now for our account service, we have three instances.

The first one gets created here in app component, we provide it here, app component receives its own service or it's own instance of the service I should say as do all the child components. Now you might say fine because new account and account are child components, right but there we provide it again (**in the new.account.component.ts**) and hence we overwrite the instance we would get from the app component.

Now new account and accounts component have their own instances, so these are the other two instances of this service class. And therefore if we add a new service, we call add account, there we push it on this array of accounts and this actually works,

this.accountsService.addAccount(accountName, accountStatus);

but it's a totally different instance of the service than the one we use here in app component, the one through which we loop. How can we fix it then here for our app? Simple, we just remove it from the providers array.

**$ //remove this from new-account.component.ts**

**$ // remove AccountService ………………. Providers[LoggingService, AccountService]**

Don't remove it from the constructor, we of course do need it here because that tells Angular that we want an instance, the providers array basically tells which instance so there we need it to remove it

and the same for account component. Leave it in the constructor, leave the import at the top but remove it from the providers array.

**$ //remove this from account.component.ts**

**$ // remove AccountService ………………. Providers[LoggingService, AccountService]**

With that little change, if we go back to the application after saving it, let's add another account. Now you'll see it at the bottom, now you can change the status again. So this is the difference, different instances of the service. Now we're using one and the same, before we weren't. Both

may be the behavior you want in your app but make sure that you get the correct one.

**#### 111. INJECTING SERVICES INTO SERVICES**

In the last lecture we learned about this hierarchical injector and that we can choose between one instance or multiple instances of the same service class. I mentioned at the highest possible level.

It's actually not the app component. So let's remove it here.

$ **//remove** **providers: [AccountServices]**

This provider's array leave it in the constructor and leave the import but the highest possible level

is **in the app module**. Here we also have a provider's array added

**$ providers: [ ]**

**$ bootstrap: [AccountComponent ]**

. If you don't have it and here we can also provide the AccountsService. Make sure to also add to import at the top here.

$ **import { AccountsService } from ‘./accounts.service’;**

Then with that little addition heater we're making sure that our whole application everything in our application receives the same instance of the service unless it overrides it as we did before.

Now with that we can now even inject a service into another service because that's not possible by providing it on a component level for that we need to provide it in the app module so let's say we want to log something if we call actually in the account service. So it would have to provide a logging service in my app module

$ **providers: [AccountsService, LoggingService]**

now and therefore I will simply comment out providers here( **inside new-account.component.ts** ) and in my app component decorators and I don't want to call logging service here anymore so will comment this out (**inside account.component.ts**) and I will actually leave the injection here so that you have it for reference if you download the code.

$ **//comment** **this.loggingService.logStatusChange(status)**

I'll also comment it out here in you account component instead. Let's say I want to use this logging service and the account service and I want to call the logStatusChange method there because this isn't the end the place where the status changes even because we added a new account or because we called update status so to use the service there (**inside accounts.service.ts**).

Generally we start in the same way we add a constructor and we checked our logging service in this case

set the type import.

$ **constructor ( private loggingService: LoggingService){ }** ,

This type at the top make sure to add to import and we provided it an app module so you could think that we are set to go that we can now go to add account called logging service log the status change pass to status here and do the same here for update status( ).

$ **this.loggingService.logStatusChange(status);**

You could think this is how we do it. And it's not a logical to think so but let's see if this works if we load our app we already get an **error cannot resolve all parameters for account service.**

Something's not working.

The reason is if you inject a service into something this something needs to have some metadata attached

to it. Now a component has some metadata because we have @component a directive has some metadata because we have @directive account or excuse me a service doesn't have any metadata. It's just how Angular works that we need to attach some metadata. And of course not any metadata. You shouldn't attach at components here. There is a specific metadata you should attach in such cases. It's **@ injectable( )** which needs to be imported from at angular core.

This tells Angular that this service is injectable or that something can be injected in there to be precise. So that's important. You don't add at injectable to the service you want to inject but to the service where you want to inject something. So the receiving service you could say that to be injected in service and you don't need to add it to any other service if you don't want to inject anything in the service like here for the logging service.

You don't need to add @injectable only added If you expect to get something injected. Now one important note here as I explained we don't need to add injectable the injectable decorator here to this class because this is a service which we only inject somewhere else but where we don't inject anything into this service and you technically really only need injectable if you inject something into this service. Not if you just inject this service somewhere else in newer wordings of angular however it is recommended that you always add at injectable it's technically makes no difference right now in the future it might though.

So it's already a habit you might get into that you added here. I do admit it but you could consider adding it also to ensure that in the future it doesn't unexpectedly break. What I explained about why you need it and that you don't need it here technically still holds though with this if I now safeties and we let the app reload now it works. And now you see that if I save this we still log it to the console with the correct status the same happens if I click here. But now everything is centralized into our service and now we're also injecting a service into a service.

**#### 112. USING SERVICES FOR CROSS COMPONENT COMMUNICATION**

We talked a lot about services in this module and I hope you saw how services can clean our app up, make the code a bit leaner, more centralized, easier to maintain, you don't have to build these complex output input chains where you pass events and properties to get data from component A to component B, it's much leaner now. Let me show you how much we actually saved.

Let's say that if we click on the button here in the **account.component**, for some reason, we want to output something in the new account component. Normally without services, we would have to emit an event in the code component, that something was clicked or something happened so that we change the status for example, we would have to catch the event here in app account.

As a side note, we can of course remove these events here, they aren't triggered anymore. So we would have to catch these events here and then we would have to pass the new data down via property binding to the component where we wanted to handle it, pretty complicated and building these chains of property and event binding is not the most convenient way of writing code, it's so much easier with services. Let's say in our accounts service here, we want to provide some event which we can trigger in one component and listen to in another.

So we could simply add the status updated event which could be a new event emitter which we import from @angular/core, so make sure to add this import at the top which will pass on the new event, excuse me, the new status, so a string.

$ **statusUpdated = new EventEmitter<string>( );**

yes we could trigger it now here in the update status method but we can also since we inject the account service here where we set a new status, call account service, status updated and emit this with the new status.

$ **this.accountService.statusUpdated.emit(status);**

So now I'm emitting any event I set up in the service and as a side note later in the observable section, you will learn about another construct you can use to submit or to emit events and subscribe to it instead of using the event emitter but for now, it's absolutely fine. So here we are emitting an event, again the event emitter lives in our service and in the new account, I now want to listen to it.

So here, let's say I want to throw an alert. So I will do this in the constructor for now**( inside new-account.component.ts**), access my account service and now for the status updated event here, I will subscribe to it because event emitter in the end just kind of wraps an observable. So here, I could then receive the new status which I know will be a string and then it will simply throw an alert where I say new status, status, something like that.

**$ this.accountService.statusUpdated.subscribe(**

**$ (status:string) => alert( ‘New Status: ‘ + status)**

So now, I'm not building any chain of property and event binding, I do have cross-component communication through a service with the event emitter. And you can see this if I click a button here, it actually got opened on another window but you see this alert was thrown, with the new status unknown,

same here in active, active. So now we're communicating between components through a service which really can save you a lot of time and with that, I hope you saw a lot of reasons why services can be very helpful and how you can use them to your advantages. Make sure to use the right amount of instances

and if you inject services into services, make sure to provide the service on the app module level and to add @injectable to a service where you want to inject it in. Let's move on to some assignments now.

**#### 113. SERVICES IN ANGULAR 6+**

If you’re using “Angular 6+” (check your package.json to find out) you can provide application-wide services in a different way.

Instead of adding a service class to the “providers[ ] array in “AppModule”, you can set the following config in “@Injectable( )”

**$ @Injectable( { providedIn: ‘root’ })**

**$ export class MyService { . . . . . }**

This is exactly the same as:

$ **export class MySerivce { . . . . . . }**

And

**$ import { MyService } from ‘./path/to/my.service’;**

**$**

**$ @NgModule( {**

**$ . . . . .**

**$ providers: [MyService]**

**$ })**

**$ export class AppModule { . . . . }**

Using this new syntax in “completely optional” the traditional syntax ( using “providers[ ]” ) will still work. The “new syntax” does offer one advantage though: Serivce can be loaded lazily by Angular ( behind the scenes ) and redundant code can be removed automatically. This can lead to a better performance and loading speed – though this really only kicks in for bigger service and apps in general.

**#### ASSINGMENT : PRACTICING SERVICES**

**THERE ARE TWO PARTS FOR THIS ASSIGNMENT**

**PART 1 : WITHOUT SERVICES PROJECT NAME – “services-assignment-start-mysolution”**

**PART 2 : USING SERVICES PROJECT NAME – “services-assignment-start-mysolution-part-II”**

Time to practice the usage of Services. You’re going to build, inject and use your own Services. Make sure you get the instance management right!

So where are you successful. Here's my solution.

I will create “**users.service.ts**” and in there we of course **export a class UsersService.**

**$ export class UserService { ………………………………. }**

We don't need to add an injectable right now because right now we're not injecting a service in there

and here and I want to manage my users. Now you could create a one array of users and they store objects which for each object. It also tracks the state of the user.

I would go with two arrays so I will have my **active** and my **inactive** users to two arrays from the app component so I'll copy it from there and paste it into my user service.

  activeUsers = ['Max', 'Anna'];

  inactiveUsers = ['Chris', 'Manu'];

But again as an improvement you could store objects in one single array there which then allows you

to switch to status for each object for each user. So there's this and then I want to **add a set to active method here**.

$ **setToActive( ){…………………}**

I expect to get the id of the element ( **id: number** ) of should switch and **add a set to an active method**.

$ **setToInactive( ){…………………}**

And here of course also you could optimize this by having one method which does both and which receives an additional argument which determines whether you want to switch active to active or the other way around.

Well in **setToActive** I want to **reach out to my active users** and **push a new element** to them that **new**

**element is in my in active users** the user with the idea I'm receiving there.

$ **this.activeUsers.push(this.inactiveUser[id])**

And thereafter I can **simply splice** that user splice will simply remove one element in this case here.

$ **this.inactiveUsers.splice(id, 1);**

Well now I can copy that pasted in here and reworks it. Push it to the inactive users from the active users and splice it from the active users.

setToInactive(id: number) {

    this.inactiveUsers.push(this.activeUsers[id]);

    this.activeUsers.splice(id, 1);

Now this service is prepared and now in my app component I can't get rid of these two methods. Remove **setToActive** and **setToInactive**. I also can delete my **users** here. So we can completely clean this component and we can clean everything in the template of the up component. We don't need to pass the properties to our components there because these components can reach out to the service on their own.

Remove everything inside <app-active-users> and <app-inactive-users>

Now it is important to make sure that we provide the service on the right level.

If we were provided in the active user component and the active euro component we would have two different instance of that service and therefore it would not work we could not switch because we would work with two different arrays.

So with four different arrays actually. So therefore this is not the right place to provided we could provide here in the app component or in the module. Now since we don’t plan on injecting this service in another or service or something like this there is no reason to provide an application wide though you could do that. That’s not wrong not bad. But I will simply provide in the app.component.

But again just to highlight this. Its not bad or wrong to provide it in the app module. So here I will provide my users service ( **providers: [UserService]** ) make sure to also **add the import** with that provided on this level. Active users and in active users will actually share the same service the same instance of that service. So now I can go to inactive users **add a constructor** here and get my users service here. So of type users service of course user services in the name actually make sure to add the import here too.

constructor(private userService: UserService) {}

You always need to add this data a typescript feature not related to angular. And then I can remove the input here( **@Input( ) users: string[ ]** ). Were not receiving it from outside anymore. I can remove my whole output ( **@Output( )** ) My whole event here ( **EventEmitter** ) and also remove this very I am emitted. Remove all these imports therefore. And now I want to initialize my users array. I will do it. Add on in it as. ( **implements OnInIt** )

Again it is a good practice to do initialization work in there. So let's implement on it make sure to import it and then we need to **add the ngOnInit ( )** on that method. And then there I can set my users equal to the users I've fetched from my users service and we're in the in the active component here. So I should get my in active users.

 ngOnInit() {

    this.users = this.userService.inactiveUsers;

  }

Now with that I can simply copy that code over to the active users replace my input and output here to and remove the input string. ( the **constructor and the ngOnInit( )** )

Remove all these imports but on the aberant make sure to import user service and implement on it or

just like in the broken bone and make sure to import on it too. And import the servers and here we don't have to emit this event anymore. Instead make sure that now users is in active users but the active users since we're in the active users component.

And if we click on set to in active I want to reach out to the users service and call set to an active

and pass on this ID. And on the other hand and in active service I want to do the same reach out to the users servers and hear a call set to active and pass the ID data should be all. Let's see if this works if our app here is really loaded. I can still switch my users around.

So this first part does work but boy is it leaner much less passing around instead having a clean central

service. Now the other part of the task was to add to counter-service. So let's add this to the counter service. Yes File. Let's export a class here. The counter service like dis.

And here I simply want to manage my active two in active counter which starts at zero. And well you guessed it in active to active counter which also starts at zero. And now I simply want to add two methods where I say increment active to in active. Of course you can choose another name and he will simply say active to inactive count plus plus and locked a new counter.

So the active to inactive counter. And then I will just duplicate this to also have my in active to active counter where increased in active to act of counter property and locked in active to active counter property. You probably dream about in active to active Does night but does this are counter-service. Now here you could it in multiple places you could add a year in the onset to active already on onset to an act of Moffats that would be all right. But I want to add here in my user service.

So I want to inject that into a service. And for days we have to take two things into account. First of all we have to make sure that we provide a counter service here in the AB module. So counter service make sure to add the import here to now it's available every application wide and therefore can also be injected into our services segment to inject a service into a service. What would you need to do. You need to add at injectable to the service where you want to inject that in. Right.

So do the users service here not to counter-service to use service to the receiving service. So here I will add an injectable. Don't forget the parentheses that's a common mistake and D.M. or it. And with that I can now inject the counter-service in the users service. So your counter WIPs counter-service of type counter service make sure to import this type here at the top to and with that and check that here in the set to be active I can call this counter-service. And since we set something to act that I want to call the increment in active to active because we set something from inactive to active and on the other hand.

And he said to an active I want to increment the well active two in active. So this one here with Dad if we now save this and that are rabri Lote and I clicked to set to active we see that this one maybe improved the output here that we say active to in active and then we output the number on the arahant on. We have an active two active. So now let's just reload active to inactive for one 2.

And now we should add 4 to the inactive to active side and some to the active to inactive. So this is working. Everything seems to work as intended. This is my solution. Of course you might have a different solution. There are different ways of achieving the same overall. You should feel confident using services.

Now you should see their benefits and how you might use them in your next project.

**#### 115. INTRODUCTION**

Welcome back to the recipe project.

We already added a lot of features to it, now it's time to add services to it which will give us the possibility to greatly enhance this project.

Which services do we need?

Let's have a look at our application again, this is the application the way it looks right now. Now services allow us to centralize tasks, manage data in a central place and so on. So we probably need a service for both feature areas, we will need a shopping list service and a recipe service. The shopping list service managing our shopping list and we will also access it from the recipe, from the recipe area because we want to add items to the shopping list from there and well the recipe service, that will be responsible for managing our recipe, managing the data there and so on.

So we'll use both services and as always, the challenge for you, create both services. For now only create the files and try to find places in our app structure where they just fit in and thereafter, we will fill them with some life together. So let's start together in the next section, try to create these files on your own.

**#### 116. SETTING UP SERVICES**

So let's add some services manually.

Now we should place the services next to the feature or in the folder holding the feature they belong to.

So if we create a recipe service, the recipes folder looks like a good place for it. So here, I'll add a recipe service and of course the file is named **“recipe.service.ts”** due to that and here I'll simply **export a class** which is named “**RecipeService**”.

Remember, a service simply is a TypeScript class, you don't need to add any special decorator to it, at least not if you don't plan on injecting a service into the service. Well and the same for the shopping list service. I'll place it in the shopping list folder, here I'll name it **“shopping-list.service.ts”**, that's the file name and then inside the file of course, I'll **export a class** named **shopping-list service**.

So these are the two service files. Now let's start working on the recipe service first,

let's start working on it in the next lecture.



**#### 117. MANAGING RECIPES IN RECIPE SERVICE**

What should our recipe service do for us?

I guess it makes sense if the recipe service is the place in our app where we manage our recipes. Therefore, we should take our recipes which we currently manage in our recipe list and manage it in the recipe service, that's the first thing with which I'll start, so I'll simply copy:

  recipes: Recipe[] = [

    new Recipe('A Test Recipe', 'This is simply a test', 'https://upload.wikimedia.org/wikipedia/commons/1/15/Recipe\_logo.jpeg'),

    new Recipe('Another Test Recipe', 'This is simply a test', 'https://upload.wikimedia.org/wikipedia/commons/1/15/Recipe\_logo.jpeg')

  ];

it for now and add it as a property in the recipe service.

Of course we now have to **add the import** pointing to the recipe model file which sits in the same folder

as the service. So with that, we get an array of recipes in the recipe service and I will actually make this private so that you can't directly access this array from outside.

private recipes: Recipe[] =

Now with this added, I also want to **add a getRecipes method** which will return this array, so that we

can get access to it from outside. If I were to return it like this:

 getRecipes(){

          return this.recipes;

      }

I actually return the direct reference to this array and since arrays and objects are reference types in Javascript, so has nothing to do with Angular, vanilla Javascript that is, well if we now change something on this array, we will change it on the array in this service.

Therefore, I will **call slice** with no arguments, this will simply return a new array which is an exact copy of the one in this service file. So therefore, we really can't access the recipes array stored here from outside, we only get a copy, so that is a way to get our recipe.

Now with that, we can and we have to add our service, actually both services to a place in our app to provide them so that we can inject them. Where should we provide both services? Well let's start with providing the recipe service in our recipe component, we can always put it somewhere else later if we recognize that we want access to it from another part in our app but for now, I'll **add providers**, this array to the recipe component, therefore this component and all other recipe related components, since they are all child components to this component in the end, will share the same instance of this service but you won't be able to access the service or at least not the same instance in other components like the shopping list component.

So here in providers, I will list my **RecipeService**, of course you have to **add an import** at the top telling TypeScript where to find this type. Now with this, we use our or we can use our recipe service here,

now let's use it in the recipe list component. Here right now, we already manage our recipe array,

now I don't want to do that here anymore, it will be undefined initially, we will get the value here in ngOnInit and for this, we need to inject our service. Now we can do this since we just provided it on the parent component, so here I will provide my recipe service which is of type recipe service.

constructor( private recipeService: RecipeService) { };

Of course make sure to import recipe service from the appropriate file and this should be private here, to use this TypeScript shortcut of instantly assigning a property with the same name. In ngOnInit, I can now say this recipe equals this recipe service get recipes and we will get this copy of recipes, this copy of that array.

  ngOnInit() {

    this.recipes = this.recipeService.getRecipes();

  }

Well, let's see this in action. If we save this, we should and we do see the same as before, the app works just like before because the only thing we changed right now is where and how we manage our array of recipes.

An important first step.

**#### 118. USING A SERVICE FOR CROSS COMPONENT COMMUNICATION**

In the last lecture, we used our recipe service and we create it in the first place, we filled it with some life

to manage our recipes in a central place. That is working great, now let's use that service to improve our app. Remember, we have this long chain of inputs and outputs getting a selected item from the recipe item component over the recipe list component to the recipes component, here

(recipeWasSelected)="selectedRecipe= $event">

where we listen to that event and then all the way down using the selected recipe property to the recipe

detail component, so this cross-component communication. That is a super long way for just informing

another component that we selected an item but we can do better and we should.

So what my goal is that in the recipe-item.component where we have is **(click)=onSelected( )**, that here we can directly inform the recipe detail component.

So in onClick what I actually want to do here is, I want to not emit an event here,

@Output() recipeSelected = new EventEmitter<void>();

I will get rid of it here( from recipe-item.component.ts) and therefore I can remove the imports from, of output an event emitter, instead here I want to call some method( **onSelected( )** ) in my service which will then transfer this data for me.

So in the recipe service, I'll add a new property, the public property which will name recipeSelected, of course you could choose a different name, it will be an object instantiated by using event emitter from @angular/core, so you need to add this import, let's put it at the top of that file, that's kind of a good practice to do with your Angular imports.

$ **recipeSelected = new EventEmitter** **<Recipe>( )**  // inside recipe.service.ts

It will hold some recipe data and that's all I want to do in the recipe service. With this set up and of course you could also encapsulate this by providing a method to get access to it and want to emit the event

but I will use the property itself. So by adding this, I can now go back to my recipe item component and in onSelected( ), I want to do something with that property in the recipe service.

So first of all, let's inject it, let's inject the recipe service as you learned to do it, make sure to import it, point it to the recipe service file on the right folder and then on the recipe service, we can use this recipe selected event emitter and call emit and emit the recipe of this recipe item component because that is the one we selected and that's the data we want to pass.

this.recipeService.recipeSelected.emit(this.recipe);

Now with this trick or with this approach, we can go to the recipe list component and remove this listener

(recipeSelected)="onRecipeSelected(recipeEl)"

where we listened to whether we did select it or not, we can get rid of the onRecipeSelected code

onRecipeSelected(recipe: Recipe) {

    this.recipeWasSelected.emit(recipe);

   }

and get rid of our own emitted event here,

@Output() recipeWasSelected = new EventEmitter<Recipe>();

so we're already saving a lot of code, get rid of all these imports which we don't need anymore and in the recipes.component.html file, we can also get rid of this event here

(recipeWasSelected)="selectedRecipe= $event">

on the recipe list because we no longer use this chain of data to pass it around and now all I want to do is in this component here( **recipes.component.ts** ) , I want to listen to this event in the recipe service

so that I can now correctly decide whether we do have a selected recipe and should therefore render

the detail component or if we should still show this template with this please select a recipe text.

Therefore, I will inject the service here (**recipes.component.ts)** to recipe service and since I provide the recipe service on this component, this component and all the other components in this recipe folder use the same instance of the service which of course is super important, otherwise the service where I emit the event would be a different one to the one I listened to it, so I would never get informed about the event but here it will work because we're using the same instance.

constructor( private recipeService: RecipeService) { }

So here(**recipes.component.ts)** in ngOnInit for example, I can set up my listener, so on the recipe selected, I can subscribe to it and get informed about any changes. And now here, I know I will receive some data which is of type recipe because that's how we configure the event emitter, it would give us a recipe and in this ES6 arrow function here, that is the argument list and here is the function body, here I simply want to set this selected recipe equal to the recipe we got with the event.

 ngOnInit() {

    this.recipeService.recipeSelected

      .subscribe(

        (recipe: Recipe) => {

          this.selectedRecipe = recipe;

        }

      )

  }

And with that, we should get the same behavior as before if we reload the app but now, having a much leaner approach of using a service for cross-component communication which of course makes it much easier to pass this data as you saw and we don't have to build this complex chain of events and property binding.

**#### 119. ADDING THE SHOPPING LIST COMPONENT**

So in the last lectures, we improved our recipe section a lot with the service, now let's work on the shopping list side, on the shopping list service. This service is empty right now, now here's your challenge,

fill it some life. It would be nice if this service could manage our shopping list, so our list of ingredients,

maybe you even add the addIngredient method here so that you can add ingredients using this service

too, we already got this add functionality here in the shopping list component.

It shouldn't live here anymore though and with that, we would take a huge step forward. So your chance to pause the video and try this on your own, thereafter we'll solve it together.

So were you successful?

Let's fill this shopping service with some life. For this, I'll first of all copy my ingredients here, inside shopping-list.service:

   ingredients: Ingredient[] = [

   new Ingredient('Apples', 5),

   new Ingredient('Tomatoes', 10),

  ];

and add it here as a property to the service and of course I need to import ingredient from the shared folder, from the ingredient model there.

Now with that, we got the data in here and I will make **this private**. Now just like in the shopping, excuse me, in the recipe service, I will add a **getIngredients( )** method here where I will return my ingredients but only a copy of it with the slice method, so that I can't access the original array stored in this service.

  getIngredients( ){

      return this.ingredients.slice();

  }

With that, I can go back to my shopping list component and I will set this to an uninitialized property but I can inject my shopping list service.

ingredeints: Ingredient[];

Now again the question is, where should we provide it? Now for now,

we can provide it here in the shopping list component and therefore it would also be available in the shopping edit component but actually later I also want to access it from my recipe section. That is why I will actually provide it in app module and here, I will add it to this providers array therefore, shopping

list service.

providers: [ShoppingListService],

Make sure to also **add the import** at the top here. So now we will be able to use a service application-wide including in other services and I exactly need this behavior because we will soon use it in another service.

So back to the shopping list component then, we can now inject a shopping list service and I will bind it to a property named slService, of course you can choose a different name which will be of type shopping list service and you need to add the import pointing to the shopping list.service file.

constructor(private slSerice: ShoppingListService) { }

With that in ngOnInit because you should do all tasks which required a bit more heavy lifting or in general as a good practice, all initializations in ngOnInit.

Therefore here, I will simply assign my ingredients to whatever the shopping list service returns me

if I call getIngredients here.

ngOnInit() {

    this.ingredeints = this.slSerice.getIngredients();

  }

With this in place, if we save this and we go to a shopping list area, this is still working so that's great, now let's make sure that we can also add items. So I'll **add the addIngredient( )** method here (inside shopping-list.service) and I know that I will **receive an ingredient** here of **type ingredient** and all I want to do then is I want to **access my ingredients** and push this **new ingredient on it,** so here, ingredient.

 addIngredient(ingredient: Ingredient){

      this.ingredients.push(ingredient);

  }

Now if we go to the shopping edit component, right now we're emitting an event with our own ingredient. We no longer need to do this so we can get rid of the event,

this.ingredientAdded.emit(newIngredient);

of output and of the imports here related to this,

@Output() ingredientAdded = new EventEmitter<Ingredient>();

we can also go to the shopping-list.component.html file and remove our listener where we listen to

(ingredientAdded)="onIngredientAdded($event)"

this event and in the shopping-list.component.ts file, we can get rid of the onIngredientAdded method because we'll now do this in the service.

 onIngredientAdded(ingredient: Ingredient) {

    this.ingredients.push(ingredient);

  }

So back to shopping edit component where we create a new ingredient, here want to use the service and for this, I will access my service so I will inject slService here, like this,

  constructor( private slService: ShoppingListService) { }

make sure to add the import of course and in onAddItem, I'll add slService and add an ingredient to it, this new ingredient here.

this.slService.addIngredient(newIngredient);

Now if we save this and now try adding a new item here, bread one, we don't see it but we also don't get an error. Now there is something wrong here and we will find out what and fix it in the next lecture.

**#### 120. USING SERVICE FOR PUSHING DATA FROM A TO B**

In the last lecture, we basically finished our shopping list service but here's one thing which is not working.

I can click this add button as often as I want, I don't get an error but I also don't see it getting added to this list. The reason for this is that when we call getIngredients which we of course do to populate this list,

we only get a slice of the ingredients array,

  getIngredients(){

      return this.ingredients.slice();

  }

so we get a copy. Therefore once we add a new ingredient which we of course don't add to the copy but to the original array, that is not reflected in our components where we only have that copy to work with.

Now there are a couple of ways of solving this, for the easiest solution, **you could remove slice** and

that's not necessarily bad, you would then simply return the original array and if you're sure you don't accidentally edit it somewhere, this might not be wrong. I choose this approach though because I want to show you a different solution, we basically have to inform our component that new data is available.

So I will simply add a new event emitter here and I'll name it **ingredientsChanged** and I'll set this to or I'll of course set this to a new event emitter to be imported from @angular/core. This event emitter now can emit our ingredient array, so the type it will pass on is an array of ingredients

 ingredientsChanged = new EventEmitter<Ingredient[]>();

and now here whenever we change this array, we simply call this **ingredientsChanged** and emit a new event

  addIngredient(ingredient: Ingredient){

      this.ingredients.push(ingredient);

      this.ingredientsChanged.emit();

  }

and we of course pass a value here, our original ingredients array, though to be precise again, only a copy of it.

this.ingredientsChanged.emit(this.ingredients.slice());

So now we always have the right ingredients array in the service and we inform other interested components about that change in the event we're emitting here.

So now in the shopping list component, besides getting the ingredients at the point of time we load the app, I also want to reach out to my shopping list service and simply subscribe to that ingredients changed event.

So now whenever ingredients change, I know that I will get them, this will be of type ingredient array here as we learned and now I can use them here in the body of this anonymous function I'm passing to subscribe and here, I can now simply set this ingredients equal to the ingredients I got here. And with this change in place,

 this.slSerice.ingredientsChanged

      .subscribe(

        (this.ingredients: Ingredient[]) =>{

          this.ingredients = this.ingredients;

        }

      )

now if we let the application reload here and go to shopping list and now I add bread and now you see it gets added here again without any issues because now, we're getting informed that something changed.

**#### 121. ADDING INGREDIENTS TO RECIPES**

We're almost finished, our application is taking shape and services really improved it. Now I want to do something here on the recipe section, I want each recipe to have some ingredients, so we need to implement this and thereafter, I want to make these two shopping list buttons work, so that we can send our ingredients to the shopping list.

So to do this, what I will do is first of all, I will work on the recipe model, it should now also have ingredients. So this will be of type ingredient array, ingredient: (inside recipe.model.ts)

public ingredients: Ingredient[];

of course needs to be imported from the respective model file in the shared folder and when we create a new instance of this model, I expect to get some ingredients here:

constructor(name: string, desc: string, imagePath: string, ingredients:Ingredient[]){

Again, this will be of type ingredient array, so that here we can assign ingredients equal to the ingredients we get as an argument here.

this.ingredients = ingredients;

So with that, we are prepared to have some ingredients, now in the recipe service, I also need to add some here.

So when we create a new recipe and I simply structure this over multiple lines to make it easier to read, I also now will pass an array of ingredients, the same for the second recipe here and now I guess it's time to also turn this into some real recipe, so let me quickly do this.

  private recipes: Recipe[] = [

        new Recipe(

            'A Test Recipe',

            'This is simply a test',

‘https://……………………………………..

‘[ ])

‘

So I quickly changed the pictures in the text to have a schnitzel and a burger here and now let's add

some ingredients here in this array of ingredients.

Inside the empty array [ ]

We can use the ingredient constructor for this and of course we need to import the ingredient type here from the model file located in the shared folder for that and now for the Schnitzel, let's say we need the meat, with an amount of one maybe and then we also want to have some french fries here, now let's say exactly 20, with one extra it would probably not taste well I guess.

  [

                new Ingredient('Meat',1),

                new Ingredient('French Frieds', 20)

            ]),

So now with the array for the burger, here we need a new ingredient, we need the buns, so the bread basically, let's say 2 and we also want to have the meat here, of course feel free to add more ingredients.

 [

                new Ingredient('Buns', 2),

                new Ingredient('Meat', 1),

                new Ingredient('Topping', 2)

            ])

So now that we added that in the recipe detail component, here where we have this ingredients placeholder, we can actually now output them here. So let's add something here, let's add an unordered list with a class of list-group and now here I want to add a list-group-item. This item here will actually be replicated because again, this will of course be my ingredient.

So here, ngFor and now I will loop through all the ingredients, so ingredient of and now of recipe ingredients of course because remember in the recipe detail component, we have a recipe and that has some ingredients since a couple of seconds at least. So now, we can simply use string interpolation here to output the ingredient name and then maybe separated by a dash, the amount, of course any format you like.

 <ul class="list-group">

            <li

            class="list-group-item"

            \*ngFor="let ingredient of recipe.ingredients">

            {{ingredient.name}} - {{ingredient.amount}}

            </li>

        </ul>

So that's the first step, now if we select a recipe, we see a list of ingredients at the bottom. Now I want to send these to the shopping list if I click this to shopping list button. So that is what we'll work on in the next lecture.

**#### 122. PASSING INGREDIENTS FROM RECIPES TO SHOPPING LIST ( VIA SERVICE )**

In the last lecture we added some ingredients, now I want you to make this to shopping list buttons work. You can come up with some ideas, some solution on how you can send these items to the shopping list

so that once we go to the shopping list, we see them here. Think about informing the component that something changed and also think about how to add more than one item at once to the shopping list.

I wish you luck with that, time to pause, in a couple of seconds, we'll do it together.

So were you successful?

Let's add the shopping list items together. We already have this button in the **recipe.detail component**,

so the first step of course is to hook this button up. I will remove the ref here: ( **href=”#”** )

and instead add an inline style to still have a cursor which looks like a pointer but then here on the click listener, I'll call **onAddToShoppingList( ),** of course you can choose a different name.

<li><a (click)="onAddToShoppingList()" style="cursor: pointer;">To shopping list</a></li>

Now that will be executed here in the component body, in the **recipe-detail.component.ts** file and here:

we either need to get access to the shopping list service or to the recipe service and then the recipe service accesses the shopping list service. You can find arguments for both, you could say we're in the recipe section therefore we should mainly work with the recipe service, you could say why would we take this extra way, this extra step, we can directly access the shopping list service, again arguments can be found for both directions, I will choose the route where we use the recipe service.

So I will inject the recipe service here, type recipe service of course, make sure to add the import

constructor(private recipeService: RecipeService) { }

and now I need to add a method in the recipe service. This is the **addIngredientsToShoppingList( )** method, of course you can also pick a shorter name if this one is a bit too long for you. Here I expect to receive my ingredients, so that will be of type ingredients

    addIngredientsToShoppingList(ingredient: Ingredient[]){

      }

and before we continue working on this in the recipe service, let's call this method.

So in the recipe detail component, I will access my recipe service, call this brand new method with the extremely short name and pass on the ingredients of this recipe here, so these ingredients.

 onAddToShoppingList(){

    this.recipeService.addIngredientsToShoppingList(this.recipe.ingredients)

  }

So with that, these ingredients are passed to the recipe service, now here we need to access the shopping list service. Now for that, we need to get access to the shopping list service in this file and if you want to inject a service into a service as we want to do here, you need to add @injectable to that service, make sure to import it from @angular/core. With that added here, we can inject the service now in the constructor of this recipe service class. Here: @Injectable( )

constructor(priviate slService: ShoppingListService){}

I'll add the slService, so the shopping list service, to which you'll need it to add an import at the top pointing to the shopping list folder and then the shopping list service file and with that added, we can now reach out to that here, slService and here we need to add a new method

 addIngredientsToShoppingList(ingredient: Ingredient[]){

        this.slService;

      }

. So let's go to the shopping list service and let's add an addIngredients method, like that.

 addIngredients(){

  }

Now in here, we will receive our list of ingredients, so of type ingredient array and now we need a logic to add them.

Now a quick and easy way would be to simply loop through them, so let ingredient of ingredients, like that and then we could simply call this addIngredient for each ingredient and that would be absolutely fine,

  addIngredients(ingredients: Ingredient[]){

    for ( let ingredient of ingredients){

        this.addIngredient(ingredient);

    }

  }

the only downside probably is that we emit a lot of events. Now it won't be too bad because even a recipe with like 30 ingredients wouldn't blow up our app but still, there are a lot of unnecessary event emissions.

So that is a viable option but I'll comment it out, a different one would be to directly add all ingredients

in one go and then emit our event. So we could access our ingredients here and call the push method and now we can use an ES6 feature, the spread operator which allows us to basically turn an array of elements into a list of elements because the push method here is able to handle multiple objects.

However, it is not able to handle an array or to be precise, it can handle an array but then it would push this array as a single object to the other array. So by adding the spread operator which is simply three dots, might look strange but that is how the operator looks, we can now simply spread our ingredients into a list of single ingredients which are now pushed without issues to our ingredients array and then don't forget it, we have to emit that our ingredients changed, so why don't we simply pass a copy of it.

  addIngredients(ingredients: Ingredient[]){

    // for ( let ingredient of ingredients){

    //     this.addIngredient(ingredient);

    // }

    this.ingredients.push(...ingredients);

    this.ingredientsChanged.emit(this.ingredients.slice())

  }

So with that now, we get a method we can call, **addIngredients( ).** So back in the recipe service on the shopping list service, we can now call **addIngredients( )** and pass our ingredients to that service.

 addIngredientsToShoppingList(ingredients: Ingredient[]){

        this.slService.addIngredients(ingredients);

      }

With that let's save this and let's simply check this. Right now we only have the default values of apples and tomatoes on our shopping list, now on the Schnitzel we got meat and french fries, let's click to shopping list and let's go to the shopping list and we see meat and french fries, awesome, that looks really great.

Of course if you click this multiple times, it will duplicate it and you can of course write some logic to check if an item has already been added and then group it together by name, I won't do this in this application.

The main thing here is it works, we can send data around through our app by using services. We took a major step forward, now in the next section, we're going to have a look at routing and thereafter, we're going to add routing to this application to now get a proper navigation and also to make this new recipe

button here work. So I'll see you there.

**SECTION 11: CHANGES PAGES WITH ROUTING**

**#### 123 . MODULE INTRODUCTION**

Welcome to another course module, welcome to this course module. We already learned a lot about Angular and thus far, we really built a single page application. We use directives and so on to change portions of this page but in the end, we always were on the same page,

if you looked at the URL at the top, it was always localhost:4200.

Now what if you wanted to have a page where you actually display several pages, like you have a

/users or a /accounts page? Now you might think that for this, you indeed need multiple pages, multiple index.html files or something like this but you don't, 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 a lot of parts of that page, so that to the user, it really looks like a new page was loaded because maybe only the header is the same but behind the scenes, it is still Javascript changing a lot of parts in your DOM, in

your page, making it look like a new page was rendered but you're still in the Angular world in your

single page application. Let's take a closer look in this module.

**#### 124 . MODULE INTRODUCTION**

Attached to this lecture, you find this example project here. Now on the first look, it might look nice but it has one fundamental issue, we can't switch the pages here.

We get some tabs at the top but these tabs are not working and therefore what we see on this home page

are actually all the components. I later want to split up over multiple pages or multiple pages in quotation marks I should say in this module. We do have a home, a servers and a users page and as you can see here, we got users, servers and the home page on the same page. So that is something we can improve but it would be nice to have separate pages as I said, so a /users page or a /servers page and so on.

So this is what we're going to work in this module and for this, we need routes, we need the Angular

router and the Angular router needs to know which routes our front-end application has. So you might know this pattern from a full-stack application where you also register some routes the user can then visit, we can do the same in Angular and we will do this in the next lecture

**#### 125. WHY DO WE NEED ROUTING?**

Attached to this lecture, you find this example project here. ( Original 4 > Routing Start )

Now on the first look, it might look nice but it has one fundamental issue, we can't switch the pages here.

We get some tabs ( home / servers / users ) at the top but these tabs are not working and therefore what we see on this home page are actually all the components I later want to split up over multiple pages or multiple pages in quotation marks I should say in this module. We do have a home, a servers and a users page and as you can see here, we got users, servers and the home page on the same page.

So that is something we can improve but it would be nice to have separate pages as I said, so a /users page or a /servers page and so on. So this is what we're going to work in this module and for this, we need routes, we need the Angular router and the Angular router needs to know which routes our front-end application has.

So you might know this pattern from a full-stack application where you also register some routes the user can then visit, we can do the same in Angular and we will do this in the next lecture.

**####: UNDERSTANDING THE EXAMPLE PROJECT**

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 definitely feel free to play around with it – besides routing, everything should be working fine.

**#### 126: SETTING UP AND LOADING ROUTES**

So here I am in the source code responsible for the app you saw in the last lecture and as you can see, here I got three major components and then some subcomponents of these components. In the app component, I'm loading them all, my home component, my users component and servers component.

But for now, I'm going to ignore my subcomponents here, edit server and a single server, so we will later nest this even more but for now it would be nice if of these three components, I would only display one at a time so that we can dynamically load them by clicking on these links here

<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>

at the top and we really load a different route in our app.

Now where do we register such routes?

Let's think about this logically. These routes are responsible for our whole app. Everywhere in our application, if we enter /users, we want to load the users component as our main page let's say. So since this is really a core part of our app, where should we register it? Well, the app module sounds like a good place because here is where we configure our app, where we add all these components and so on, so this might be a good place to also inform Angular about the routes our application has and it is.

So here above this ***@NgModule*** decorator here, I will **add a new constant** and I will name it appRoutes

const appRoute

because it should hold all the routes of my application. Now these routes constant should be of a specific type, the routes type, which needs to **be imported from @Angular/router**.

const appRoute : Routes

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

So make sure to add this import at the top, this is the type informing Angular or giving these routes some structure I should say. You don't necessarily have to add it here but it is a good practice. Now with this import added and the constant set up, what should this constant actually hold? It should **hold an array** because we will have multiple routes. So we add all the routes we want to set up to this array for now and each route now is just a Javascript object in this array. Now the question is, how should such a route be configured in an Angular app?

const appRoute : Routes=[

  {

  }

]

It has to follow a specific pattern, a specific structure for Angular to be able to use it and this structure always **needs a path**, this is what gets entered in the URL after your domain. This should be a string and the first route I want to add could be my users route, this allows me to then visit localhost or whatever your domain is, 4200 in this case here and then /users.

const appRoute : Routes=[

  { path: 'users' }

]

So that is your path without the slash and this is important, make sure you don't add the slash here, this would be incorrect added without the slash. So this is now the path and this will later allow Angular to determine which route in our array of routes here should be selected if we enter something in the URL or click some link because it clearly defines well which path this route should be.

Now we also need to define what should happen when is path here is reached. Right now, nothing happens, it's a route but yes, there is no action attached to it. The **action typically is a component** so that you inform Angular that once this part here is reached, a certain component should be loaded and this component will then be the page you could say which gets loaded. So here for users, it sounds about right to load our **users component** and of course it's up to you to make sure that in your application, you are indeed configuring your components you want to serve as pages, to look like pages, to have all the content you want to have on this page.

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

In this case, that's the case for my users component, here at least for now it will output my list of users and then the single user component next to it and we will later improve this to also be able to select the user from this list of users here but for now, that's fine.

So this is now my first route with the users component getting loaded once we target /users. Now let's add a couple of other routes, I'll duplicate this or replicate this twice. I also want to have a certain page I display once we have just localhost:4200, so an empty path after it, you could say. So we should also create a route for this with just an empty path and here I want to load my home component, this is the starting page you could say. You of course don't need to have such an empty path but then you somehow need to handle this, well the user accesses an empty path differently and I will come back to this later.

{ path: '', component: HomeComponent },

So for now we have an empty path, just localhost:4200 loading the home component, /users will load the users component and I also want to have /servers which will load my server or servers component.

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

With that, three routes are set up here but that alone won't do anything. How would Angular know that you want to use this constant? You could have added any constant here and the name appRoutes, well that is totally up to you. So right now, these routes will be ignored by Angular and the type routes here doesn't do anything when it comes to this.

const appRoute : Routes=[

  { path: '', component: HomeComponent },

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

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

]

So we somehow need to register these routes in our app and we do this by adding a new import here to this imports array, we need to add the RouterModule here and this RouterModule should be imported from our @angular/router package here.

 imports: [

    BrowserModule,

    FormsModule,

    RouterModule

  ],

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

So with this added, we're adding the routing functionality to app but still, our routes are not registered. That is why this RouterModule has a special method we can call, forRoot, which allows us to register some routes for our main application here.

RouterModule.forRoot()

So forRoot will now receive our **appRoute constant** here as an **argument a**nd with that, our routes are now registered in our Angular app on this RouterModule which gives us this routing functionality and now Angular knows our routes. The missing piece is someplace to render the currently selected component because if you have a look at our app component template, we still add our components here with their selector. Now OK we visit /users, so Angular knows we want to load the users page but how does it know where to display it?

How does it know if you want to display it here or here or maybe here in the header, it doesn't know. So the right place to inform where to load it is to get rid of all these rows here:

*<!-- <div class="row">*

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

*<app-home></app-home>*

*</div>*

*</div>*

*<div class="row">*

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

*<app-users></app-users>*

*</div>*

*</div>*

*<div class="row">*

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

*<app-servers></app-servers>*

*</div>*

*</div> -->*

but one, so only have the place where you actually want to render your route and now you don't add your component with its selector but you add a special directive shipping with Angular, the router-outlet.

  <div *class*="row">

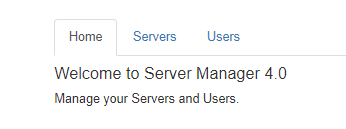
    <div *class*="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

      <router-outlet></router-outlet>

    </div>

  </div>

Now this looks like a component but it still only is a directive, keep in mind directives may have any selector including this component or this elements style selector. And with this, this now simply marks the place in our document where we want the Angular router to load the component of the currently selected route. And with this all set up, if we now save this and let it rebuild therefore, we should see that this rebuild successfully and if we visit our page now, you see we only see the home page now



and if we now enter /users here, we see the users page and if we enter /servers, we see the servers page. So this is working, of course our links are not working, this is something we will work on next.

**#### 127: NAVIGATING WITH ROUTING LINK**

In the last lecture, we made our routing work, we added routes and we gave Angular a place to load our currently active component but one issue we have is right now we can only navigate around by manually typing it here in the address bar.

Not super convenient and while it is important to be able to load a route directly from the address bar, it would also be nice to have some working links here in our navigation to have these tabs work. So let's add some links to our application and if we go to our app component, we see here's our navigation. The first thing you might think about is, hey we can or we already have this **href** attribute here to be precise, so here we could have **href**, well just slash to load our root route, here we could have **href=”/servers”** and here we could have **href=”/users”** and if we now save this, we added all the routes, the paths we set up in our routes configuration to our navigation here and hence if this reloads and we click the tabs, hey this is indeed working, now we are correctly loading the route we want to load.

OK this tab is not updated but I will come back to this later but the reloading is working and that's the issue, it's reloading the app. Watch this reload icon here, do you see this? We refreshed the app with every link we click and that's the natural behavior because with every link we click, a new request gets sent to the server and it returns as a new page and since this page is still our Angular app with the routes registered on it, it is able to give us the correct route.

So the same thing happens as if we enter this manually here. That however is not the best behavior because it means it restarts our app on every navigation, that of course means our whole application state will be lost and it might not really be the behavior we want to offer to the user.

So this is not how you should implement navigation. How should you implement it then? There is a special directive Angular gives us, let's get rid of all these ref attributes here, so this is not how we will navigate around. Instead let's use this special directive, it's called **routerLink.**

Now routerLink, like this simply is able to parse a string where we could pass just slash ***( routerLInk=” / “ , instead of href=” / “ ),*** so just this string slash to routerLink here. Now this will tell Angular that this element on which routerLink is placed here, this anchor tag here will serve as a link in the end but it will handle a click differently as you will see in a second. Let's now also add this to the service links, so routerLink equals /servers and I will come back to how you may write the path here in the next lecture. Another way of using routerLink is with property binding, so you can enclose it in square brackets and now of course you can't just pass /users here because this would now search for a property with that name which would even be an invalid name in Javascript.

So now you have to pass a string here with single quotation marks or better, an array which gives you a more finegrained control over routerLink and I will come back to this later when this is especially useful. Here in this array, you now specify all the segments of your path as elements in this array. So the first segment in this case is only a string/users and if you had /users slash something, you would have a second element here which is then just something without a slash.

**$ <a [routerLInk]=”[‘/users’, ‘something’]”> Users </a>**

The leading slash here is only needed to make this an absolute path and I will come back to this in the next lecture.

For now, I will get rid of this, now this array notation is not super convenient but soon you will see when it really gives you an advantage or when you need this notation, it allows you to construct more complex paths very easily. So with this, we set up our three links using routerLink either by passing a string or this array which allows us to define our individual path elements. With that if we save this and have a look at our application, now you see it's still reloads or it still gives us the components but it doesn't reload the

page. If you watch this reload icon, nothing is happening there because routerLink catches the click on the element, prevents the default which would be to send a request and instead analyzes what we passed here to the routerLink directive, so this path or this array of path elements and then parses it and checks if it finds a fitting route in our configuration which it of course does because we defined all the paths we're using here.

And this is how we can navigate around with a routerLink and this actually is how we should navigate around because it gives us the better user experience, it doesn't restart our app, therefore it keeps the app state and it's much faster than reloading the page all the time. 3You can of course still type something here manually, so that will still work but if you are inside of the app, using routerLink is much better.

**#### 128. UNDERSTANDING NAVIGATION PATHS**

In the last lecture, we added this navigation and I also mentioned that you can change the way you use your paths here. Right now, we use //servers or even with this more complex notation, we have /users as the first element. Now what happens if we just use servers here, like this and just use users like this?

**$ <a [routerLInk]=”[‘users’]”> Users </a>**

Let's save this without the leading slash. Now if we click there on our links, you'll still see this works absolutely fine, nothing changed.

Now let's go to our servers.component template file here and for now, let's quickly add an anchor element here where we also add a routerLink where I add servers again, Reload Page would be a fitting description.

<a *routerLink*="servers">Reload Page</a>

So on the servers component, I now added this routerLink where I basically want to make sure that I stay on the servers page which I'm already on.

So if we go to servers here, we see the reload page here and if I click it, you'll see we got an error. This error here occurs because it doesn't find a route which is servers/servers. Now this error won't occur if on the servers component, I turn this to /servers, so to an absolute path, it was a relative path. Here if we now have our console still open and I click reload page, nothing occurs here, nothing, no error occurs because now it simply navigates to localhost:4200/servers as you can see down there and if we turn this back to a relative path, the difference was that now it tries to go /servers/Servers.

So with a relative path, it always append the path you specify in the routerLink to the end of your current path and important here, the current path depends on which component you are currently on.

That is why we could remove that in our navigation where we also use relative paths because that is in our root component, this component is not loaded through the router, so this root component always sits at localhost:4200 slash nothing. So this is always at our root level which is why we can use relative paths here too but one layer below that once we loaded a route, for example the servers component, this page is only loaded if we are at /servers.

So if we add a relative route here, this will now get appended to this route we are on, so it will try to load servers/servers and therefore it gives us an error in our case because we haven't registered a servers/servers route. Now of course using relative paths inside of active components, active routes might be a nice thing if you have nested routes, right now it will only break the app though.

Now you can also use relative paths with ./ at the beginning, this is the same as having nothing there

and you can even navigate around like in a folder directory, you could add ../ which means go up one level and then add this to the path here. Now if we do this inside of our a servers component, you will see if I hover over this, now it again only tries to load /servers and therefore, we don't see an error because now it first goes back one level which basically is to just our root paths, so it removes one path and important path depends on the route you're on.

So if you were on servers slash something and this would load this component, it would remove both,

so this does not necessarily only remove one segment here, it removes the currently loaded segment you could say. So in this case, it removes this and then adds /servers again which is why this works and this is how you can navigate around. Absolute paths with a slash at the beginning which will always get appended to the root domain, relative paths without a slash or with ./ which means added to the currently loaded path or ../ and you can even add more of them to say well go back ones path, go back

another path, so always seen from the currently loaded component on and if that is something which is not 100% clear, you will see throughout this module but also feel free to play around with it, add your own routes, add some nesting, add more complex paths and see how this works out.

But for now, let's continue with our application, I will remove this reload page link and let's enhance this a bit more.

**#### 129. STYLING ACTIVE ROUTER LINK**

We of course added navigation to our page but one issue is that whilst it does work fine, we get no visual indication of what the currently active route is. It would be nice if the selected tab would change and that of course is only a CSS thing because if you check your app component here, you'll see we get this **active class on the list item** which determines whether this is marked as active if this link gets the right CSS classes or not. So it would be nice if we could dynamically set this class.

I will remove it from here and how can we now set it dynamically? **( class=”active”)** Angular gives us a specific directive for this, the **routerLinkActive** directive and the convenient thing about this directive is, you can add it to a wrapping element, like we do here, to the list item or also of course to the link itself and it will attach the class you specify in between here, for example active, so this should be a string, to that element it sits on.

**$ routerLinkActive=”active”**

Now active on the link itself wouldn't do anything when using Bootstrap, here for these tabs look we need to add it to the list item but this of course depends on which package or which CSS code you are using and here

<li *role*="presentation"><a *routerLink*="/" *routerLinkActive*="">Home</a></li>

you could specify any class you want, like my active class or whatever that is. So here, it is active to make it look nice in Bootstrap's world and I will add it to all my links here.

So now theoretically, all my links should receive this active CSS class

<li *role*="presentation" *routerLinkActive*="active"><a *routerLink*="/">Home</a></li>

        <li *role*="presentation" *routerLinkActive*="active"><a *routerLink*="/servers">Servers</a></li>

        <li *role*="presentation" *routerLinkActive*="active"><a *routerLink*="/users">Users</a></li>

once they are clicked or when they are active. So if we have a look at this, it seems to work because we still have homes set as our active page, even though we remove this class, this active class we had there before. If I click servers, it gets marked as active too and the same for users, so this works but somehow, home is always marked as active.

So that does not seem to work correctly, what's wrong here? Well the routerLink active directive does one thing, it analyzes your currently loaded path and then checks which links lead to a route which uses this path and by default, it marks an element as active, it marks this CSS class if it contains the path you are on or if this link is part of the path which is currently loaded.

So for the /servers link or the /users link, this is of course the case once we click servers or users. For the slash nothing link, which is the first one, the home page, this is always the case, we always have slash nothing at the beginning of our path here, so if we are on /users, well there is this empty path in-between you could say, the same for servers and the same for just home of course.

So the empty path segment is part of all paths, still we don't want to have the home being marked as active all the time. So to fix this, we can add some configuration to this routerLink active directive and I will split this up all over multiple lines to make this a bit easier to read. So we can add a special configuration to our routerLink active directive, so on the same element as routerLink active was added to, we can add routerLink active options and this now needs some property binding because we don't just pass a string there, instead we pass a Javascript object and that would not work without having our square brackets.

 <li *role*="presentation"

*routerLinkActive*="active"

*[rounterLinkActiveOptions]*="{}"

        ><a *routerLink*="/">Home</a></li>

So with square brackets, we can pass anything which will be resolved dynamically, like this Javascript

Object and here, we can basically add one important configuration, say exact and set this to true.

$ **{exact: true }**

  <li *role*="presentation"

*routerLinkActive*="active"

*[rounterLinkActiveOptions]*="{exact: true}"

        ><a *routerLink*="/">Home</a></li>

So exact is a kind of a reserved property on this object you passed to routerLink active options and this will now basically tell Angular only add this routerLink active class, so this active CSS class in this case, if the exact, the full path is well whatever does link leads to, so only if everything is just slash and not if it is only part of the path. So with this in place, now you see it still is marked as active on the empty path, at the starting page but if you go to /servers, now only servers is marked as active because now we fixed this, we change the default behavior to make sure we only mark this if the full path is whatever we have on our routerLink.

#### 130. NAVIGATING PROGRAMMATICALLY

Well we already covered quite some things. We learned how to add routes and how to load them, either by typing it manually into our navigation bar here or by clicking some links, we also learned how to style these links. But what now if we wanted to load a route programmatically?

So we don't have a link the user can click but we've finished some operation or the user clicked some button and then we want to trigger the navigation from our TypeScript code, we can do this. Let's say in our home.component here, I add a new button to it and on this button, I simply want to load the server, the servers page.

$ <button>Load Servers </button>

So here, now we could try adding routerLink but let's say I want to have a click listener and execute some method because we do something else than just navigating there. I will also add some CSS classes to make this button look nice. So on this click listener, I want to execute onLoadServers and here,

<button

*class*="btn btn-primary"

*(click)*="onLoadServers()">Load Servers </button>

this method now gets implemented in the home component and in onLoadServers, I now, still again I simply want to navigate to the servers component. So we could for this example of course also use a routerLink but let's say here, we have some complex calculation

onLoadServers(){

*//complex calculations*

}

or we reach out to our back-end, we store something on the server and once we are done, now we want to navigate away. To do so, we somehow need to get access to our router, this Angular router because we need to tell it hey please navigate somewhere else.

The nice thing is we can inject this router. So let's bind it to a private property, maybe we name it router and this will be of type router, router needs to be imported from @angular/router.

constructor(private *router*: Router) { }

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

With this injected, we can use this router here, so this router and then we get a couple of methods there, one of the most important ones being **navigate.** Now navigate takes an argument which allows us to guess what? Navigate to a new route and here, a route is defined as an array of the single or the different elements of this new path.

onLoadServers(){

*//complex calculations*

**this**.router.navigate([]);

}

So just like here in app component where I also explain this alternative way of binding routerLink, binding

it to an array where you could have users for the users part of your path and then another element for maybe something, for the something part of your path **( [routerLink]=”[‘users’]”** ), now you always have to pass such an array when using the navigate method but in here still, the first element is simply the first element or the first segment of your path.

So if let's say we want to go to /servers here, we could add /servers here and as before, this is now an absolute path, you could have a relative one but here, you have to control to what this should be relative to and I will come back to this. So for now, let's make it an **absolute path** here and once we did this, let's check our page here, click this button and we go to servers.

onLoadServers(){

*//complex calculations*

**this**.router.navigate(['/servers']);

}

**So this is now programmatically routing to a different page, still it doesn't reload our page, it does the same as if we clicked a routerLink but with this router navigate method, we are able to trigger this programmatically, so trigger this in our code.**

**#### 131. USING RELATIVE PATH IN PROGRAMMATIC NAVIGATION**

In the last lecture, we learned how to navigate programmatically, now I want to also have a look at how we can use relative paths there. So back in our servers.component.html , let's say we have again our reload page link but this time it's a button here with class btn button primary and still we might use this approach because we want to do some different things.

<button *class*="btn btn-primary">Reload Page</button>

So reload page and here, I will have my click listener, onReload maybe.

<button

*class*="btn btn-primary"

*(click)*="onReload()">Reload Page</button>

So if we now go to our TypeScript code, here onReload will simply or will be the method which gets triggered and we already inject this serversService here, I also want to inject the router, so let's add router here, make sure to add the import from @angular/router.

 constructor(private *serversService*: ServersService,

              private *router*: Router) { }

So we learned that now we could say this router navigate and now let's add /servers here.

 onReload(){

**this**.router.navigate(['/servers'])

  }

If we do this and we save this page and we go to the servers page and click reload, nothing happens, we don't get an error because we are already on that page. Technically it still reloads the page but keep in mind, it never sends a request so this reload icon will never spin but Angular simply determined that we are on this page, so no further action is needed. Now let's say we remove the slash at the beginning,

$ **this.outer.nativate([‘servers’])**

so we turn this into a relative path and we still are on the servers component, keep this in mind.

So now if this reloaded and we click this page, we still don't get an error and now this might be strange because before when we use the routerLink, you saw that we did get an error because it would try to add servers to the end of this path and of course this route now doesn't exist.

So why does it work fine if we use the navigate method? Because unlike the routerLink, the navigate method doesn't know on which route you are currently on, so to say. The routerLink always knows in which component it sits, in which components template and therefore it knows what the currently loaded route is. Now you could argue that here we also call this inside the component TypeScript code, so it could determine where it is in but it's just not how it works.

So to tell this where we currently are in, we have to pass a second argument to the navigate method which is a Javascript object and here we can configure this navigation action. Now I will come back to this later because there are more things we can add but one configuration is the **relativeTo** property here. Here we define relative to which route this link should be loaded and by default, this is always the root domain, which is why we didn't get an error for this servers path.

$  onReload(){

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

  }

Here we have to give a route though, so we don't pass a string here, instead the route is something we can inject here too. We can get the currently active route by injecting route which is of type ActivatedRoute and make sure to import ActivatedRoute from the @angular/router package too.

constructor(private *serversService*: ServersService,

              private *router*: Router,

              private *route*: ActivatedRoute) { }

Now ActivatedRoute like the name implies simply injects the currently active routes, so for the component you loaded, this will be the route which loaded this component and the route simply is kind of a complex Javascript object which keeps a lot of meta information about the currently active route.

Now we can set this as a value, so this route, this injected route for the relativeTo property and now with that extra piece of information, Angular knows what our currently active route is and theoretically you could also set this to some other route which you would have to get by some way. So here, we are telling it now our currently active routes is this ActivatedRoute, so relative to this route you should navigate and then it will simply resolve all the relative paths you might have here relative to this route.

  onReload(){

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

  }

So with this change in place, if we now click this button again, now we broke the app because now it again tries to find servers/servers which doesn't exist. So great we broke the app but with that, you should understand how you can work with relative paths in the navigate method you defined relative to what this paths should be loaded, the default is the route URL, that might be the behavior you're looking for, it might not be the behavior you're looking for, you can change it with this extra Javascript object, second argument to the navigate method where you have the relativeTo property you can set to the route you want to navigate relativeTo. For now, I will simply comment this out to disable this not working button and still keep the code in this example project.

**#### 132. PASSING PARAMETERS TO ROUTES**

With the basics about routings set, let's enhance our application a bit by going back to the app module and adding some other routes to it. Let's say that besides our users route here,

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

we also want to be able to load a single user, so to load the user component which simply well gives us that single user. For that, we might want to load a user dynamically because if we have a look at our users component, we see we get three users with different IDs, so it would be nice if we could pass the ID of the user we want to load in that route path.

One approach would be to set up a route with user/1 and then one with user/2

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

  { path: 'users/2', component: UserComponent },

and it doesn't look very optimal because it's not very dynamic and still, how would we know inside of this component which route was loaded? Okay we could parse the path manually but that's not really a great approach. Instead, we can add parameters to our routes, dynamic segments in our paths. We do this by adding a colon and then any name you like, like for example ID, you will later be able to retrieve the parameter inside of the loaded component by that name you specify here, so by ID in this case and the colon simply tells Angular that this is a dynamic part of the path.

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

Without colon, only routes which are users/id and with ID, I literally mean the word ID, would lead to this component, with a colon user, slash anything else would load this component and anything else would be interpreted as the ID. Now with this in place, we are able to load a component, the user component with this dynamic piece sent to it.

So if we have a look to this user component or at this user/1 component, here we simply say user

with ID and now we have a dummy text here loaded and username with some dummy text here.

<p>User with ID \_ID\_ loaded.</p>

<p>User name is \_NAME\_</p>

Now for now, we won't touch this, we won't use this parameter, we will do in the next lecture, let's now see if we successfully reach this. So if we save this to let it recompile and we go back and we are in the users,

let's add slash “something” **( localhost:4200/users/something )** here and you see, the single user component was loaded, the same if we add /1 here. So this thing, the segment after /users is indeed interpreted as dynamic, otherwise we would get the error that users/1 is an unknown route.

It isn't because we set it up to be a known one because again, we added this dynamic path segment here. Now this is how we can dynamically load some route, how we can encode some data into our path, how can we get access to this data in the loaded component though because that of course is another key question. Let's take a closer look in the next lecture.

**#### 133: FETCHING ROUTE PARAMETERS**

In the last lecture, we created our route with a dynamic path segment, now we want to get access to the data the user sent us or which is encoded in the URL I should say. So we know that we will load the user component, user component, this one here and we know that there will be some data in the URL for us.

How can we get access to it?

Well, it is in the TypeScript file where we can get access and there, we need to inject something we already injected before, the active route. So this is again on ActivatedRoute, so make sure to use this type and import it from @angular/router and by injecting this, we get access to the currently loaded route.

 constructor( private *route*: ActivatedRoute) { }

Now I told you that this currently loaded route is a Javascript object with a lot of metadata about this currently loaded route, one of this important pieces of information is the currently active user. Now you see that in this user component, I already defined a user object at the top which is undefined for now, it should have the following structure and it's not used right now but we could load our user by simply getting access or retrieving this parameter from our URL.

So it would be nice if let's say in ngOnInit when our component gets initialized, we get our user. So we maybe want to say this user equals and now we assign it to a Javascript object because that is the type of it, a Javascript object with an ID and with a name. Now the value for the ID can be fetched from our route and there, we have a snapshot property and on this snapshot of our currently active route, we have a params Javascript object and here we can get our ID and now you will only have the access to properties here which you defined in your route parameters.

So this part here, we named it the ID here so we can retrieve the ID from this params object here. Now the name is something which is not encoded in the route right now, so maybe we should do this too. Let's go to app module and let's simply add another dynamic part here, :name.

 ngOnInit() {

**this**.user = {

      id: **this**.route.snapshot.params['id'],

      name: **this**.route.snapshot.params['name']

    }

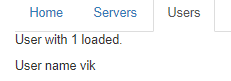
  }

Now we also will get this as a parameter, so we can retrieve it the same way we retrieve the ID, let's retrieve the name and it's in the same params object because it's all part of the route which was loaded due to this path we set up in app module being triggered which holds two dynamic pieces and you can add as many parameters here as you want, they will all be retrievable on this params object of the loaded route. With that, we assign something to our user, now we could go to our HTML document and output that data. So with string interpolation, here we could output user ID, it will now be set, we know that and username maybe, both will be available because we set it in ngOnInit.

<p>User with {{user.id}} loaded.</p>

<p>User name {{user.name}}</p>

Now this should work, so if we save this and we target /users/1/Vik maybe so that we have both the ID and the name and we hit enter, we correctly see ID one, name Vik,



if we change the ID to three, we see ID three here. So this seems to work and you even see that this is still marked as active because this currently active path still contains /users which was the route for this link, so this routerLink active directive is still doing its job. Now this seems to be working fine, now let me add something to it which will break it though. I'll do this in the next lecture.

**#### 134: FETCHING ROUTE PARAMETERS REACTIVELY**

In the last lecture, we saw how we can retrieve our route parameters and this is working fine but there are ways to break this, there are cases where this approach will not work and I will show you a typical example. On our users component, we right now always saw that we have the user ID and name we passed on our URL.

Now in here, let me quickly add a routerLink, so I'll add a horizontal line and then add an anchor tag where I remove the ref and instead say routerLink and now I will actually use this syntax with the square brackets where I pass an array and I want to load /users, an absolute path to load the correct route. The second element will be the ID, so this might be let's say 10, could be a number, could be a string and then the name, let's say Anna. So this will now construct a route which is /users/10/Anna and this is where this array approach might be better because you can really structure this really well and we will soon see how you can even make this more dynamic.

<a *[rounterLink]*="['/users',10,'Anna']"></a>

So here, I will say load Anna, ID 10 something like that,

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

any text you like and now let's see this here. We get a link on our currently loaded page with users 3 Max and if I click this, you'll see that the URL was updated. Now it's users 10 Anna but the text here wasn't and this is not a bug, this is the default behavior and what is happening here, do you have an idea? The behavior here as I just mentioned is not a bug. We load our data by using this snapshot object on the route.

Now if we load a new route, what happens? Angular has a look at our app module, finds the fitting route here hopefully, loads the component, initializes the component and gives us the data by accessing the snapshot here. Now that only happens if we haven't been on this component before but if we click this link, which is on the user component, well then the URL still changes but we already are on the component which should get loaded.

So Angular cleverly doesn't really instantiate this component, that would only cost us performance,

why would it re-render a component we already are on? Now you might say because the data we want to load changed but Angular doesn't know and it's good that by default, it won't recreate the whole component and destroy the old one if we already are on that component.

Still of course you want to get access to the updated data and you can. It's fine to use this snapshot for the first initialization but to be able to react to subsequent changes, we need a different approach. In our TypeScript code, in ngOnInit here, let's say after we assign this initial set up, we can use our route object and instead of using the snapshot here, there is some params property on this route object itself.

Now we didn't use that before, we had the snapshot in between, what's the difference? Params here is an observable and now observables is something I will come to right after this section where we will take a closer look. Basically, observables are a feature added by some other third-party package, not by Angular but heavily used by Angular which allow you to easily work with asynchronous tasks and this is an asynchronous task because the parameters of your currently loaded route might change at some point in the future if the user clicks this link but you don't know when, you didn't know if and you don't know how long it will take.

So therefore, you can't block your code and wait for this to happen here because it might never happen.

So an observable is an easy way to subscribe to some event which might happen in the future, to then execute some code when it happens without having to wait for it now and that is what params is. It is such an observable and as the name implies, we can observe it and we do so by subscribing to it. So you can call the subscribe method on it and this is still called on params, I'm just bringing it in a new line to make it easier to read and now subscribe can take three functions you pass here as arguments.

**this**.route.params

      .subscribe(

      );

Now the first one is the most important one here, it'll be fired whenever new data is sent through that observable, so to put in easier words, whenever the parameters change in this use case. So then this first argument which should be a function here is executed, I will use an ES6 arrow function and here, we will get the updated params as an argument and you can set this up type params, doesn't really make a difference, it has to be imported from @angular/router then though but anyways, params will always be an object just like here on the snapshot which holds the parameters you defined in the route as properties.

**this**.route.params

      .subscribe(

        (*params*: Params) =>{ }

      );

So in the function body of this arrow function, so of the function which gets executed whenever the parameter changes, here we can now update our user object and we can say this user ID should now be params ID, referring to the params we passed here and the same of course for the name, with a semi-colon at the end of course.

**this**.route.params

      .subscribe(

        (*params*: Params) =>{

**this**.user.id = params['id'];

**this**.user.name = params['name'];

        }

      );

So this will now update our user object whenever the parameter change. That's why its inside of this callback and of this anonymous function passed to the subscribe method here. This code will not be executed when ngOnInit runs through, this code here will, so the subscription will be set up but only if the parameters then change, only in this case we will exchange them in our user object. That is why with this change in place, if we now go back to ( localhost: 4200/users/3/Max), we see 3 and Max is displayed here but if I now click this link ( Load Anna (10) ), it correctly updates here ( localhost: 4200/users/10/Anna), too because our observable fires and we then retrieve the updated parameters and assign them to our user object and this therefore actually is the approach you should take to be really safe against changes not being reflected in your template.

Now if you know that the component you're on may never be reloaded from within that component as we're doing it here, then you might not need this addition, you might simply use the snapshot because if you know your component will 100% of the time be recreated when it is reached, because there's no other way of reaching it, there is no way of reaching it whilst being on that component, if you know that, you don't need to subscribe. In all other cases, make sure to use this approach to get informed about any changes in your route parameters.

**#### 135: AN IMPORTANT NOTE ABOUT ROUTE OBSERVABLE**

In the last lecture, we learned how we can subscribe to our params to update them or to react to any

changes to them and update our page if we already are on that page. Now we set up this subscription using observables and this works fine and you don't have to change anything here, I want to bring something to your attention though, the fact that you don't have to add anything else to this component here simply is because Angular does something for you in the background which is super

important, it cleans up the subscription you set up here whenever this component is destroyed because if it wouldn't do this, what are you doing here? You're subscribing to parameter changes and let's say you then leave this component and later you come back.

Well once you left, this component will be destroyed and when you come back, a new one will be created but this subscription here will always live on in memory because it's not closely tied to your component, so if the component is destroyed, the subscription won't. Now it will be here because Angular handles this destroying of the subscription for you but theoretically, you might want to implement the **onDestroy** lifecycle hook, importing it from @angular/core, therefore you have to implement **ngOnDestroy** and then you could store the subscription here in some property, let's name it

params subscription which will be of type subscription and subscription needs to be imported from

rxjs/subscription.

export class UserComponent implements OnInit, OnDestroy {

Rxjs is the package offering all these observables functionality, as I mentioned it's not shipping with Angular but Angular is using this package. So this type here, this subscription, this params subscription property is now bound to this subscription we're setting up here when calling subscribe,

 paramsSubscription: Subscription

this returns us the subscription and once the component gets destroyed, we could now access the subscription and unsubscribe.

**this**.paramsSubscription = **this**.route.params

      .subscribe(

        (*params*: Params) =>{

**this**.user.id = params['id'];

**this**.user.name = params['name'];

        }

      );

ngOnDestroy(){

**this**.paramsSubscription.unsubscribe();

}

And again because it's important, you don't have to do this, you can leave it as it was before because Angular will do this for you regarding these route observables but if you add your own observables and I will come back to this in the observable section following this section, you have to unsubscribe on your own and I just wanted to bring it to your attention because it's always super important to understand what's happening behind the scenes. Again, it's not necessary here but it doesn't hurt to also do it manually too here, it won't add anything bad to your app by doing it like this.

**#### 136: PASSING QUERY PARAMETERS AND FRAGMENTS**

In the last lecture, we learned how we can retrieve our route params. That's awesome and this is super important and something you will probably use a lot. Now there are more things you can add to your URL though. You might have query parameters, these are the parameters separated by a question mark, like mode equals editing, anything and you can have multiple ones separated with @ signs.

**$ localhost:4200/users/10/Anna?mode=editing&**

The question now is how you can pass them using Angular links, routerLinks and how can you then retrieve them?

You might also have a hash fragment, so separated with the hash sign ( **?mode=editing#** ) to jump to a specific place in your app, loading for example ( **?mode=editing#loading** ) or maybe you don't really want to use this to jump there, you would have to add this scroll down there behavior on your own, maybe you just want to retrieve it to use it as some extra information. Let's take a closer look at how we can pass this extra information on our links, both when using the routerLink directive and the navigate method in code and how we can then retrieve that information. Let's start by passing it first.

Let's say in our app module, we want to add more routes. For our servers here, we also want to add one other route and that would be the route which allows us to edit a certain server, so we might load a server with some ID and then have slash added at the end to describe in the URL what will happen in this component and here it would be the edit-server.component we want to load.

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

So we added this new route which should load this component in this edit server file,

now to be able to load this route on my servers component here, I want to hook up these links here

      <a

*href*="#"

*class*="list-group-item"

*\*ngFor*="let server of servers">

        {{ server.name }}

      </a>

In this list, we still have to do the same for the users component later.

So here, I will add my routerLink and I will use the square brackets syntax again so that I can pass an array here because I want to go to /servers, then let's say a server with ID 5, we will soon learn how to make this more dynamic and then also add it, so this is this newly created route I want to call in this case.

<a

*[routerLink]*="['/servers', 5, 'edit']"

If we save this and we go to servers and click on this, you'll see we loaded servers/5/edit and the

same for every link of course, again I will change this soon. Now let's say we also want to have some query parameter deciding on whether we are allowed to edit the server or not.

So we want to add this questionmark and we don't do it by adding questionmark equals something here

as another element in this array ( **‘?=’ ‘** ), instead when using a link in the template, we add a new property of this routerLink directive we can bind to the query params property.

Now query params and that's important is not a new directive, it's just another bindable property of the routerLink directive and here we have to pass a Javascript object and on query params, we can now define key-value pairs of the parameters we want to edit. So for example say allowEdit here and set this to one, now let's see what happens if we now click this link.

 <a

*[routerLink]*="['/servers', 5, 'edit']"

*[queryParams]*="{allowEdit: '1'}"

If we now click any link here, you'll see we have questionmark allowEdit 1 here, so that's the key-value pair we set up in this object and of course you could have more key-value pairs here which would then be separated with an and sign in between in your link.

$ [**http://localhost:4200/servers/5/edit?allowEdit=1**](http://localhost:4200/servers/5/edit?allowEdit=1)

So that's the query params property, we also have the fragment property and that is only fragment because you may only have one fragment. Here, you could simply pass a string with single quotation marks or therefore, omit the square brackets and set this to let's say loading.

      <a

*[routerLink]*="['/servers', 5, 'edit']"

*[queryParams]*="{allowEdit: '1'}"

*fragment*="loading"

So if we now save this and we go back to the servers page, now you see we have hash loading at the end, this is how you can add a fragment. So super easy to add in our routerLink here in our template, let's now see how we can do the same programmatically.

$ [**http://localhost:4200/servers/5/edit?allowEdit=1#loading**](http://localhost:4200/servers/5/edit?allowEdit=1#loading)

So in our home component where we already have this load service button, let's say we want to load server 1.

<button

*class*="btn btn-primary"

*(click)*="onLoadServers(1)">Load Servers </button>

So here I will change this name to load server and pass one as an argument maybe, adjust this method here, load server and the ID will be passed as an argument as we just saw and here I therefore want to navigate to servers, pass the ID, we're already seeing how this gets more dynamic, now we're not hardcoding 5 or something like this, we're using this argument here instead and then as a last argument, edit.

Now again, we still want to add query params and this fragment. So here in the navigate method, we use this object we already saw where we already added relativeTo. Now I don't need relativeTo here,

instead here I can also set up my query params and that again is then just a Javascript object where we say allowEdit and set this to one maybe and just as I add query params, I can also add the fragment here and set this to loading.

onLoadServers(*id*: number){

*//complex calculations*

**this**.router.navigate(['/servers', id, 'edit'], {queryParams: {allowEdit: '1'}});

}

onLoadServers(*id*: number){

*//complex calculations*

**this**.router.navigate(['/servers', id, 'edit'], {queryParams: {allowEdit: '1'}, fragment:'loading'});

}

Now with this if we save this and we go back to our home page and click load server, you'll see we add the query parameter, allowEdit and set it equal to one and the hash fragment here. So that is how we can use query parameters and fragments on both the programmatical routing approach, calling it from the TypeScript code or on the routerLink, like here. Now let's also learn how we can retrieve the data

**#### 137. RETRIEVING QUERY PARAMETERS AND FRAGMENTS**

In the last lecture, we learned how we can pass parameters and fragments, now let's make sure we can also retrieve them. So we're loading this edit-server.component in the end, so that is probably where we want to retrieve that information. We can easily do this by going there, edit server component and you see there already is a lot of code in there, well everything what's happening here however is it's using our serversService to get the server with the ID one and to be able to update it and definitely dive into this serversService file if you want to learn more about what's happening here.

It's super simple though, we simply are able to get this array of servers, get a single server or update

a single server, so overwrite it. So that's not the interesting part, the interesting part in the edit component here is how we can get access to our query parameter and the fragment.

We need our ActivatedRoute for this, so let's inject it just like we did before, ActivatedRoute needs to be imported from the @angular/router package, so make sure to add this import at the top and with this added and let's say ngOnInit, we can retrieve our query parameters and the fragment and just like before, there are two ways of retrieving it.

constructor(private *serversService*: ServersService,

              private *route*: ActivatedRoute) { }

For the first approach, we can simply access the snapshot of our route and access query params here or our fragment, just like that.

  ngOnInit() {

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

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

Now this might bring the same problem as with the params, this is only run or updated at the time this component is created. So if there is a chance of changing your query params from the page you're currently on, you might not want to use this approach because it won't be reactive, it won't display or allow you to react to any changes which happen after this component has been loaded.

So for now, let's use the snapshot though and let's watch the console, once this reloads, notice we already have a query parameter and a fragment and we see both here. This is our query parameter object where we have this key-value pair mapping again, allowEdit to one and this is just a bit broken but here you see if I make my console a bit broader, you see the loading hash fragment.

So this is how you can retrieve it. The alternative of course is to use the route and just like we had params as observable, we also have query params as an observable we can subscribe to, like this and of course we also have fragment as an observable we can subscribe to. So just like before, this will now allow you to react to changed query parameters.

**this**.route.queryParams.subscribe();

**this**.route.fragment.subscribe();

Now that's something we'll use later in this module, for now nothing will happen here but it is important to keep in mind like for params, you do have the same for query params and fragment. You don't need to unsubscribe here, Angular will handle it for you just like it did for params but this is how you can now get access to these extra features and how you can make sure that you don't miss that data.

**#### 138. PRACTICING SOME COMMON GOTCHAS**

So now that we learned a lot about navigating passing parameters, let's improve our app a little bit by going to the users component where we have a list of links in the end to single users and let's add the routerLink here,

 <a

*[routerLink]*="[]"

pass an array as an argument and here, we want to target /users and we already created our route, right?

We have /users, /ID and /name, so we can use that information here in users component, the second element here in this array there should be the ID and since we have a for loop where we loop through all our users, we can easily dynamically add the ID here by accessing user ID and the same for the name,

the third element, the third segment in this path, the user name, we can access username. This is how we can dynamically construct such a link and you already saw this with the navigate method a couple of lectures earlier.

<a

*[routerLink]*="['/users', user.id, user.name]"

So this is our routerLink here, let's do the same for the servers. So here in the servers component, let's also replace the five here with server ID.

*[routerLink]*="['/servers', 5, 'edit']"

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

We do have a server ID if we have a look at the ServersService, you see we have IDs and names here.

So we want to pass the ID and we can now do this by dynamic fetching it here and I will remove edit

though because I not only want to load the edit component here, I want to load the single server component instead.

*[routerLink]*="['/servers', server.id]"

With my link added here, we need to add the fitting route, so in our app module here, we need to add another route where we have servers and then just ID to load a single server with the server component here.

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

Right now, we're always loading the edit-server.component so make sure to add this extra route and of course, this will be loaded because we remove the edit at the end, so we're not editing this anymore, this just loads the single component.

And of course now, in this single server component, this one here where we want to output our server data, we need to get that data we pass in the route path in the URL.

<h5>{{ server.name }}</h5>

<p>Server status is {{ server.status }}</p>

We pass the ID there, we do have the ServersService where we are able to get a server by ID and we already do this in our server component here in ngOnInit, right now always for the server with the ID one. So now, use your chance to update this code to do this with the ID being passed with our route. You learn everything you need for this, time for you to do it on your own, I will give you a chance to pause this video before we do this together.

So were you successful? Let's try it together. Of course to get access to the router parameter, we need

to inject our activated route. So we learned how to do this, add this route property and make it of type ActivatedRoute which you need to import from @angular/router

constructor(private *serversService*: ServersService,

              private *route*: ActivatedRoute) { }

and now in ngOnInit, you can get your ID from this ActivatedRoute from the snapshot, the params object here and then the ID which is encoded in the URL and now you can use this ID here to get that server.

Now if you also want to react to any changes thereafter, you need to subscribe to your params observable here and basically do the same. You get the params here which is of type params, make sure to import that from @angular/router and once you get this, you can get a new server whenever your params should change.

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

So here, you could then access params ID like this to update the server whenever change is triggered on this page. So with this in place, if we reload our app and go to servers, you'll see we get an error here which makes sense because keep in mind on our servers component here, we still are always loading our single server component and therefore, we are also loading it

if we don't have this ID available.

  ngOnInit() {

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

**this**.server = **this**.serversService.getServer(id);

**this**.route.params

      .subscribe(

        (*params*: Params) => {

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

        }

      )

So for now I will comment this out, ( **// <app-server></app-server>** ) we will soon learn how we can nest another router in there to have some kind of child routing, to load this route dynamically next to our menu here. So for now, I simply comment it out. If I now click on a server here, we still get an error and now this can be a tricky one to spot, what's going wrong here? The issue is if we parse a parameter from our URL, it will always be a string because our whole URL is simply just text, so ID here will actually not be one in the case of /1, it will be a string of one. So if we try to get a server by this ID, it will actually search a server which has an ID of one but a string one, however we use numbers here.

So we simply have to make sure to convert our ID to a number and we can do this by adding a plus in

front of it and this is perfectly fine to do.

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

With that, you're making sure that the ID you're retrieving is treated as a number, of course you also need to make sure to do this here in your observable, so convert the params here to a number too.

.subscribe(

        (*params*: Params) => {

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

        }

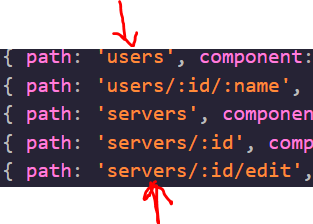
      );

So with that if we now click on servers and choose a server, you see production server here, test server in this case and with that, this is now working as expected.

We now improved our app because now we got working links here, we get working links here and we're loading the correct server.

So with that, we added a lot but we see that our app has some limitations, it would be nice if we don't load a new page with the user or the server here but load it next to our menu in both cases. So we somehow need some nested routing, some child routes kind of, to have a router inside a router. Let's have a closer look at this in the next lectures.

**#### 139. SETTING UP CHILD NESTED ROUTES**

We improved our app quite a bit but the issue we have is that if we click on a server or a user, we load a brand new page. It would be nicer if we would load it next to this menu, so we need some nested routing. We can also see that in the app module, we kind of have some duplication here, all these routes here 

start with servers and all these routes with users, so it would be nice to nest them and to have some child routes which all start with servers then here in this case for example, let's add such child routes. To do this, I'll go to my servers component route, to the /servers route and I'll add another property, children.

{ path: 'servers', component: ServersComponent, children: },

Children takes another array of routes, so now I can simply take my two other servers routes here and add them inside of this array as children.

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

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

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

  ]

}

Now servers should now be removed because that is now always prepended at the beginning anyways,

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

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

so we have to make sure to get rid of it here and now we have just ID and id/edit. So this is now already better because now, we group that together, it gives us more than this visual grouping here though. You notice that on the servers here, we have the servers component as the component to be loaded still, the question now is where will the server component be loaded then because that is yet another child route of our servers component?

And you can see that this is an issue, if we reload our app here, everything's working, if I click on test server, now we get an error because now the error tells us that it cannot find an outlet to load our server component in and indeed it can't because the only router outlet, the only hook in our code where it should load components is in our app component here.

  <router-outlet></router-outlet>

Now that is reserved for all our routes on the top level, so slash nothing, /users, users ID name and servers but the child routes of servers need a separate outlet because they can't overwrite the servers component, instead they should be loaded nested into this servers component and that actually is the behavior we want.

So we can quickly get there by going to the servers.component.html file and here where we load the edit page or the app server page, well here I will simply comment out all that code and instead add a router outlet here.

    <button

*class*="btn btn-primary"

*(click)*="onReload()">Reload Page</button>

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

    <hr>

<router-outlet></router-outlet>

This now adds a new hook which will be used on all child routes of the route being loaded on the servers component which of course is our /servers route here, so all these child routes will be loaded in this router at the place now.

So if we save this and now we let this reload, you don't see anything here but if I click a server, it's now loaded next to the menu because this is where we added our second router outlet and this is how you can easily add child routes, let's do the same for the user routes now.

In the app module, I'll add children here, so this property which takes an array of routes and I only have one nested route here but that's fine, so I'll add it here. Get rid of the users at the beginning, just have slash, well then the ID and name,

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

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

the two dynamic parameters and load the user component and now in my users.component.html file, I'll replace my app user here

<app-user></app-user>

with another router outlet

<router-outlet></router-outlet>

where all the user related child routes or the one user related child route we have will be loaded.

So if we save this, you now see we load the user next to it and it is why it is super important to dynamically update the ID and so on because the component wasn't exchanged, we were able to switch the loaded user while this old component, the user component, the single user component was already loaded. And this is how you can implement child routing nested routes with these children property which then holds all the nested routes.

**#### 140. USING QUERY PARAMETER - PRACTICE**

In the last lecture, we added child routing, let's now improve this app a little bit more. In our servers component.html, we have our links loading to individual servers, so the single server component, this component here with this “**allowEdit**” query parameter. So now in the server.component.html, we should add some button where we say edit server ( **<button> Edit Server </button**> ) because right now, we get no way of reaching the edit-server.component besides manually typing this route here, this path here into our URL.

**$ {path: ‘:id/edit’, component: EditServerComponent }**

So let's add a button to the server.component.html , I'll add some CSS classes to make this look nice and of course, I'll add a click listener where I say “onEdit” ( (**click)=”onEdit( )”** ). Now again, this is on a single server.component.ts. So here I'll add the onEdit method ( **onEdit( )** )and in here, I now want to navigate to the edit-server.component. We learned how to do this, we need to get access to the router so that we are able to call the navigate method, so let's inject it ( **private router: Router** ), make sure to also add the import from **@angular/router** and with that injected in onEdit, we can navigate. You learned how to do this, so maybe a time to pause the video and try it on your own. This is how I would do it, I would call my router, then of course we have the navigate method and here we need to pass an array and we simply want to append edit to the end of the currently loaded route.

So you could of course also use /servers, the relative path, then use the ID of the currently loaded server again, server ID and then append edit

**this**.router.navigate(['/servers', **this**.server.id,'edit'])

but since we already are on this path, why not just use a relative route and just use edit?

**this**.router.navigate(['edit'])

Now you learned if you want to use a relative path here, when using the navigate method, you need to set up the relativeTo property here on the second argument, on this object you can pass to navigate and reference the currently loaded route, so that the @angular/router knows to which route you want to navigate relatively.

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

So with this, we should load the /edit route, we should append it to the end of our currently loaded

route, let's see if this works. If we go to servers, click on a server and I click on “**Edit Server**” button, we load the edit server page and now here, we want to control on where the user is allowed to update, to edit the server or not. The issue is, you see we theoretically have our query params here ( [**http://localhost:4500/servers/1?allowEdit=1#loading**](http://localhost:4500/servers/1?allowEdit=1#loading) **),** allowEdit always set to **1** in this case but we lose this information as soon as we click edit server, so that is something we need to fix before we can use the information there.

We will fix that in the next lecture, before doing so let's implement the rest of the logic though. Let's

say here on the server.component.html where we set the allowEdit query parameter, we don't always want to set this to 1, to true therefore, we want to decide on whether we want to allow the editing depending on the server ID let's say. So only if the server ID equals three let's say, only in this case we want to allow edit, so we want to set it to 1, otherwise we will set it to zero.

*[queryParams]*="{allowEdit: server.id === 3 ? '1' : '0' }"

So that's just a ternary expression, checking the server ID and if it's not equal to 3, we will set it to zero,

so we will set the allowEdit parameter to zero. That's the first step. Now in edit.server.component.ts of course, we want to be able to retrieve our query params.

So here, we already get the subscribe method on the query params, now here I want to determine on whether we are allowed to edit or not. So I will add a new allowEdit property here

export class EditServerComponent implements OnInit {

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

  serverName = '';

  serverStatus = '';

  allowEdit = false;

and set it to false initially but whenever this changes here in the query parameters, I want to get my query params which are also of type params to be imported from @angular/router and then here in the method body, I will set this allowEdit equal to my query params here equal to the allowEdit ID or property we set there and now this either is one, string of one, of course you could also pass true as a string here, in which case I want to set allowEdit to true. If it is not one, I will set it to false though

**this**.route.queryParams.subscribe(

      (*queryParams*: Params ) =>{

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

      }

    );

and in the template of this component.html, we could simply add a heading let's say where we say “you're not allowed to edit” if and we can use ngIf for this, if allowEdit is false, so if not allowEdit

<h4 *\*ngIf*="!allowEdit">You're Not Allowed To Edit </h4>

and we wrap all the rest in a new div, so all this part here only if it is true.

**$ <div \*ngIf=”allowEdit”></div>**

And now all of a sudden, we would control this but again the issue is if you click on edit server, our query params disappear, so it is always in the you're not allowed to edit mode. Let's fix it in the next lecture, let's see how we can actually preserve our query params once we navigate again.

**#### 141. CONFIGURING THE HANDLING OF QUERY PARAMETERS**

So in the last lecture, we try to improve our application a bit more but the issue is our query parameters are gone whenever we navigate away from our single server component, to the edit server component for example.

So we want to preserve them because we already set this information at the point of time we visit a single server, now we want to preserve this information once we navigate one step further to the edit server component. To preserve that information, we get a simple way of doing so. In the server component.ts where we navigate, we can pass another property or add another property to this Javascript object we use to configure our navigation and here, we get the query “**params handling property**”.

Now query params handling takes a string as a value and this could be merge, to merge our old query params with any new we might add here. Now we don't add any new ones, so we can simply choose preserve instead and preserve which will overwrite the default behavior which is to simply drop them and make sure that the old ones are kept.

**this**.router.navigate(['/servers', **this**.server.id, 'edit'], {relativeTo: **this**.route, queryParamsHandling: 'preserve'})

Now if we were to add new ones here, the old ones would actually overwrite the new ones, so we should use merge in this case but since we don't add new ones, preserve should make sure that we don't lose them. So now if we click on this again and click edit server, now you'll see allowEdit equal to zero is still set and therefore on the dev server where it is set to 1 if we click edit server, we do see the edit menu again.

[**http://localhost:4500/servers/3?allowEdit=1#loading**](http://localhost:4500/servers/3?allowEdit=1#loading)

So query params handling here, super important, super useful to know to make sure you don't lose the

information you had before.

**#### 142. REDIRECTING AND WILD CARD ROUTES**

You already learned a lot about routing. We're nearing the end but did you ever wonder what happens if we enter something here which we don't cover, like let's say just slash something. ( localhost/something ) We get an error because clearly, we don't have a route with slash something here. So that is something we probably want to improve, we want to do some 404 error handling, we maybe want to redirect the user to a specific page whenever he tries to visit a page we don't have.

These are the topics I'll focus on next, setting up a router which handles all routes we don't have set up and redirecting requests, not necessarily to topics you use together but also not that uncommon. So let's start with redirecting. Let's say we add a new component to our application and I will use the CLI for this with the gc for generate component command and I'll name it ***page-not-found***, the page not found component which was added here in our app folder, in this new page not found folder. I'll get rid of the spec file and here, I will simply say ***in the template***, “***this page was not found, something like this***”.

<p>This page was not found </p>

Now I want to render this page in let's say the case that we visit a certain route, I'll add it here **(app.module.ts)** at the bottom where the path is for now something and in this something case here, I want to load my page not found component. Of course, make sure that you added it to declarations and so on because we're still using it in our app of course.

{ path: 'something', component: PageNotFoundComponent}

So with this in place, if we now save this, go back to our application and visit slash something, we of course see this page was not found. ( localhost:4200/something )

Now let's change something to not found and let's make sure that once we add another new route which

then is something let's say, we don't also have to add component here but we redirect to another route, to the not found route maybe. For this, we get another property we can add to any route config here, so of course to any of the other routes. If you don't want to specify a component to load, you can say redirectTo.

Now this is an alternative to component, component loads the specific component, redirectTo redirects to another path. So here, you can specify any path and the path maybe now is /not-found.

{ path: 'something', redirectTo: '/not-found'}

So now if you do this, you'll see that it automatically redirected, so let me show you this again by typing slash something, watch the URL, it redirects to ***localhost:4200/not-found*** and all the other routes of course still work.

So this is how we can use redirection. Now of course, it's not very convenient to define slash something here because you probably can't define all possible wrong URLs the user can enter. So a more convenient way to catch all routes which are not covered by your app is to use the ***double asterisk route here***, this is the wildcard route which means catch all paths you don't know and the order is super important here. Make sure that this very generic route is the last one in your array of routes because your routes get parsed from top to bottom here, so if this was actually at the beginning, you would always get redirected

as I can quickly show you by saving this like this, now no matter where you click, you always are redirected

to not found.

So make sure this is the last route here in your set up and with this, now all the other routes work but if you enter anything invalid here, you always get redirected to not found because now you're using this wildcard route,

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

catching all routes which are unknown to Angular at this point of time once it reaches this line here and which then redirects to the not found route which displays the page not found page or component. This is how you can redirect and make sure you catch routes you're not covering right now.

**#### 143. 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 the just “/”

$ { path : ‘ ‘ , redirectTo: ‘/something-else’ }

Actually, Angular will give you an error here, because that’s a common gotcha: this route will now “ALWAYS” redict 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 parth 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 )

**#### 144. OUTSOURCING THE ROUTE CONFIGURATION**

So a lot was covered, let's dive even deeper. If we have a look at our application, our app module, you already see that the routing takes up some significant space in our app module. So typically, if you have more than let's say two or three routes, you don't add it directly in the app module, instead you **add a new file** which is for the application as a whole, typically is called **AppRoutingModule**.

So the apps-routing.module.ts file now will hold a second module and I will have a whole module course section about Angular modules later in the course. There, we will dive much deeper and learn how you can define and use multiple modules, use them together, this is a super easy use case here so let's add it now. Here I will give this a name of **AppRoutingModule** because that is what it is and it receives this @NgModule decorator.

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

@NgModule

export class AppRoutingModule{

}

So this is configured with a Javascript object being passed to it and this will be a super simple module.

Now as mentioned, I will dive deeper into this later, you can basically build your Angular app from multiple modules, this simple module here will handle all our routing related tasks. So I will cut all my routes, this appRoutes constant and add it in there, of course now I need to make sure that all these imports here are added.

const appRoute : Routes=[

  { path: '', component: HomeComponent },

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

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

  ] },

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

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

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

  ] },

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

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

]

So make sure to add all the imports to all these components and also of course, here to routes from the @angular/router package and you don't need to add declarations here because these components already are declared in your app module and we will soon add the AppRoutingModule to this root module.

So no need to redeclare them, this would actually even give you an error.

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

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

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

import { ServersComponent } from "./servers/servers.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";

We also have to remove the router module from here, from our root module,

*RouterModule.forRoot(appRoute)*

therefore we can remove the imports from the router package, make sure to import router module in your AppRoutingModule now, so to add this import here and now we need to configure something in @NgModule. We basically want to add imports again, we had this in the app module too and in imports, we want to use this router module, use forRoot as before and pass our routes array to it, so just like we did before but that alone is not enough now because as I mentioned, we simply want to use this app routing module to outsource our routes.

Therefore, we need to add our AppRoutingModule back to our main module and for this, we need to add the exports array here and I just can mention it again, we will dive deeper into modules later in the course. Exports simply tells Angular, hey from this module, if I were to add this module to the imports of another module, what should be accessible to this module which imports this module?

@NgModule({

    imports: [

        RouterModule.forRoot(appRoute)

    ]

})

And the one thing we want to make accessible is our ***RouterModule***. Now here, you don't call forRoot because you did this here, so in the first step you configure this router module, added your own routes to it, now we simply export this configured router module and therefore in app.module.ts, we can now import our own AppRoutingModule. So here on the imports array, we can simply add AppRoutingModule,

  imports: [

    BrowserModule,

    FormsModule,

    AppRoutingModule

make sure to add to import path here and with that, we have the same setup as before but with a bit of a leaner app module and our routing functionality outsourced in this separate module which is kind of a good practice to do to keep your files organized and easy to read.

So same behavior as before, if we save this with ng serve still running, we should therefore see that everything still works and this looks good to me but again important to understand, now with a different set up, a bit more readable and a good practice to implement if your application grows more complex.

**#### 145. AN INTRODUCTION TO GUARDS**

Throughout this section, you learned a lot about routing. You know how to set up routes, to load them, both through routerLink or programmatically, how to pass params, query params, how to use that data, how to load different components and much more. One important feature is left in this module which I definitely want to cover, route guards.

So basically functionality, logic, code which is executed before a route is loaded or once you want to

leave a route. Let's start with a simple use case, let's say you only want to give access to your server component, your single server component or the edit server component if a user is logged in. Now we're not going to add a full login functionality, for this we will later have the authentication section in this course but we will fake it for now.

Still, we want to check this before any of our subroutes here, so servers/3, servers/1 and then also the added routes are accessed. Now manually checking this in the onInit function of our server and edit server component would be very cumbersome because you would have to add it in both components and if you add more components belonging to the servers feature, you would have to add this over and over again.

So therefore, we want to use a feature built into the Angular router, running some code before the component is loaded, we will use the CanActivateGuard and we will do so in the next lecture.

**#### 146. PROTECTING ROUTES WITH CANACTIVATE**

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. I will add a new file in our root folder and I will name it ***auth-guard.service.ts*** because the code we run there is stored in such a service. I'll name it auth guard though because the feature of the @angular/router is called guards, it guards certain actions like navigating to, around or away from it.

But in the end, it is a normal service and therefore as such, I will export a class which I'll name AuthGuard, now you could add service here to be fully in line with all the other services but since we will only use this as a guard, I will leave it at the auth guard name and now here, I will implement the canActivate interface. This interface is provided by the @angular/router package, so you need to import it and it forces you to have a ***canActivate( )*** method in this class. The canActivate method now will receive two arguments, the ***ActivatedRouteSnapshot*** and the state of the router, so the ***RouterStateSnapshot.***

So make sure to also add these imports at the top here from an @angular/router and to add them here

as types for the two parameters we get in canActivate.

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";

export class AuthGuard implements CanActivate{

    canActivate(*route*: ActivatedRouteSnapshot,

*state*: RouterStateSnapshot )

}

Now you might ask, where are we getting these arguments from? Keep in mind, we'll soon define that Angular should execute this code before a route is loaded, so it will give us this data and we simply need to be able to handle the data. So that is canActivate with the arguments it requires, canActivate also returns something, it either returns an ***observable***, so make sure to import observable, from rxjs/Observable is the correct path in this case. This observable then will wrap a boolean, so it will in the end resolve to a true or false value.

Alternatively this route returns a promise, also returning the boolean in the end or it returns just

a boolean.

export class AuthGuard implements CanActivate{

    canActivate(*route*: ActivatedRouteSnapshot,

*state*: RouterStateSnapshot )

                : Observable<boolean> | Promise<boolean> | boolean{

                }

}

So canActivate can run both asynchronously, returning an observable or a promise or synchronously 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 a 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, you'll see an example in a second.

So here, we want to be able to login or out. For this, let's say we have another service, the ***auth.service.ts*** and this is just a fake service here. So I will name it ***AuthService*** and in a real application, this might reach out to a server and allow us to login or logout and check our current authentication state. I will track the 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*** just like this and a ***logout metho***d which was set ***loggedIn to false***, to again fake this behavior. And I will have a method which allows us to check the state, authenticated and here I want to simulate that this takes some time to finish because maybe we reach out to a server.

export class AuthService{

    loggedIn = false;

    login(){

**this**.loggedIn= true;}

    logout(){

**this**.loggedIn = false;

    }

}

Therefore here, I will return a promise, a new promise therefore, this promise will as always take a

method, a function as an argument with the resolve and reject methods we can execute and in this promise, here I will execute setTimeout to wait let's say 800 milliseconds and then execute another method in which I will resolve the promise and return this loggedIn, again just to fake that this takes a couple of seconds to finish or some time to finish.

   isAuthenticated(){

        const promise = new Promise (

            (*resolve*, *reject*)=>{

                setTimeout( ()=>{

                    resolve(**this**.loggedIn)

                }, 800);

            }

        )

    }

So with this auth service added, I now want to use it in my auth guard and as this is also just a service,

we need to add @injectable to be able to reach out to the other service, to inject that service into this guard. So make sure to import @injectable from @angular/core and then I will add a constructor to my auth guard where I reach out to my fake auth service here, of type auth service, therefore make sure to add this import too and now in the canActivate guard, I simply want to check whether the user is logged in or not.

So here, I can reach out to my auth service, to the isAuthenticates **this**.authService.isAuthenticated;

method which again keep in mind returns a promise, well therefore we should return this promise here I guess.

                 resolve(**this**.loggedIn)

                }, 800);

            }

        )

        return promise;

    }

So it returns us this promise and here, I then want to be able to handle that whenever this promise in the auth service resolves, I know that there, I will get back a boolean, so this authentication state, this loggedIn boolean is what we resolve here in the end and this could be false or true.

So I know that I will get back a boolean and here, I then want to check if this is true, in which case I want to return true and otherwise, I want to navigate away because I don't want to allow the user access to the route you wanted to go to originally, I will navigate away to force the user to go somewhere else.

So I will inject the Angular router for this because this is how we can navigate with the navigate method,

make sure to add the import from @angular/router and then here, we can navigate with the navigate

method and let's say we just want to go back to our root page with just slash.

So in this case, our canActivate method will navigate away and I will now simply return this promise it

still is in the end because if we return something inside of the promise, it will give us back another

promise. So now we're returning a promise here which gives us back true or simply navigates us away, canceling the old navigation anyways therefore.

So this now allows us to control access to whatever is controlled by this canActivate guard here. Now we're still not using this guard of course. So to use it, I'll go to my app-routing.module and now we want to define which route or routes should be protected by this guard and we do so by going to that route,

it's the servers route and all its child routes and adding canActivate, this property to it. Now canActivate takes an array of all the code basically, all the guards you want to apply to this route and it will automatically get applied to all the child routes. So here, canActivate will use my auth guard, make sure to add the import here pointing to your file and this will make sure that servers is now only accessible and all the child routes, only accessible if the auth guard canActivate method returns true in the end which will only happen if in the auth service, loggedIn is set to true.

Since this is set to false and right now we never call login, this should always deny us access. Now before trying this, we need to go to our app module and add to the two new services we added. So add the auth service and the auth guard which also is a service, otherwise Angular won't be able to inject them for us. So with that set up, let's go to our application, click on servers and I can click there as often as I

want, I can't reach it.

Users works but servers always navigates us back to home after this 800 milliseconds because that is

how long it takes to resolve the information whether we are authenticated or not, we defined these 800 seconds here. So our guard is working, however on our whole servers tab. Now I want to be able to see the list of servers and only protected child routes, how can we do this?

then