**ANGULAR 8**

**#SECTION I : GETTING STARTED**

**#### 1. COURSE INTRODUCTION**

Welcome to my complete guide on Angular it's so great to have you on board in this course. You're going to learn what angler is and how you can use it to build awesome front end web applications with it. Very reactive web applications with great user experiences for that throughout this course. You're going to learn all the individual building blocks which make up angular before we then also dive into these more advanced features. And we're going to build a project throughout the course putting all these things you'll learn in theory kind of in which you practice throughout the course into the perspective of a real project so that you can see how you would apply them there. And that as you can see how you build up a real web applications with the things you learn here. So with that I'd say let's dive into it and let's dive into the most important question first.

What is angular ?

**#### 2. WHAT IS ANGULAR?**

Angular is a framework which allows you to create reactive, single page applications. This sounds great, but what exactly does it mean?

A Single Page Application (SPA) is an application like the one shown here( the project ) ; the project we're going to build throughout the course. You can navigate around and in the URL, you can see that we seem to visit different pages, but in the end, our page never changes. It's only one HTML file and a bunch of JavaScript code we got from the server and everything which you see here, every change, is rendered in the browser.

Now, why is that awesome?

It gives the user a very reactive user experience (UX). JavaScript is much faster than having to reach out to a server for every page change and for every new piece of data you want to display. Therefore, this approach allows you to create web applications which look and feel almost like mobile applications; very fast! Everything happens instantly. If you do need some data from a server, you simply load it in the background so that the user never leaves this experience of having a reactive web application to use.

So every click I do here simply changes this one single page we're using; this one HTML page.

So, how is this done? Well, JavaScript changes the DOM, changes whatever is displayed here (in the browser), by changing the HTML code during runtime (so to say). That is why you never see the refresh icon on the top-left spin; because we're only changing the currently-loaded page. You can even see that if you inspect the source code of a page like this.

That is the HTML file and as you can see, it doesn't seem to contain the content you are seeing on this page. We only have one single HTML element which doesn't seem to be a built-in one (a native one), but that's Angular doing its job. Throughout the course, you're going to learn exactly what happens here and how Angular does this job.

**\**

**#### 3. JOIN OUR ONLINE LEARNING COMMUNITY**

As a student of this course you also get free access to our “Academind Community” on Discord”

https://discord.gg/gxvEWGU

there, you can find like minded people discuss issues help each other share progress successes and ideas and simply have a good time.

I believe that you learn the most if you don’t learn alone but find learning partner and other people with similar interests. Our community is a great place for this – it’s the perfect complimentary resource for this course.

Joining it is of course free and 100% optional.

**#### 4. ANGULAR VS ANGULAR 2 VS ANGULAR 8**

Now one thing that can be confusing when you're getting started with angular is the amount of angular

worshippers you find out there. Now in this lecture we'll have a look at the different versions how they are connected. If that means that angular changes all the time.

Spoiler It does not.

So we had angular one. It was released a couple of years ago and angular one was a big new thing back then. Now this course is not about angular one though because while said was the big new thing it also had some issues that could lead to worse performance and bigger applications ends on this question of angular angular one is nowadays referred to as angular J.S. written like this.

So if you see people right angle or J J.S. in some Internet forum on Reddit or anything like that they

are most likely referring to angular one which is totally different to the Anglo world and you're learning

about than discourse because there was a complete rewrite of the angular framework between angular one and angular two angular Q was released in 2016 and it was rewritten from the ground up.

It works totally different than angular one to fix all the issues angular one had now since then since

that initial resolve angular two we had a couple of other versions of angular with angle of four angular

free was skipped for internal reasons so that whereas a number was skipped we had angular five six seven

and now we have angular nine. Why do we have all these angular weird numbers does it mean that angular was reinvented eight times since angular one No this is not the case instead since the release of angular to the angle or a team simply adheres to versioning scheme where a new word of angular is released every six months now that new word however is not a complete rewrite it does not change everything.

Indeed most updates change almost nothing only some behind the scenes stuff or add some new features

without breaking existing features. Indeed angular 9 is pretty much the same as angular 2. I initially created this course for angular 2 and I updated it multiple times and it is totally up to date day 4 but when we have a look at the core syntax at the philosophy ends on nothing changed since angular 2.

Indeed if you learned angular 2 in 2016 and you then slept for 3 years you can still use the exact same

syntax. Some minor things changed and I updated the cores over to time to reflect that but overall it's the

same framework. We have small incremental backwards compatible changes between these versions.

It's simply just a commitment which the angular team made to release such a major new version every

six months. It does not break or change everything every time it's released and therefore we have angular just is worse and one thing which is totally different. And then what we just refer to as just angular angular 2 4 5 6 and so on.

This is all just angular.

You shouldn't really use the version because in 6 months it will be outdated anyways.

Instead it's just angular and this course is just about angular because as I said angular 9 is basically

the same as angular to an angular 10 will be the same and as well we just will see small improvements

over time. That's the idea behind these versions.

**#### 5. CLI DEEP DIVE AND TROUBLESHOOTING**

In the next lecture, we’re going to build our first little app!

If the CLI prompts you to “answer some questions” ( some version do that ) you can simple hit enter for all questions this will accept the default settings which are fine for this course.

The CLI generates a different welcome screens than you’re going to see in my video though. No worries you’ll still be able to follow along without issues! Just make sure to code along so that your code equals mine Angular itself didn’t change a bit

If you want a dive deeper into the CLI and learn more about its usage have a look at this: <https://github.com/angular/angular-cli/wiki>

You encountered issues during the installation of the CLI or set up of a new project?

A lot of problems are solved by making sure you’re using the lastest version of Node js and npm and the CLI itself

**Updating NodeJS:**

Go to nodejs.or and download the latest version- uninstall ( all ) installed version on your machine first

Updating npm:

**$ Run npm install –g npm**

**$ Updating the CLI**

**$ Npm uninstall –g angular-cli@angular/cli**

**$ Npm cache clean**

**$ Npm install –g@angular/cli**

Here are some common issues and solutions:

-Creation of new project takes forever ( longer than 3 minutes )

- that happens on window from time to time > Try running the command line as administrator

-You get an EADDR error ( Address already in use )

-You might already have another ng serve process running make sure to quit that or use “ng serve - - port 4200 or 4201 or 4202, depends which ones you haven’t used. ANOTHERPORT to serve your project on a new port

-My changes are not reflected in the browser ( app is not compiling )

Check if the window running “ng serve” displays an error. If that’s not the case make sure you’re using the latest CLI version and try restarting your cli

**#### 6. PROJECT SETUP AND FIRST APP:**

So now with the definition of angular and the different worship is out of the way. Let us simply dive into it and let's build our first angular app for that. We'll use the official angular command line interface CLI and that is the recommended and best way of creating angular projects because angular projects are actually a bit more elaborate regarding their build workflow.

There are a couple of files that need to be converted before they can run in the browser and to see

CLI does all of that and it also heavily optimizes our code so that we ship highly optimized code version in to the browser. Once we finally deploy our app which will all the data in this course in the end of course.

So the Angular CLI and you can see all the steps you need to run on the left here and these steps

are the same for Windows and Mac.

**$ -npm install –g@angular/cli**

**$ -ng new my-dream-app**

**$ -cd my-dream-app**

**$ -ng serve**

Now for them to work you need one additional tool and that's node J.S. Node JS is a service side language

and you might be wondering what are we doing with it. In this course it's the angular course not a node course and you'd be right. We won't write any node chess code here but note J S will be used behind the scenes by the CLI to bundle and optimize our project and will use NPM the node package manager to manage two different dependencies and angular project test dependencies are things like the angular framework itself but also some other libraries.

That framework uses so you can simply download the latest version here by clicking on that button. In my case it's 10. 0 but that might differ for you and you will get an installer for which you can walk which will conveniently install node.Js and this NPM tool on your machine. Once you've finished installation we can run these commands and we run them in our terminal or command prompt.

Now here I'm on a Mac. Therefore I used to normal terminal. If you were on windows you would open the normal command prompt possibly open that as an administrator by right clicking on the executable and clicking Run as Administrator to rule out some potential issues. But in the end you should be able to simply run NPM using that node package venture “**install dash g**” to install it globally on your machine, **“npm install –g @angular/cli@latest”**

Is not so you absolutely need to run this now on Windows. This should just work on mac or linux you probably need to add a pseudo in front of this to give yourself the right permissions hit enter and now I'm prompt to enter my password. You may or may not be and once you confirm everything.

This should now download the angular CLI from this node package manager repository and install it

on your machine. Now as you can see you might get some errors in between like I'm getting here.

You may ignore them as long as it's successfully installed CLI as you can tell by the output here

at the bottom so it added to it updated to packages or added them if you installed the first time and

you see CLI and the version installed now the wording can differ depending on the time when you install

this. By the way the general functionality will not differ.

So this did work.

Note that we can create our first project and we do this with this **“ng”** you command here which is

now available since we installed CLI For that navigate into a folder where you want to create that project and once you navigate it there with the D command you can simply run **“ng new – my-first-app”**.

This name is totally up to you. So “**ng new**” is a set command to create a new project. It must not be test or something like that though because that actually is a reserved word which will not work.

Now depending on the wording you're using you might be asked two questions here a couple of questions.

The first one is would you like to add **“angular routing”.** Now you can just type in here or hit enter to use the default which is also no. And then here which styles you'd form and you want to use and there you also want to go with the default for discourse which is already selected so you can hit Enter here too.

So essentially the result is hit Enter two times and now it will create a new project. And now this will create a new folder with a couple of files and dependencies on the entire build workflows setup in there. And as I said we need this more complex setup because for example angular uses typescript superset of javascript.

So basically a language that looks a bit like Javascript and is compiled down to JavaScript in the end

by that workflow but which also offer some extra features and to do that computation as well as a couple

of other optimization steps. We need this more complex setup with all these dependencies. Now once this finished successfully and if it didn't check out the lecture prior to this video here I do share some common errors and fixes there.

So once this finished successfully we can navigate into that with the cd command and then the name

of the project in my case it's my-first-dash-app And once you're in there you can run **“ng serve”** to basically bring up a development server that will run your built and for development optimized app so that you can see it in the browser the server runs on locally hosted 4200, localhost:4200 by default you see the output here.

So you can simply go to the browser enter localhost:4200 and you should see something like this.( welcome app with angular logo ) This is just a simple starting screen provided by the CLI project.

Welcome to app. And then some useful links to the official docs and so on. Awesome.

Now on the screen you're seeing actually all might look something like this.The starting screen simply was updated with newer versions of the CLI . This does not impact what you learn or the version of angular you're using everything you'll learn in this course is fully up to date with the latest version of angular.

I kept this course up to date in the past. I will do so in the future. It's only that starting screen that changed.

Now in order to follow along smoothly attached you actually find the app component.html file and

you can simply go into your angular project that was created on your hard drive with the engine you

command and there into the source folder src see folder into the app folder and then simply drag the

attached app component H2 out file in there and replace the existing app component HDD him and follow

you find in there with it and with that if you reload local host:4200 you don't even need to restart the server. You'll find the same looking starting scream I have here again.

That's just for you to follow along smoothly will work on these files in the next lecture so we'll dive

into what you've got there in this folder and what this app component.html file actually does. This is just so that you can follow along without problems. Angular itself has not changed its justice starting scream.

Now this is our first application.

Now we set this up.

Of course not too much is happening here so let's edit this first step and do something more fancy with

it and also understand what exactly the cli like created in the next lecture.



**#### ANGULAR CLI**

Run these on the command line

**$ -npm install –g@angular/cli**

**$ -ng new my-dream-app**

**$ -cd my-dream-app**

**$ -ng serve**

**#### 7. EDITING THE FIRST APP**

In the last lecture, we created our first Angular app but we haven't changed anything there. So it's time to do that. For that you need an IDE or editor where you can write and edit your code. Now I'm using WebStorm here. Webstorm is a great IDE, great for Angular development; but it's not free. A great free alternative would be Visual Studio Code which you can find on <https://code.visualstudio.com/> This is a great IDE and it is free.

You can simply download and install it both for Mac and Windows and then you can start developing Angular with that. Whichever IDE or editor you install, you'll always have to open a new folder (in WebStorm's case it's just called Open) and then simply navigate to the folder you created.

So in my case here it's my-first-app. Select that folder and open it and now it will load that folder and all the files into the IDE on your system. So in my case, it now looks like that. Now this is the project loaded into the IDE. It's still indexing it here but it'll be done in a second. These are all the folders and files The Angular CLI created for you. This is your entire Angular project. Now as I said, this might look intimidating because you've got so many files in there. Most of these files are just doing some configuration work and you don't really need to touch them. One interesting file is the package.json file.

Here you can see all the dependencies of your project like Angular 6 and these are third-party packages

your project needs to run correctly. All dev Dependencies are only required for development; for this build workflow I was talking about. But we're here to edit our code right? So let's jump into the src folder because that is where our code is. e2e is for end-to-end testing (we'll ignore this) and “node\_modules” is where all these dependencies you see in the package.json file actually were installed. But we're interested in the source code so let's go to src. Then we got a bunch of other configuration files and then here, we've got this app folder and in this app folder we see some other files. Of course I'll dive into what they all do in detail in this course.

For now let's open the “app.component.html” file. Here we indeed see something which looks deceivingly like what we saw in the browser. Do you remember? It's Welcome to app! and so on.

We see that here too. By the way, make sure you keep the localhost:4200 process running (the ng serve process you started here in the terminal). Make sure to keep that running, but if you're done with development for the day You can quit it with Ctrl-C and you'll get out of there, but as long as you are developing you should keep it running because it will automatically watch your files and rebuild your project whenever you change and save something.

Now since I quit it

I'll need to restart it and I will actually restart it here in my IDE. In here there is a built in terminal but it's the normal system terminal. So here I can also run ng serve to bring that back up. But back to the app.component.html file, we're outputting something here and if we change anything here, like,

'Hi, this' (instead of 'Welcome to') and we save it with ng serve running, then if we go back, you'll see it automatically updated; Hi, this app!

Now, one strange thing we see is this 'app' here, but we actually only see these curly braces and title in the app.component.html file and therefore, we can already see some of the work Angular does here.

Angular is, of course, not a tool to allow us to write static HTML files. We wouldn't need a framework for that. It allows us to mix static HTML code and dynamic things we want to output in that code and actually, what we have here is one of these components Angular works with; the app component. A component always has a template, the HTML code, possibly has some styling in the CSS file (though it's empty here as you can see) and importantly, it has a Typescript (.ts) file. If we enter this, this is Typescript and this is now the definition of the component. This is what will be converted to normal JavaScript by the build workflow. And in this file, we see a couple of interesting things like @Component. I'll come back to that and what in detail is happening here in the next module. We also see title = 'app'. Now, what's that?

If you go back to the app.component.html file, we also saw title here right; between the curly braces.

So an assumption would be that the title in the Typescript file is related to that. So let's maybe change this to 'my app' and now if we save this it recompiles and if we go back, we see 'Hi, this my app!'. Now this might not be perfect English, but we can definitely see that something changed. This is so-called data binding in action; a concept we'll also dive into deeply in this course. This is how we can output dynamic content.

This could of course be content that is calculated dynamically or retrieved from a server in our HTML code. Now interestingly, if we right-click on the loaded page and inspect the page source we don't actually see that code there. We just see a bunch of script imports at the bottom. That is our build code and the Angular framework code, the head tag here and then this strange app-root part. Now, app-root is also something we see in the app.component.ts file here in the selector.

So it looks like this is basically like our own HTML tag we're creating and that's indeed what's happening here. The page we're viewing here, to which this source belongs, actually is this index.html file in the src folder. Here we also see app-root. The script imports are missing because they are injected dynamically.

Now, what Angular does in the the end is always load this page. Then we have these dynamically injected script imports and these script imports will dynamically replace app-root with our own component. And of course we can have more than one component in Angular apps.

Again, these are all things we'll dive into in this course. Now that's nice, but let's do something more fancy. Let's change title dynamically. So back in the app component, I'll actually get rid of the entire code in here and I'll add an input instead; a normal HTML input (type="text"). Below that I'll add a paragraph where I want to output, let's say, a name. Now, name is not something we've defined yet, we'll do so in a second.

Let's go to app.component.ts and change title to name and I'll say Max here (because that is my name) and if I save that and go back, we see the input and we see the name. Now I want to be able to enter something in the input and automatically change the name. We can do this with a tool provided by Angular; a so-called directive which is called ngModel.

Now, you'll learn the exact syntax throughout the course. For now, let's simply add [(ngModel)],

written like that. Make sure to get the casing correct. There is no dash in between or anything like that.

So ngModel in square brackets and parentheses on the input and set it equal to name. So, to the same name you're outputting here. Now this is a so-called directive and what it does is it basically tells Angular to listen to anything you enter here and store it in this name property, in this name model, but also on the other hand, output the value of the name model in this input.

This is what we're doing here. Now, if we save this we don't see anything on the page and if we open the developer tools, which you can simply do from the menu or with the shortcut. You reach it under View -> Developer -> Developer Tools (in Chrome) and I strongly recommend using Chrome for this course since it has the best developer tools. So, then we see here in the console-part there is an error. Can't bind to 'ngModel' since it isn't a known property of 'input'. So somehow Angular doesn't understand ngModel.

Now that's strange, because as I said it's built-in right? Angular is actually split up into multiple modules;

sub-packages you could say. We need to add them if you want to use a certain feature from them. And in this course you will learn about the different packages and which features they contain. To add such a feature, we go to another file that we haven't had a look at yet; the app.module.ts file. This is basically where we tell Angular which pieces belong to our app and there we have to add something to imports to import another package from Angular. So we need to import it at the top of the file first because Typescript always needs to know where things are. So, import { FormsModule } from '@angular/forms';

This is one package from the Angular framework and from that package we can import the Forms Module; written like this: import { FormsModule } from '@angular/forms';

So make sure to add this import. This is not an Angular feature; it's a Typescript feature. As I said Typescript needs to know where things are. But now with that imported we can add it to this imports array down here (which is not related to the imports up there). This is simply a feature understood by Angular since it's part of an Angular module. And now it tells Angular that we want to import some form features. And this directive we're using (ngModel) is such a form feature.

So now if we save that, our app will rebuild and once it's done, we can go back and see our input

with Max inside of it. And if I start typing there, you'll see that the text below it automatically updates.

This is some magic done by ngModel. We'll dive much deeper into everything we learned here. But this is now our first app; with some changes.

\*\*\*\*\*\*\*\*\*\*\*\*\*

Download the latest version of Node js before you run the above command lines. It will create a folder with the app name and use default setting by hitting NO ( or N ) for angular routing and CSS just press ENTER Open the IDE and the folder will be “my-first-angular-app” and go to the src folder Angular is, of course not a tool to allow us to write static HTML files. We wouldn’t need a framework for that. It allows us to mix static HTML code and actually what we have here is one of these components Angular works with; the “app component” We also see title = 'app'. In “app.component.ts”

Now, what Angular does in the end is always load this page. Then we have these dynamically-injected script imports and these script imports will dynamically replace app-root with our own component. And of course we can have more than one component in Angular apps.

Make sure to get the casing correct. There is no dash in between or anything like that. So ngModel in square brackets and parentheses on the input and set it equal to name. So, to the same name you're outputting here. Now this is a so-called directive and what it does is it basically tells Angular to listen to anything you enter here and store it in this name property, in this name model, but also on the other hand, output the value of the name model in this input. This is what we're doing here. Now, if we save this we don't see anything on the page and if we open the developer tools, which you can simply do from the menu or with the shortcut.

You reach it under View -> Developer -> Developer Tools (in Chrome) and I strongly recommend using Chrome for this course since it has the best developer tools. So, then we see here in the console-part there is an error.

Can't bind to **'ngModel**' since it isn't a known property of 'input'. So somehow Angular doesn't understand ngModel. Now that's strange, because as I said it's built-in right? Angular is actually split up into multiple modules; sub-packages you could say. We need to add them if you want to use a certain feature from them. And in this course you will learn about the different packages and which features they contain. To add such a feature, we go to another file that we haven't had a look at yet; the app.module.ts file. This is basically where we tell Angular which pieces belong to our app and there we have to add something to imports to import another package from Angular.

So we need to import it at the top of the file first because Typescript always needs to know where things

are. So, **import { FormsModule } from '@angular/forms';** This is one package from the Angular framework and from that package we can import the Forms Module; written like this:

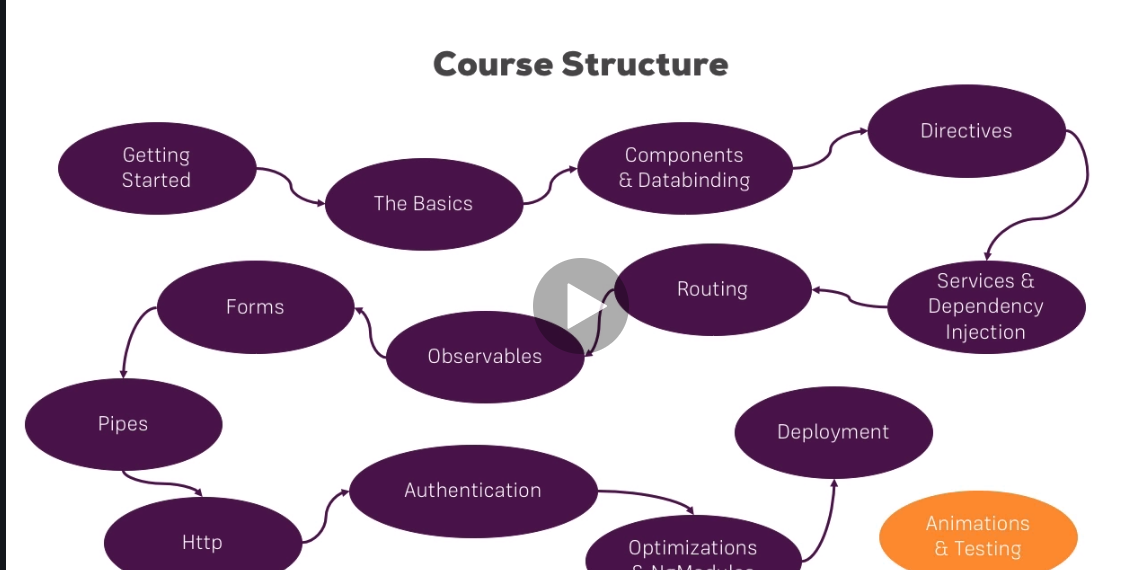
**$ import { FormsModule } from '@angular/forms';**

So make sure to add this import. This is not an Angular feature; it's a Typescript feature. As I said Typescript needs to know where things are. But now with that imported we can add it to this “**imports array”** down here (which is not related to the imports up there). This is simply a feature understood by Angular since it's part of an Angular module. And now it tells Angular that we want to import some form features. And this directive we're using (ngModel) is such a form feature. So now if we save that, our app will rebuild and once it's done, we can go back and see our input with Max inside of it. And if I start typing there, you'll see that the text below it automatically updates. This is some magic done by ngModel. We'll dive much deeper into everything we learned here. But this is now our first app; with some changes.

**#### 8. THE COURSE STRUCTURE**

What does this course offer you? We're just **getting started** and we just built and edited our first Angular application.

But of course we didn't fully understand what we did there. So that's why in the next lecture we're going to dive into ***the basics*** of Angular.



**#### 10. WHAT IS TYPESCRIPT**

We're about to get started soon, but I did mention that we're going to use Typescript in this course.

We already did use it when we changed the first app. What is Typescript? Typescript, really is just a superset of JavaScript.

It offers more features than vanilla JavaScript, like classes, interfaces and, very important, which gives it the name, types; strong typing. So, you define in Typescript if a certain variable is a number, a string or something else. You don't do this in vanilla JavaScript. There, you have dynamic typing. You can have a string variable and then you can assign a number and that's totally fine. That won't work in Typescript.

It will give you an error and therefore it allows you to write much more robust code which gets checked at the time you write it; and not just at the time you run it. This is a great enhancement.

However, Typescript doesn't run in the browser, so it is compiled to JavaScript in the end. This compilation is handled by the CLI; one of the reasons why we need the CLI, why we need a project management tool like the CLI. Now, this compilation is really fast and therefore, in the end, in the browser JavaScript is going to run. We're not writing the Angular app in JavaScript though because, whilst technically possible, that wouldn't be much fun. A lot of the features really only exist in Typescript and Angular is meant to be used together with Typescript. That being said, you don't have to learn Typescript for this course.

You should be able to pick it up along the way. I do have a whole section giving a brief introduction into Typescript at the end of this course, so feel free to jump ahead. Go through that section right now if you feel like you need it or, throughout the course if you feel like you need some deeper dive into some of the features you see there and you should be fine, you should be able to follow along just fine. Of course, if you feel like it, feel free to pick up another course or tutorial focused on Typescript so that you have something to look at if something is unclear in this course. Generally though, Typescript is an addition to JavaScript; not a complete replacement, so a lot of the commands will look really familiar.

**#### What are *components***?

What did we do there with this **“two-way data binding”** and how does that all work. This section will answer it and we'll take a very detailed look at it. We'll also explain how all these files you saw in your project are connected to each other. Then we're going to look at ***components*** and **data binding**; two important key features and we actually saw both already in our first application. Angular apps are built up from components and data binding is simply how you output data in your DOM in the end and, as you will also learn, react to user events. Angular has another key feature; ***directives***. ngModel, which we used with ***two-way data binding, actually is such a directive***. You will learn more about the built-in directives in this section and, also very important, build your own directives; nice little helpers (instructions) you can place in your templates, in your HTML code, which will then do something at runtime depending on the commands you wrote in there. Well, after having a look at this, we're going to learn more about ***services and dependency injection;*** a core feature of Angular which makes it really easy for you to have your different pieces in your app communicate with each other, to centralize code and to manage the state of your application. Once we're finished with that it's time to have a look at routing because, thus far, we will only have been on one page.

Well, we're always on one page since it's a single page application, but to the user it really looked like one page. With routing, we introduce the management of different URLs so that, to the user,

it looks like we're switching pages even though technically, we will still remain on that single page.

Sounds great! It is and routing shows you how it works. Then it's time to have a look at ***Observables***; something which will make more sense once you've been through the routing section. It is a concept allowing you to work with ***asynchronous code***.

Angular embraces it, it's really powerful and this section explains how it works. Then we'll have a look at ***forms***, because handling forms, handling user input is a key task of almost any application and this section takes a very close look at it. After we're done with forms we're going to have a look at ***pipes***; another nice feature which makes it easy for you to transform the output (what you display on the template at runtime). Well, we're nearing the end, but not before having a look at HTTP.

What if you need to reach out to a web server?

What if you need to store some data in a database?

Angular can't connect to a database directly, but it can connect to a server which is able to and the Http Section shows just that. Thereafter, we'll have a look at ***authentication.***

**What does authentication mean in an Angular application?**

How does it work? While we're at it, we'll implement it in an application. Then we're going to have a look at some optimizations we can put into place and how we can manage different modules in our application; something you will really understand once we are at this point. Then, we're going to deploy an application, learn how we can get our Angular application from our local machine to a place in the Internet where we can view it. And if that's not enough for you, we're also going to have a look at animations and testing Angular applications.

Besides that, throughout the whole course, we're going to build a real project putting all these things you'll learn in the individual sections into practice and, therefore, see how they are used in a real project. So with that, I feel very confident you're going to be an Angular Master after finishing this course and I can't wait to start this journey together with you.

**#### HOW TO GET MOST OUT OF IT**

-watch the videos and read the transcript

-do the assignments

-do the course project

-use the Q&A section

***What is TypeScript ?***

It offers more features than vanilla JS like classes interfaces and very important which gives it the name types: strong typing

So you define in Typescript if a certain variable is a number a string or something else. You don’t do this in vanilla js. There you have a dynamic typing You can have a string variable and then you can assign a number that’s totally fine. That wont work in Typescript. It will give you an error and therefore it allows you to write much more robust code which gets check at the time you write it and not just at the time you run it. Typescript doesn’t run in the browser so it is compiled to JS. At the end of the course you can go through the Typescript.

**#### SECTION 2: BASIC PROJECT SETUP**

So this is the page as we see it right now in the project we created at the end of the last section,

an empty page here, Angular, The Complete Guide, we reach it at localhost:4200(newer versions have different initial screen, we have used app.component.html which is downloaded from the course. Use this template and you will see the Angular logo), this is where our development server hosted by the CLI or spun up by the CLI will host our Angular application.

Now the content we see here can be changed by going to the ***app.component.html*** of file in our app folder

and in here, we can output some text let's say I'm the app component here in a **“h3 tag”**, a normal

HTML tag therefore. If we save this, we will see it here and it automatically has the bootstrap text style and so on because we added bootstrap at the end of the last section too.

But whilst it might seem kind of obvious that we see it here, it is also kind of strange, how does our browser or how does the server hosting our app know that it should render the content of “app.component.html “here? You could argue it's the only component we have right now and we will take a closer look at component. Soon but that is not the reason and actually, this is not the file served by the server, instead the index.html file here is served by the server and remember that I told you that Angular is a framework which allows you to create single page application, this is the single page which is served, the index.html file.

Now if you have a look at this file index.html, we see this is a normal HTML file, we define the title we could see on the tab in the browser here and the body of this file is very interesting. Here we get this app-root (index.html file) thing with loading.... in between. Now clearly we don't see loading... here( on the local hos), so somehow this index.html file seems to have changed and it did, app-root here here of course is not a default HTML element, instead this is one of our own components.

We will soon dive into how we create our own components but the CLI created one for us, the root component of our application, the component which will tie together our whole application in the end and all the files in the app folder here which have component in their name, so these files are related to this component.

I will go into more detail about these files in one of the next lectures, for now let's take a closer

look at the app.component.ts file, the TypeScript file here. Here you can see that we have this ***@component*** decorator, this seems to be important but more importantly right now, you'll see that there, we have this **“selector property which assigns a string as a value”** and this string holds *app-root*. Now this clearly is the same text as in our index.html file and this actually is the information Angular needed to be able to replace this part here in this index.html file with the template of this component, the component having this selector and the template on this component simply is the content here in this app.component.html file.

Now whilst I will dive deeper into how to create components and how to configure them, this is what basically happens at the startup, though the missing information is how is Angular triggered, how is it kicked off to actually run over our body here of this index.html file? And the answer is in the final index.html file, getting served in the browser and we can verify this by inspecting the source code here, we got a couple of script imports at the end. These are injected by the CLI automatically, so that is why we don't see it here in the raw index.html file, here we don't have any script imports but whenever this ng serve process rebuilds our project, it will create bundles, Javascript script bundles and automatically add the right imports in the index.html file, a little convenience functionality for us.

So in the final file, these script imports here are present and these script imports will contain our own code too. So these script files are therefore executed and they're actually the first code to be executed and that is just something you have to keep in mind, is the code we write in our main.ts file, that is why it's called main, this is the first code which gets executed. Let's have a closer look at it then. Here you see that we get a couple of imports, then we check if we are in production mode or not to basically turn off some warning messages I can tell you that but most importantly here, this line, this now bootstraps starts our Angular application by passing an app module to this method and app module refers to this file here. Now if we have a look at this file, here we actually see that somehow we kind of like with the component have this strange @ thing here, I will dive into this in a couple of lectures too but most importantly for now, here we get this bootstrap array which basically lists all the components which should be known to Angular at the point of time it analyzes our index.html file and here the circle closes because here, we reference our app component.

So Angular gets started, this main.ts file gets started, there we bootstrap an Angular application and we pass this module as an argument. In this module, we tell Angular hey there is this app component which you know when you try to start yourself and Angular now analyzes the app component, reads the set up we pass here and therefore knows this selector, app-root and now Angular is able to handle app-root in the index.html file and it knows all right this is the selector I know, you told me that I should know it because it was listed in this bootstrap array in the app module, this component.

So now I know that here I should insert the app component and the app component happens to have some HTML code, a template attached to it which is this h3 tag and this is how the Angular application starts here, this is why we see what we see. Now that was a load of information regarding the app startup but it's key to understand how this starts, with this information we can now proceed and dive deeper into how components actually work, what they are, how we can create our own ones and what this app module here does in detail.

So let's move on.

**#### 15. COMPONENTS ARE IMPORTANT**



In the last lecture, we had a close look at what happens at the point of time we visit localhost:4200

here in the browser. We understood that the index.html file is served, that it will contain a bunch of scripts here which get executed which then basically start the Angular app. The Angular app gets the important information, that it should know, the app component that it should analyze it with that information the Angular code is able to parse this here, this app-root component here understand it and insert our Angular application at this point and that is why we don't see loading... which would still be in the source code of this page as you can see but instead why we see this because Angular overwrites this at runtime because that is what it does, what its job is.

So now that we had a close look at this startup process, let's have a closer look at this component and

or @components in general. Components are a key feature in Angular, you build your whole application by composing it from a couple of components which you create on your own. Now we do start with this app component, the root component you should say which holds our entire application basically in the end. So this root component, this app component will be the component where we later nest or add our other components too, so to this template, this HTML file of the app component, this is where we will later add our other components. Typically, we might have a web page like this, with a header, with home and users, then a main area and a side bar maybe. Now if we have a look, this would be a fitting component, the header and maybe also the items on the header could be their own component, could be, you don't have to, you will learn throughout the course how I approach the splitting up of an application into separate components.

We also have of course our main area and the side bar here as components and that is the core idea behind Angular, build components, each component has its own template, its own HTML code, maybe its own styling and more importantly also, its own business logic and this is the great benefit. It allows you to split up your complex application, your complex webpage into reusable parts, you may use a component more than once and that allows you to easily replicate that business logic, replicate that styling or in general, make a finely controlled piece in your application without having to crunch everything into one single script file, one single HTML file, instead it's very easy to update, very easy to exchange and again re-usable. So let's have a closer look at how we actually create a component and what this @component thing does in the next lecture.

**#### 16. CREATING NEW COMPONENT**

Components are important right, that is why we start with one, with the app component. It on the one hand is a normal Angular component but on the other hand, it's also kind of special because it serves as our root component, it's in the end listed here in the app module in this bootstrap array which tells Angular, hey this is a special component, you should bootstrap the whole application with that component being the root component. So all other components we create will not be added to the index.html file, there selectors will not be added here, there selectors will be added to the app.component.html file because this is now the root component of our app where we add the other parts. Enough talking about adding components, let's add one.

Let's say we want to output some information about a server. We're building a back-end for our server management application and we want to output some server information. So I will store this in a new folder which is a subfolder of the app folder because generally in a Angular CLI project, all your app related content will go into this ***app folder***. I will name it **“server”** because it will hold my **“ServerComponent”** and this is a good practice, having the folder name equal your component name and each component typically should have its own folder, though it's not a hard rule but generally, it makes sense.

So here in the server folder, I will add my ServerComponent. Now a good naming convention is to have server, so the name of your component first, **then a dot and then component,** we will later learn that there are other building blocks in an Angular application too, so by using filenames like this, it's very easy to see what's inside the file. And very important, don't forget to add .ts, so for TypeScript because this is what we will write this component in.

So now we get an empty file for our new component, how do we create a component now? First of all, ***a component simply is just a class, a TypeScript class, so that Angular is able to instantiate it to create objects based on the blueprint we set up*** here you could say. So let's export this class so that we can use it outside of the file too because as I mentioned before, we're going to use our component in the app component for example to add it there. So we will add a class here and this class should now be named ServerComponent.

Again the naming convention here, have the name of the component first, server and then also the description, component because later we will also learn about other parts which might then be named differently in the Angular app. This is our class and right now, it's a normal TypeScript class, nothing special about it, we can't use it like this, Angular doesn't have all the information it requires.

**$ export class ServerComponent {**

**$**

**$ }**

So we should add something to it which tells Angular that this is not only a normal TypeScript class but instead something special, a component. We do this by adding a special decorator, ***decorators are a TypeScript feature which allow you to enhance your classes*** for example, enhance elements you use in your code, it's not restricted to classes but here we will use a class decorator. It's the component decorator and decorators are always attached by adding an @ sign in front of them.

Now this component decorator is not something TypeScript knows from the start, so we have to import it. We have to add an import and this import now needs to give us access to component, so we need to import component between curly braces because from the file where we're going to import this, we could import other parts too, so we have to specifically pick the component by using this syntax and then it's from and now the package where we import component from is @angular/core.

**$ import { Component } from ‘@angular/core’**

**$**

**$ @Component( )**

**$ export class SErverComponent{**

**$**

**$ }**

Now Angular ships with a couple of packages where it basically groups its functionalities and the core

package as the name implies gives us access to some of the core functionalities of Angular. So with that, we imported component, now this @component decorator is known to TypeScript, so when it parses this file and compiles it to Javascript, it is able to understand it, now we need to pass a Javascript object to this component decorator to configure it because without any configuration, it's still not that valuable to Angular. But here, we can set up some important information which will be stored as metadata for this class in the background then which will tell Angular what to do with this class and one important information piece is the selector.

So basically, the HTML tag by which you're able to use this component later in your other components templates, we will see this in action soon. The selector should be a string and here you may set up any name you want but you should make sure that it is a unique selector, that you don't accidentally overwrite a default HTML element or something like this. So typically, you prefix it with app- and then a fitting name, like server because it's a ServerComponent. This is my own selector by which I can now later use this component in my other components HTML files.

**$ @Component( {**

**$ selector: ‘app-server**

**$ })**

**$ export class SErverComponent{**

**$**

**$ }**

So with that set up, the other important piece we need to have is the template and here let's reference another external file, I'll come back to another alternative to this soon and this external file of course needs to be created. So in the same folder, I will create a server.component and now .html file, this will hold the template, the HTML code of my component here. So here, I can now basically add any content I want, like for example the Server Component to start simple and back to the TypeScript file, I can now point to this HTML file. We need a relative path for this, in the end this will all get bundled by webpack, so we need to tell it where to find the HTML file.

**$ @Component( {**

**$ selector: ‘app-server,**

**$ templateUrl: ‘./server.html**

**$ })**

**$ export class SErverComponent{**

**$**

**$ }**

Well and relative to the TypeScript file, the path simply is ./server.component.html, like this, this is pointing to this file. With this we created our first component, now to use it we need to dive into app module and understand what this does because we need to change something here to use our own component.

Let's do this in the next lecture.

**#### 17. UNDERSTANDING THE ROLE OF APPMODULE AND COMPONENT DECLARATION**

In the last lecture we created our own component, the server component here. It's almost ready to be used but to use it, we need to change something about our app module, what is an app module? Angular uses components to build web pages and uses modules to basically bundle different pieces, for example components of your app into packages.

Now this is kind of an advanced feature but later in the course, I will have a whole section about modules. For the majority of this course, we will only use the app module which is absolutely fine for a lot of projects, only in bigger projects you might think about splitting up your app into multiple modules. But what does this app module do, **what is a module then? Well it's as I said bundle of functionalities of our app and it basically gives Angular the information which features does my app have and use.**

As you can see, it also is just an empty TypeScript class, like our component ( **export class AppModule { }** ) and as the component, we transform it into something else by adding a decorator, here it's the @NgModule decorator which is also imported from @angular/core. Now in there, we see four properties we set up on the object we passed to ***@NgModule, declarations, imports providers and bootstrap***. We had a look at bootstrap, this was responsible for telling Angular hey which component should you be aware of at the point of time the whole application starts, so which component would you basically recognize in the index.html file and that was the app component.

Now I already mentioned that we won't add any more component selectors to the index.html file, that's just not how it works, therefore we won't touch the bootstrap array. Still, we added a new component to Angular and this new component now has to be registered here in the module so that Angular knows that this component exists because and this is important, by default Angular will not scan all your files here. So if you don't tell it that the server component exists, it doesn't know it, just creating the file is not enough. That is why we have to register it here in the @NgModule to tell Angular hey part of this module and therefore right here of our whole app since we only have this module is the server component.

We do register new components in this declarations array, we already registered the app component here because even though it's also used in bootstrap, it also is a part of our app, now we also need to add the server component here. Adding it like this is not enough though because now Angular would kind of know that this is part of the app but TypeScript wouldn't know where to find this type, where to find this server component.

**$ @NgModule({**

**$ declarations: [**

**$ AppComponent,**

**$ ServerComponent**

**$ ]**

So when we try to bundle our code, we would get an error which you can actually see if you save this, triggering the recompilation, you see that we get an error in the terminal down there, that it can't find server component because and this is not Angular, this is TypeScript, it doesn't know where this file is, where to find this class.

So we need to add an import at the top manually pointing to this. So we add server components here as an import from ./server and then here in the server component and you omit the .ts, you don't add .ts to your import, it's just server.component, so the file name without the extension, the extension is added by webpack which bundles our project automatically.

$ **import { ServerComponent } from ‘./server/server.component’ ;**

So with this and it's just so important, this is not Angular, this is a TypeScript feature, now TypeScript knows where to find this, therefore everything can be bundled, so therefore now Angular when it runs knows that we have a server component and that it is part of our app and with this, we can now finally use our component.

We will do this in the next lecture but maybe you're also wondering what imports and providers does here in the app module. I'll come back to providers later in the services section, imports simply allows us to add some other modules to this module because I told you, you can split up your app into multiple modules and then you can import these modules to basically make this module a bit leaner and outsource some stuff into another module. Now I told you that we are only going to use one module, so what are we importing here? Some modules built into Angular because Angular itself is split up into modules, so the browser module for example gives us all the base functionality we need to start our app and forms and HTTP module, that is something I'll come back to in the forms or HTTP section of the course. So with this, the basics about this module should be clear, again we'll dive deeper into this later in the course. We also registered our own component, let's now use it in the next lecture.

**#### 18. USING CUSTOM COMPONENTS**

With our component registered, let's use it. So we have our selector here, **“app-server”.** Now I told you we are not going to add it to the index.html file, there it wouldn't work instead let's go to the “**app.component.htm**” l file, the only other HTML we have right now where we could add it and here I'll add a horizontal line and below this, I'll add my “app-server” element here and the shortcut I used here is just a little plugin called **emmet**, available for many IDEs and editors which gives you a quicker way of writing HTML. So here, app-server is our own selector, our own element kind of we created. If we now save this, it should successfully recompile and if we have a look at our running application, you see the server component here, our own component being added here. So that is how you can use your components then with their selector after adding them to app module and then displaying them on the page.

**#### 19. CREATING COMPONENTS WITH THE CLI AND NESTING COMPONENTS**

In the last lectures, we learned a lot including the creation of a new component by hand.

Now there also is an alternative to creating it manually. When using the CLI, you can open a new terminal window or command line, leave the ng serve window running, that's important, don't close it and in this new window we can use some CLI command to also create a new component.

Let's say we want to have our single server components here in another component, so nested into another component, a servers, multiple servers component. We can create this manually by adding a servers folder with our component or we use the CLI with **“ng”** and there we have a generate command, which allows us to generate some elements, some elements supported by Angular. One such element is a component, so we can create a new component with “***ng generate component and then passing the name***, servers like this. You can also use ***a shortcut and instead of ng generate component, you could say “ng g c servers”*** which does the same.

So if you hit enter, what this will do is it will give you a new folder in your app folder with the name of your component, servers and in there, you basically find the same files as you had for the app component, HTML, CSS, TS and this spec file which basically is used for testing. Now I will remove this, we will later have a section about the testing and we'll have a look at the styling later too, for now let's say it is in this servers component that we want to output our single server.

Well we already have this selector, app server, so in this template of the servers component, we can overwrite the default text with app server and maybe simply replicate this so that we have two components because remember, components are reusable. We can use the same component as often as we want by simply using its selector as often as we want. So now we have two servers here and with this in our servers component, you need to make sure that you also update the app module. Now the CLI should do this automatically which is why you already find the import and the declaration in this file but make sure you really do have it here. With this, we can go to our servers.component.ts file to check the selector which is app-servers and then we can go to our app.component.html file and update the selector to use this new component instead of the old one.

With this if we now save this and go back to our application, it looks almost the same as before, we see

our single server component twice but if we inspect the code here in the developer tools of Chrome for example, you'll see that we have this outer app servers component, this newly created component and in between, we have two single app server components. So with that, we learned a couple of important things about how to create components manually or with the CLI, how we can nest them into each other, that we can use components selectors in any other template of any other component and how we can replicate components by simply using them multiple times, using their selectors multiple times.

Let's dive into how we can actually style them, what we can do regarding the style and the template

of components.

**#### 20. WORKING WITH COMPONENT TEMPLATES**

Thus far, we always used an external template file and we haven't done anything regarding the styling

of components, time to change both. Now instead of using an external template file, you can also use an inline template which means you define the HTML code in the TypeScript code, sounds strange?

Well it's possible. We can do this by simply going to our servers components here for example and right

now, we point to this HTML file here, now we can change “templateUrl” to template, one of the two has to be present, either linked to an external template or just the template to define it in this file but each component needs to have a template, this is the one property you have to have at all times. So with that, now I change it to template, now we can write our template code here. So here we could now replicate app-server, like this, maybe twice, we have to make sure that we don't wrap this line because this is not supported in a default TypeScript string and if I now save this, we see the same result as before.

**$ template: ‘<app-server></app-server><app-server></app-server>**

So it looks like nothing changed and that is good in this case because we try to make it look like this

but actually, we did change the way we set up our template here, now we're defining the HTML code in this same file. And actually if you want to write multi-line strings here, which you probably want ***if you add more HTML code to it, you can switch from a normal string with single quotation marks to using back ticks to use Javascript template expressions***, to be able to write multi-line strings in between there now.

So now between the opening and closing back tick, you may actually wrap the line and now write your HTML code, like this. What should you use, should you use this approach here of using template or an external file? It depends, if you have not that much code, not that much HTML code, using an inline template like this might be fine, then you have all the logic in the TypeScript code and that is absolutely OK but as soon as you have let's say more than three lines of code in the HTML part here in the template,

using an external file is a good idea to make it easier to follow and easier to understand. It is important though to keep in mind that you can use this approach, that you can inline your templates and very important, that you need to have at least a template in a component. You can omit the selector, you will later in the routing section learn a different way of loading components without using their selector and you don't have to add styles but templates have to be present. So that was templates, now let's have a look at the styles in the next lecture.

**#### 21. WORKING WITH COMPONENT STYLES**

We understood that templates are important and that we can either use them inline here or point to an external file, what about styles? We also have this **“styleUrls”** property here and we have these CSS files. What can we do with those? Let's go back to our app component for this. Here we got a h3 tag and a horizontal line and if we have a look at our application, it's not that pretty, right? The I'm in the app component text here or all the text actually is sitting right on the edge here, not super pretty. Now since we use bootstrap, of course we can use some bootstrap classes to enhance this.

So we could add a div with a container class around it, then add a row and this is just using the CSS framework, bootstrap, so these are some CSS classes provided by this framework, nothing Angular specific and then we could add a column here, xs-12 to have one column on all device widths. Now if we add this, we added some styling, some normal CSS styling, nothing Angular specific and our app looks a little bit nicer now.

This is great but we can also use some tools given to us by Angular. Let's say this, I'm in the app component heading here, we want to change that style, maybe we want to give this some blue color.

We can do it just by going to the app.component.css file which defines the styling for this component

and now we can use or write normal CSS code, so overwrite the style of the h3 tag and set its color to some blue, so maybe to this dark blue here.

If we now save this and we have a look at our application, you see it's dark blue now. This is how you can use this style file, this CSS file referenced in the app component here to define the styles for a component. Now just like we were able to choose between external file and inline code for template, we can do the same for the styling. You notice that styleUrls is an array here whereas templateUrl wasn't. It is this because we could reference multiple external stylesheets here, so you could add pointers to other stylesheets but besides that, I'm going to comment this out, you can also set up another property which is just styles. This also takes an array but now it takes an array of strings where you define the style in this string, so now in this file.

Here I'm going to **use back ticks again to be able to write a multi-line expression** here and then we could set up h3 here and to really show you that this is different, I will set the color to now let's not say dark blue but to dodger blue. And if I now save this and we go back to the application, you see the blue updated because now the inline style is taking over, I commented out the other one. Just like with the template, you have to decide which one to use, you can't combine styleUrls and styles but both important here has to be an array, that's the difference to template, you must only have one template, you may have multiple style files or style definitions and you can choose between inline and external file. The rules are basically the same as for the template. If you have more code in there, using an external file is a good idea, if it's a short style definition like this, why don't you just put it into the TypeScript file? This is how you can work with templates and styles.

**#### 22. FULLY UNDERSTANDING COMPONENT SELECTOR**

So that actually works like a CSS selector and therefore, you are not limited to selecting by element.

You could put this into square brackets to use the attribute selector, so in CSS, you can select elements by attribute by enclosing that attribute in squared brackets and now if you save this, you would see that our app actually is broken because if we have a look at the error message, app servers is an unknown element because now, our code in Angular doesn't recognize app servers anymore because we change the selector to be an attribute.

So to make this work again, we would have to comment this out or remove it and maybe add a div or any other element, a normal HTML element which has an app servers attribute now. Now with this custom attribute added now, now the app works again because now, Angular selects the element by attribute and not by the element itself because we changed the selector. Another alternative is to select by and I'm just going to comment this out so that the code is still there for reference, to select by class, so with a dot at the beginning, .app-servers, again just like in CSS. So here, we can now select by class, so let's add a new div here which has a CSS class of app-servers, which we of course could now also Style but besides that, it's also recognized by Angular as a selector which is why we still see the app.

Now these are all options you have, as a side note, selecting by ID won't work, that's not supported by Angular and all those pseudo-selectors like hover and so on also don't work and you typically use the elements style here, app servers for components, we will soon learn about directives, another feature where this is different but for components, you typically create your own elements and therefore you don't use that style and you don't use the attribute style, instead you use app servers like an element but it is important to understand that you are not limited to this and there might be use cases where you want to use a different selector.

**#### 23. Assignment**

-Create two components **“Warning component”** & **“Success component”**

-output them beneath each other in the AppComponent

-Output a warning or success message in the components

-Style these components

Solution

-Go to the app folder, right click and create a folder

-Give the name the folder **“warning-alert”**

-create a file inside the new folder created with name **“warning-alert.component.ts”**

-inside the component file, create a decorator

-the decorator will be named as **@Component**

-inside the decorate create a **selector** and **a template**

-use back ticks for the template and create a p tag with text in it.

-make sure export class **“WarningAlertComponent”**

Now another way to create a component through CLI

-Go to your folder app folder in the command prompt

-Type **ng generate component success-alert** or “**ng g c success-alert”**

-now to go the app component and wrote those two components underneath each other

-type <app-success-alert></app-success-alert>

-type <app-warning-alert></app-warning-alert>

-You will get a warning/error message

-in the app.module add the **warningAlertComponent** which you created in the warning-alert.component.ts with export and class

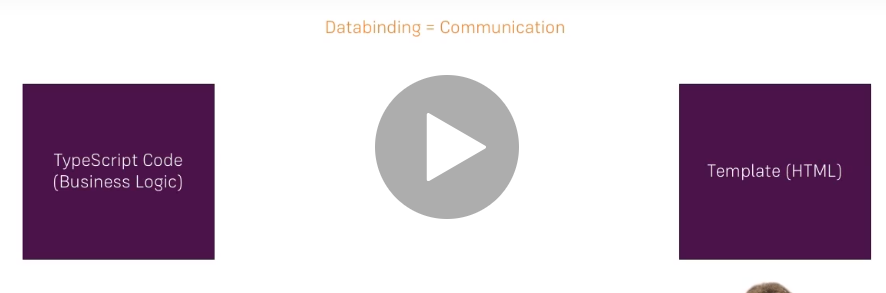
-in the app module.ts **import the warningAlertComponent** from the folder path

-you don’t have add the ts at the end, by default IDE does add this import for you, if you don’t see this add it manually

**#### 24. WHAT IS DATABINDING**

In our little basic first application we're working on, where we added a couple of components, we covered a lot of key aspects of a basic Angular app, how it starts and how to use components, the base building block of any Angular app. But thus far, we only output static content, the server component here, that's not super interesting, we did even more than that in the first section of this course.

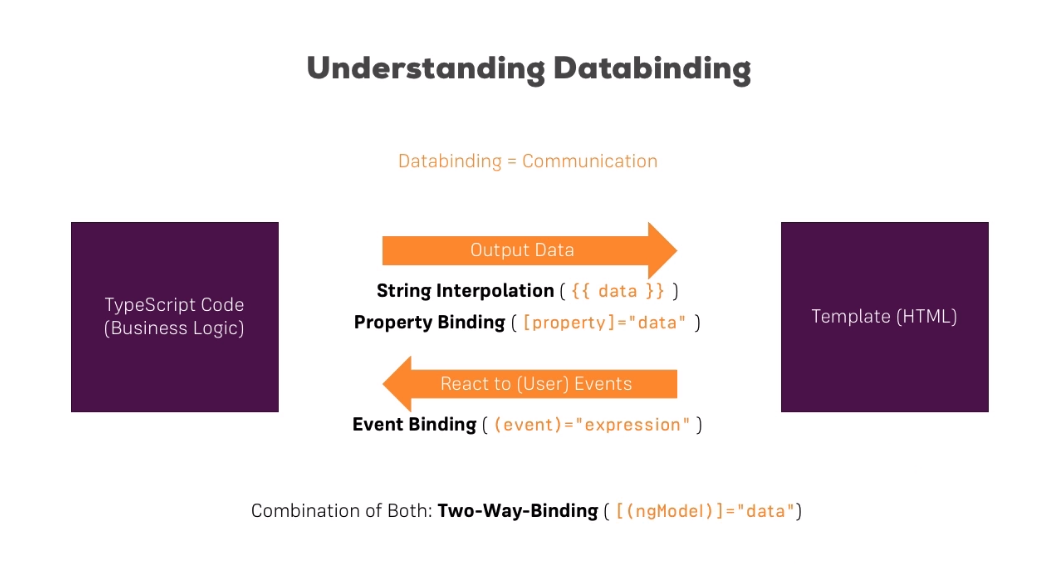
So let's change this to output some data which is kind of dynamic at least, which is not hardcoded into the template, time for some data binding. What is data binding? You could basically translate data binding with communication. Communication between your TypeScript code of your component, your business logic and the template, so what the user sees because you might have some result in your TypeScript code because you fetched something from a server or finished some calculation which you want to display to the user and the only thing the user sees is the template.



So we need some kind of communication between both pieces to be able to really do something in our app, that is where data binding comes into play because it is responsible for this communication. We get different ways of communication now, for example we want to output data from our TypeScript code in the HTML code in the template. We can use **string interpolation** for this, you saw this in the first section of course, it's the syntax with the double curly braces and then some property name or some expression in between or property binding. We will have a closer look at all these forms of course in the next lectures, so we will learn what this then is and how it works. The syntax basically uses these strange square brackets around HTML attributes but again, I will come back to this.

Sometimes though, the other direction is interesting too, if the user clicks a button, you can kind of think of the user clicking the button on the template because again, the template is the thing with which the user interacts, which he sees and where he is also able to click buttons. So if we click a button, we maybe want to trigger something in our TypeScript code, so now we need the other direction and we can get this other direction, we can react to user events with event binding.

So we can bind to for example a click event to execute some code whenever it occurs and we also have one additional form of data binding where we combine both directions, two-way data binding, where we are able to react events and output something at the same time. Might not be super clear how this works but that is what you will now have a detailed look at all four forms in the next lectures.



**#### 25. STRING INTERPOLATION**

Let's start by having a closer look at string interpolation. Here in the server component, I don't want to output this here in the HTML code like this, I don't want to hardcode the output here. It would be realistic that our server component contains information about a specific server. So maybe here in the template file, we want to output server with ID and then some “dynamic ID” here is and then the status, like “offline”.

So we have two dynamic fields here. Now what makes sense to kind of derive them in TypeScript code, in a real app that might be derived from a HTTP request to make some back-end or some calculation, for now it will hardcode it in here. So we might have a **“server ID”** and I will assign a **“value”** of let's say 10, so a number and I might also have a **“serverStatus”** which is a string offline, so two different types, a string and a number and though not necessary, you could also be very explicit by assigning the type with TypeScript's feature here of adding a colon after the property name and then the type, it will infer it automatically though, so this is not required, I'm just doing this here for demo purposes.

**$ export class ServerComponent {**

**4 serverId: number = 10;**

**$ serverStatus: string = ‘offline’;**

**$ }**

So now I want to output these two properties in my template and to get this connection, we need data binding because you learned it's all about the communication between your TypeScript code and the template. So string interpolation is a typical use case here, a typical solution for this task here, instead of the placeholder, I use my **“double curly braces”**, opening and closing, this is the string interpolation syntax and in-between these double curly braces, you can now write a TypeScript expression.

So the easiest expression is to simply reference a property, like the **{{ serverId }}** and this is a common use case, that you only output a property here or the value of the property. However, it's not the only use case, you could also for example here for server simply hardcode a string in there, any expression which can be resolved to a string in the end, that's the only condition for a string interpolation syntax here.

So whatever you have between the curly braces, in the end it somehow has to return a string, so you could call a method here which returns a string in the end. The only other restriction is you can't write multi-line expressions here, you can't write block expressions in here, so you can't add an if or for control structure in here, you could use a ternary expression though.

So this is what you can do, now let's also replace the last placeholder and here, I will again just reference a property, the server status.

**$ <p>{{ ‘Server’ }} with ID {{ serverId }} IS {{ serverStatus }} </p>**

Now this is string interpolation in usage and you'll see that we have an expression which is a string itself and then two other oppressions which simply point to properties and again, you could also call a method here which returns a string. Now interesting, the server ID is not a string but a number and I just said that string interpolation has to resolve to a string in the end, has to get a string in the end, so let's see if this works. If we save this with ng serve still running, let's also change app server back to app servers to make this work again, we see server with ID 10 is offline. We see it twice because we repeat this component and each component of course has the same content but we see that string interpolation is working and that it especially is also working here for our number.

So ID 10, that is not an issue, that is working fine because a number can be easily converted into a string.

So yes, you have to get a string in the end or something which can be converted to a string to really

be correct here. This is how you can use string interpolation and now to really complete the example, let's say the serverStatus here is somehow returned in a method, getServerStatus could be the method name,

this is how you define a method in TypeScript, so you might know this kind of method definition from other languages and here we simply return this serverStatus, so we return the property here. We could of course have accessed it directly as we already did but just to demonstrate that you can call a method,

here I am calling a method, **{{ getServerStatus( ) }}** of course leads to the same output because again, all this

method does is it returns us a string in the end or something which can be converted to a string, that

is string interpolation in action.

**#### 26. PROPERTY BINDING**

In the last lecture, we had a look at string interpolation which is a great tool for out putting data

in a template, now I want to have a look at property binding. Now one important information by the way, there are a lot of times where you can either use property binding or string interpolation

and I will show what I mean in a second.

Let's go to our ”servers component” where we manage all the servers and I want to allow the user to add new servers, at least soon I want to allow that.

So we will need a **<button >**where I say “**add server**” on it and I will just add some CSS classes, “btn” and “btn-primary” to make this part look nice, this is all using normal bootstrap CSS classes though, nothing to do with Angular and to reflect this change in my “servers component” here (template: ), I also want to comment out the inline template and point to the external template again, so “templateUrl” should point to my “servers.component.html” file, like this. So now with this, if I save this, we see that button here. Right now, it doesn't make any sense that I'm able to click this button because it doesn't do anything, we haven't learned to react to a button click yet.

So I will add a new property in my TypeScript code here which I'll name allowNewServer and set it

to false. ( under export class )

**$ export class ServerComponent implments OnInIt{**

**$ allowNewServer = false;**

**$ constructor ( ) { }**

So now this is also another property holding a boolean value, so true or false, in this case false because I don't want to allow the user to create a new server and you could again imagine that this is somehow derived dynamically , we will soon work with some dynamic data.

So in the HTML file here (servers.component.html), I now want to disable the button and as you probably know, there is a disabled attribute you can add, like this.

**$ <button class=”btn btn-primary” disabled> Add Server </button>**

If we now save this, the button is disabled, I can't click it. That's nice but that's hardcoded into HTML, now maybe this allowNewServer code here changes however, maybe it's not set to this all the time.

So then, we wouldn't be able to react to this and I can actually demonstrate this by for now in the constructor which is simply a method executed at the point of time this component is created by Angular, by there calling **“setTimeout”** ( inside the constructor ), a normal Javascript function and here I define after which period of time or after how many milliseconds something should happen, so after 2000 milliseconds which are two seconds, I want to execute a function and here I'm using an ES6 arrow function.

**$ constructor( ){**

**$ setTimeout( ( ) = > , 2000); }**

This syntax might look strange, it's almost the same as function like this, with some difference when it comes to handling the this keyword. But in the end, here you passed the arguments and here you have the function body, more importantly in here, I can set **“allowNewServer”** and now this would not work in the other syntax because this would then be referring to something else, can set it to true.

So after two seconds, this will be switched to true and since we hardcoded this,

**$ constructor( ){**

**$ setTimeout( ( ) = > {**

**$ this.allowNewServer = true } , 2000);**

**$ }**

of course this doesn't affect our button here at all, it stays disabled. So I want to bind it to allowNewServer.

So now to make this “**disabled”** functionality dynamic, we can bind to it by enclosing it in square brackets. Square brackets indicate to Angular that we're using property binding, that we want to dynamically bind some property and disable the HTML attribute, in the end, in normal HTML, only sets a specific property on the underlying DOM element. You might note that each HTML element you use is parsed by the browser and kind of translated into an element on the document object model, this is totally unrelated to Angular and therefore we have an element in this DOM and this element has a couple of properties.

A lot of these can't even be set through attributes on the HTML element. Well one of the properties is the disabled property and you can set it through the disabled attribute but here we're not using the disabled attribute anymore, with the square brackets, we are directly binding to this native disabled property this HTML element has.

So now, we can set this equal to and now between the quotation marks here, an expression which for this property resolves to a Boolean but for other properties, another type might be required because again, you can bind to a lot of properties, basically to all the HTML element properties. We'll soon learn that there are also other properties you can bind to.

So for now I can simply pass allowNewServer here,

**$ <button class=”btn btn-primary” [disabled]=”!allowNewServer”> Add Server </button>**

my property name because this will resolve to true or false, exactly the value the disabled property needs because it has to decide whether it should be disabled or not. So now with that, if we reverse this, so by adding an exclamation mark, we made sure that this is only disabled if this is false, so if allowNewServer is not true ( ! allowNewServer ), otherwise you would do exactly the opposite. So now with the exclamation mark, if we go back to the app, you see it's disabled but after two seconds, let me reload this, it becomes enabled because now we're binding to this disabled property, this native element property, we're binding this to our own TypeScript property here and the convenient thing is and this is what Angular is all about, that this will update dynamically. So once allowNewServer changes, this is changed in the DOM,

so now the button gets enabled. One of the main reasons why you use Angular, it makes it very easy for you to interact with your DOM, to change something there at runtime like we do here. That is property binding in place, now as I mentioned besides binding to HTML element properties like we do here, you can also bind to other properties, for example of directives, something we haven't learned about yet and your own component, something I will also show you later. So this is property binding. Before moving on the other two forms of data binding, let's find out where you could actually use string interpolation instead of property binding

**#### 27. PROPERTY BINDING VS STRING INTERPOLATION**

**<p> {{ allowNewServer }} </p> this is string interpolation**

**<p [interText]= “allowNewServer”> </p> this is property binding**

In the last lecture, we learned about property binding, now if we had the use case that we also want to output the current value of allowNewServer, we could of course use string interpolation to simply output allowNewServer and a boolean can also be cast into a string. So now we see false here and after two seconds, you see this changed to true because we change it in the code(inside servers html). Nice.

Now this is one of the cases here where you can easily use property binding instead of string interpolation.

So you could simply bind to a property of this element, the “innerText“ property and set this equal to

“allowNewServer”. Now I can remove the string interpolation here inside of the text and whilst it might look empty now, well we set in our text so we get the same behavior as before because the innerText property of an element is just what's between the opening and closing tag (square bracket) , so in this case, we were able to easily replace string interpolation with property binding. So when should you use which of the two?

Well basically if you want to output something in your template, print some text to it, use string interpolation, if you want to change some property, be that of a HTML element or as you will later learn, of a directive or a component, typically use property binding, that is how you can differentiate it and you will get a feeling for this once you work your way through the course project and so on.

So this is this, one important note I want to make is don't mix property binding and string interpolation. You might notice that here, we have disabled equals and then we have quotation marks and then directly, our property name. There are no curly braces between these quotation marks and there shouldn't be because that will break the app, this will not work. Between the quotation marks of property binding, you can and you must already write TypeScript code so to say, a TypeScript expression which will return the value this property expects, so for disabled some expression which returns true or false.

So just like with string interpolation, you may also call a method there but you must not add curly braces in there. Whilst it might look we simply use a HTML attribute there. We aren't, this whole syntax here is a syntax recognized by Angular, so this is why between the quotation marks, you can already and you have already, you have to write TypeScript code because again, this whole expression is evaluated by Angular, mixing in string interpolation will break it, string interpolation only works in a normal template, not within another expression of that template, not within a property binding or something like this. With that, enough about property binding and generally about outputting data in the template, let's see how we can react to events.

**#### 28. EVENT BINDING**

In the last lectures, we learned a lot about how we can use property binding and string interpolation to output data in the template, let's now react to events.

For example we have a button here (servers html) and after two seconds, this button is enabled, so it would be nice we could do something once we clicked the button. Now I will soon show you how we can actually add a new server, how we can add a new replication of our server selector, of our server component here, for now what I want to do is I want to output something.

I will remove this text here (the p tag from previous interpolation) and instead here, I want to output the value of a property. I'll create this property and again you may ignore ngOnInit for now, I will come back to this, I will name it **“serverCreationStatus”** and initially I will set it to “no server was created”, like this.

Now this property can be output via string interpolation, “serverCreationStatus” and I just told you I don't want to continue outputting data, we want to listen to events. So I promised it, we'll do it. I will add another method below here and I will name this **“method onCreateServer”**, the on at the beginning to kind of make clear that this will be triggered from within my template, you don't have to name it on something but it kind of makes it easier to understand who will call this method.

Well, the user will basically by doing something, some event will call it, that is what I want to symbolize with that. So in here, I will set the serverCreationStatus to server was created. So the missing piece of course is to call this method and now here is where event binding comes into play. We want to listen to this click event here and typically, you have the onClick attribute on the HTML element where you can now execute some Javascript code. We won't use that with Angular though, here we have the syntax Angular offers us, event binding which uses parentheses, like property binding used square brackets, for events parentheses are the signal that we're using event binding and here, we have to specify the name of the event between the parentheses, for example “**click**” and you can basically bind to all the events made available by the HTML attribute you're placing this on.

**$ <button**

**$ class=”btn btn-primary”**

**$ [disabled]=”!allowNewServer”**

**$ (click)= “ “> Add Server**

**$ </button>**

So if you have onClick, onMouseEnter, you could bind to click mouse enter without the on at the beginning. So we bind to the click event and now between the quotation marks, we put the code we want to execute once this event occurs. Now often times, you will call a method here, so let's do this. Let's point to the method we want to execute which is (**onCreateServer)** with parentheses, so this method will now get executed whenever we click the button.

As a side note, you don't have to execute a method here, you could also put the code you want to execute right between the quotation marks. Oftentimes this is not very good because you want to execute more code if some event happens but if all you want to do is set some boolean property to true for example, well you can definitely do this inline the template, as a general rule though, you shouldn't put too much logic into your templates though. So now we're binding the click event and we execute onCreateServer method when it occurs, so let's see this in action. In our application, let's reload the page, the button gets enabled and now watch this part here, if I click the button, it changes to server was created because we are triggering our own method here. And this is how a simple event binding works, use parentheses, the event name in between and then between the quotation marks, the code you want to execute when this event occurs.

**#### 29. BINDABLE PROPERTIES AND EVENTS**

How do you know to which Properties or Events of HTML elements you may bind? You can basically bind to all Properties and Events – a good idea is to “console.log( )” the element you’re interested in to see which properties and events it offers.

IMPORTANT: For events, you don’t bind onclick but only to click (=> (click) ).

The MDN ( Mozilla Developer Network ) offers nice lists of all properties and events of the element you’re interested in. Googling for “**YOUR\_ELEMENT properties**” or “**YOUR\_ELEMENT events**” should yield nice results.

**#### 30. PASSING AND USING DATA WITH EVENT BINDING**

In the last lecture, we had a look at event binding, now before focusing on two-way binding and explaining what this is, there's one other important thing you need to know about event binding.

Let's say before this button, we also have a “**label**”, server name and then more importantly, we have an “**input**” which we maybe give a **bootstrap class of form-control.** Here I want to allow the user to enter the name of the server which should get created. So here, we could listen to the input event,

$ **(input) = “ onUpdateServerName( )“**

this is a normal DOM event provided by the input element which is fired whenever the user types

and here we could execute “**onUpdateServerName**” because it will be fired on every keystroke.

So let's add this method to our component here and now I somehow want to output what the user entered right now, what the value of this field is. Now we can do this by passing **“$event”** here

$ **(input) = “ onUpdateServerName($event )“**

And $event super important is kind of a reserved variable name you can use in the template when using event binding. For this event, so only between these quotation marks here, $event will simply be the **data emitted with that event**. So input and click are default events provided by the DOM you could say

and they ship some data when they are fired, the click event gives us an object which for example holds

the coordinates where we clicked and the input event also gives us some data, some information about the event.

Now we can capture this data with “**$event**” passed as an argument to the method we're calling or used anywhere between these quotation marks in the code we're executing, $event, something to keep in mind, **reserved word which gives us access to event data**. So now we pass this to **onUpdateServerName(event: any),** so here we know we will receive this event and this will be of type any for now. So now let's simply log it to the console, so that we can see it in this debug log.

$ **onUpdateServerName(event: any){**

**$ console.log(event);**

**$ }**

If we do this and I now type something here, watch the console on the right. A couple of outputs were written there, four to be precise because I typed four characters, did four keystrokes, so the input event was fired four times and if we have a look at this event, we get a couple of information here, for example the target and the target simply is the HTML element on which this event occurred. So here on the **target,** since this is an input element, if we scroll down, we also got the “**value**” test and this is what the user entered.

So what we can do now in our TypeScript code is we can set a “**serverName**” property ( serverName = ‘ ‘)which is an empty string by default and then here in “**onUpdateServerName**”, instead of logging it, we could say this server name and we just learned that on the event which is of type event as we now also saw, here we have a target and on the target, we have a value.

**$ OnUpdateServerName( event: Event ) {**

**$ //console.log(event)**

**$ this.serverName = event.target.value;**

**$ // this.sererName = (<HTMLInputElement>.event.target).value;**

**$ }**

Now this is only available because the target is of type input element though. This is why my IDE doesn't like it, we can explicitly inform it about the type in TypeScript by adding HTML input element here in front of it, like this and again this is just needed to inform TypeScript that we know that the type of the HTML element of this event will be an HTML input element.

We do this explicit casting with this syntax, with the opening and closing tag, with the smaller and greater than sign and the type in-between. “**(HTMLInputElement>event.target).value**;”

So with that we assign this value to our server name, what we can now do is in the HTML file here,

we can output this for now. So with string interpolation, we could output the **{{ serverName }}** below the input. If we save this and go back to this application and I type test server here, you see this dynamically updates below the input. This is how you can use this $event object to fetch the event data. Now with that, let's move on to two-way data binding to kind of finish this little example app here for

now.

**IMPORTANT: FormsModule is Required for two-way-binding!**

Important: for two-way-binding( covered in the next lecture) to work, you need to enable the “ngModel” directive. This is done by adding “FormsModule” to the “imports[ ]

[” array in the AppModule.](mailto:\” array in the AppModule.You then also need to add the import from \“@angular/forms)

[You then also need to add the import from “@angular/forms](mailto:\” array in the AppModule.You then also need to add the import from \“@angular/forms)” in the app module ts file

$ **import{ FormsModule } from ‘@angular/forms’;**

**#### 32. TWO WAY BINDING**

In the last lectures, we learned a lot about event binding and property binding, now let's combine both to two-way binding. We learned that we can get the event data like this for example, there is another approach too though. If I copy this(input) and I comment it out and now I replace this input event here with something else, you will find that we have an even easier way of binding to some data. With two-way data binding, we combine property and event binding, we use it therefore by combining the syntaxes, square brackets and within these, parentheses( **[ ( ngModel ) ]** ) .

Now here, we have to use a special directive and we will learn more about directives in a second, @ngModel. Now we can set this equal to some property defined in our TypeScript code, so of to the server name for example.

$ **[ ( ngModel ) ] = “serverName”**

This set up will do the following, it will trigger on the input event and update the value of server name in our component automatically. On the other hand, since it is two-way binding, it will also update the value of the input element if we change server name somewhere else and I can demonstrate this by going back to the TypeScript code and setting server name to Testserver here initially.

$ **serverName = ‘Testserver’** ;

So before that was an empty string, now it isn't anymore. With this if we go back to the running application, you see that the input is pre-populated with that. Now if for demo purposes, I comment in the other input which does not use two-way binding, you'll see that this input is empty because again it's not using two-way binding.

**$ //commented code below**

**$ <input**

**$ type=”Text”**

**$ class=”form-control”**

**$ (input)=”OnUpdateServerName($event)”>**

Now if I type something here, you see it updates the property name and hence it updates it there where we use string interpolation and in the other input but if I type here, it's not getting changed in the first input because again, we're not using two-way binding there. This is what two-way data binding is, a very simple way of binding in both directions and for example of reacting to changes in the input value here.

So I will comment out the first approach, I will leave the method in here though, even though we're not calling it anymore but with that you now also had a look at two-way binding, a nice and easy way of reacting to events in both directions. Now let's fine tune our app here a little bit more and let's then move on to directives, another core feature of Angular.

**#### 33. COMBINING ALL FORMS OF DATABINDING**

So with all four forms of data binding covered now, lets finish our little demo app for now. We get the server name and this is live updated while we type due to two-way data binding in place here (this.servername) I will comment out this server name **{{p tag serverName}}** here, instead I want to display the server name once we clicked this “Add Server” button.

So once we create the server, we should not only say server was created but also name this and then simply add this server name like this.

**$ onCreateServer( ){**

**$ this.serverCreationStatus = ‘ Server was created! Name is ‘ + this.serverName;**

**$ }**

so now with this in place, we’re now using the data we’re changing through two-way data binding or if we chose to do so, through one-way data inding if we use the other approach which is currently commented out with this in place, now you can see that if I change the server name (Testerserver 2 in the input) and click add server, we see server was created, now is Testserver 2.

With that, we’re using all four forms of data binding together with each other, event biding to listen to the click, string interpolation to output the data here, property binding to enable this button in the first place after two seconds and two way binding to fetch our input data.

So with that we’re pretty advanced into the basics of Angular, we use components, we know how they work in general we use how to communicate with our templates from our TypeScript code and in the other direction. Now there is one more basic feature we have to cover to really able to build first little applications before we then can dive deeper into the individual building blocks.

**#### 34 ASSINGMENT ( project name basic-assignment-2/basics-assignment-mysolution )**

1. Add input field which updates a property {username} via two-way-binding
2. Output the username property via String Interpolation ( in a paragraph below the input)
3. Add a button which may only be clicked if the username is NOT an empty string
4. Upon clicking the button, the username should be reset to an empty string.

**FOR THE BOOTSTRAP TO WORK**

Download bootstrap locally in that project by typing npm install –save bootstrap@3

Once it is installed you need to make the changes in the “angular.json” file look under the build you will see script array.

In that array add “node\_modules/” right above src/styles.css. This node modules is available if the “node modules folder, scroll down to bootstrap / dist folder, css folder, bootstrap.css folder and bootstrap.min.css . so you will add “nodule\_modules/bootstrap/dist/css/bootstrap.min.css”

Solution for the above assignment is under the folder “basics-assignment-mysolution”

In this course, we use version 3 of the framework, install it via “npm install - - save bootstrap@3”

🡺3 is important

Additionally, when using a project created with Angular CLI 6+ ( check via ng –v ), you’ll have an “angular.json” file instead of “.angular-cli.json” file. In that file, you still need to add Bootstrap to the “styles[ ]” array as shown in the next video but path should be

$ node\_modules/bootstrap/dist/css/bootstrap.min.css

Do not use . . ./ node\_modules

**#### 35. WHAT ARE DIRECTIVES?**

Directives are instructions in the DOM

**$ < p appTurnGreen > Receives a green background ! </p>**

**$ @Directive({**

**$ selector: ‘[appTurnGreen]’**

**$ })**

**$ export class TurnGreenDirective {**

**$ . . . . . .**

**$ }**

We're almost done with the basics about Angular, we learned a lot about components, the basic building block of any Angular app, about how it starts the app, how we can communicate with our templates in our components so that we can output data or react to events, now there is one other key building block you probably use in any Angular app you build, directives.

**What are directives?**

Directives are instructions in the DOM and we actually already use directives without knowing it,

components are kind of such instructions in the DOM. Once we place the selector of our component somewhere in our templates, at this point of time we're instructing Angular to add the content of our component template and the business logic in our TypeScript code in this place where we use the selector.

This was our instruction, Angular please add our component in this place and indeed components are directives but directives with a template, there are also directives without a template. So an example would be the appTurnGreen directive

**< p appTurnGreen > Receives a green background ! </p>**

which would be a custom directive we could build. We typically add directives with an attributes selector but technically the selector of a directive can be configured just like the selector of a component, so you could also use CSS classes or the elements style but again typically use the attribute style and on this paragraph, this directive might simply color the text green you could say.

So Angular would find this instruction, here

**$ @Directive({**

**$ selector: ‘[appTurnGreen]’**

**$ })**

**$ export class TurnGreenDirective {**

**$ . . . . . .**

**$ }**

we would have defined our directive with the directive decorator to inform Angular that this class holds a directive and there, we might have the logic to turn this green. Now we will learn how to write our own directives in the section focused on directives in this course but there are a couple of built-in directives which are really useful. Let's take a closer look at how we use them and what they do for us in the next lectures.

**#### 36. USING ngIF TO OUTPUT DATA CONDITIONALLY**

In the example application we built thus far, we haven't used any directives besides components which are directives, that's important but we haven't used any other directives, no built-in directives. Now let's say one thing we want to do is we only want to show this server was created text here, there is no need to have this no server was created text to be displayed ( No server was created! ) . So it would be nice if we could conditionally show this message and for that, we need some help, we can use a directive shipping with Angular, the ngIf directive.

Now it works like an if statement and to be precise, it works like this. In our “**servers.component**” here where we output the server creation status,

$ **// <p>{{ serverCreationStatus }} </p>**

I will comment this out so that we can still reference this code but now I will add a new paragraph where I simply say server was created, server name is bind my server name.

**$ <p> Server was created, server name is {{ serverName }} </p>**

Now with this in place, it will still work but now we always see with every keystroke how we change this server name, not really what I want, instead I want to output it as soon as we click this button here.

So what I can do is I can add a directive here and as I said, typically directives are added by using an attribute selector and pretty much all the built-in directives use that selector, ngIf does and then ngIf is added by adding a star, this is important, ngIf. The star is required because \***ngIf is a structural directive which means it changes the structure of our DOM**, it either adds this element or it doesn't add it. So that's just some extra information for Angular, the directive itself is just ngIf but the star is required, without it, it will not work correctly.

$ **<p \*ngIf>** **Server was created, server name is {{ serverName }} </p>**

So ngIf and then we can set up our conditions here between the quotation marks. For ngIf, this has to be any expression returning true or false, deciding whether this should be added or not. So here it would make sense to add a new property which we name “**serverCreated**” and set it to false and we set this to true once the button is clicked.

**$ export class ServerComponent implments OnInit{**

**$ ………………….**

**$ ………………….**

**$ serverCreated = false;**

**$ }**

so here in “onCreateServer( )” which is triggered when the button is clicked, here we set serverCreated equal to true, like that.

**$ onCreateServer( ){**

**$ this.serverCrated = true;**

**$ this.serverCreationStatus = ‘ Server was created! Name is ‘ + this.serverName;**

**$ }**

Now with this, we can go back to our template and simply bind ngIf to serverCreated

$ **<p \*ngIf=”serverCreated”> Server was created, server name is {{ serverName }} </p>**

and again this could also call a method which returns true or false or directly perform the check between the quotation marks, anything which returns true or false. Now with this if we save this, what you see is that there is no text but if I name this Testserver 2 and click the button, now the text is added here and the interesting thing is if I reload the app and we inspect our DOM here, you'll see here that's our app server component, above that is the button but if I click this button, a new element was entered here, the paragraph and here you see kind of the hook which Angular created to know where this should be entered.

But the important thing is it's really added or removed to or from the DOM, it's not there all the time, it's not hidden, it's just not there which is super important to understand. So that's ngIf, again the star at the beginning indicates that this is a structural directive, really changing the DOM as you just saw.

**#### 37. ENHANCING NGIF WITH ELSE CONDITION**

In the last lecture, we had a look at the basic ngIf syntax and that's the syntax you're going to use

most of the time, there is one alternative to it though.

Sometimes you have not only the if condition but also an else condition, so here we might want to say server was created, server name is something with ngIf but we could also add an else block where we then want to say “no server was created”. Now we deleted that exact text for the reason that we don't want to see it but imagine that we do want to show that alternative, so no server was created could be the text here and now we want to show this paragraph only if ngIf is not true or this condition, server created, is not true.

$ **<p \*ngIf=”serverCreated”> No server was created! </p>**

We can achieve this by placing a “local reference(#)” on this element **<ng-template>** here and local reference is something I will dive deeper into in the understanding components and data binding section of this course. For now let's think of it as a marker, so I will name this noserver, any name you like, noServer with a capital s and now we need to change the paragraph here to ng-template. That is a component, that directive shipping with Angular which you can use to mark places in the DOMand now I will add the paragraph back in this template with the text in between, like this.

**$ <ng-template #noServer>**

**$ <p>**

**$ No server was created!**

**$ </p>**

**$ </ng-template>**

So now basically the text we want to output and now with ng-template, with that local reference, this marker on it, we mark a certain spot in the template which we want to show conditionally. Now to show it conditionally, we simply enhance ngIf by also adding “**else**” and then noServer, so this marker we placed on ng-template and that's all the syntax we need. If we save this,

**<p \*ngIf=”serverCreated; else noServer”> Server was created, server name is {{ serverName }} </p>**

you see no server was created and as soon as I click here, that gets replaced with server was created. So that is ngIf else, a nice little addition in a use case where you have an else condition. Of course the alternative would always be to simply use ngIf with the reversed check, so with !server created but that here is another nice alternative. Now I'm going to delete it, you will find an article after the section here though where you can find this code again in case you are interested and we'll also use this code in the course project we're going to build. I just wanted to highlight that this option exists, using ngIf like this however is the case you will see most of. Now let's continue.

**#### 38. STYLING ELEMENTS DYNAMICALLY WITH NGSTYLE**

In the last lecture, we saw ngIf, super useful directive and ngIf was a structural directive, the other

type of directives are “**attribute directives**” which are called like this because ***they really just look like normal HTML attributes without a star basically***. Let's add one, on the server component here, we output the serverStatus right and the serverStatus always is offline, let's mix this up. Let's say we want to dynamically create this, so I'll add the constructor which is just a built-in method each TypeScript class has which is called once this component is created and here I will set the serverStatus to a random value, I will use the math.random function and this gives us a random number between 0 and 1, a floating point number and if this is greater than 0.5, I want to set the status to online otherwise I will set it to offline. So with that, we shouldn't get the same status all the time and now let's check this,

**$ constructor( ) {**

**$ this.serverStatus = Math.random( ) > 0.5 ? ‘online’ : ‘offline’;**

**$ }**

we see indeed online and offline, that's lucky, could have of course also been both offline since we have a 50% chance. Now we have two different statuses here and now let's say we want to change the color, the background color of this component depending on the serverStatus. For this, we can use another directive, on this wrapping paragraph here, I can add it and it's called ngStyle.

$ **< p ngStyle> {{ ‘Server’ }} with ID {{ serverId }} is {{ getServerStatus( ) }} </p>**

Now that's a built-in directive, you can recognize this on the ng at the beginning and ngStyle, that's the directive, is pretty useless like this, we need to give it some configuration to do something, that is why we will use property binding on this directive and it's super important to understand that the square brackets here are not part of the directive name, the directive name is just ngStyle,

$ **< p [ngStyle] > {{ ‘Server’ }} with ID {{ serverId }} is {{ getServerStatus( ) }} </p>**

the square brackets **[ngStyle]** indicate that we want to bind to some property on this directive and this property name happens to also be ngStyle. We will see this in practice on our own directives in the directives section later but super important to understand that property binding is not the same as a directive, it's totally different, we are binding to a property of the directive.

This ngStyle property expects to get a Javascript object and here, we define key-value pairs of the style name as the key and the value of the style as the value. So for example, we could bind background-color and if you want to use this notation with the dash, you have to wrap it in single quotation marks to make it a valid Javascript property name or you use the camel case notation, backgroundColor like this

$ **< p [ngStyle] = “{backgroundColor: ‘ red‘ }” >**

and then you could set this to red but I don't want to set it to red, I want to set it to something depending on the serverStatus. So here we could simply call a method, **getColor( ),** so that is really mixing now all the things and make it clear that between the quotation marks, we are executing TypeScript code here, so therefore we can of course call a method.

$ **< p [ngStyle] = “{backgroundColor: getColor( ) }” >**

Now getColor is a method we can add here and getColor should now let's say return green if the status is online and red otherwise.

**$ getColor( ){**

**$ return this.serverStatus === ‘online’ ? ‘green’ : ‘red’**

**$ }**

So I will return something, first I check the serverStatus, so this serverStatus and if this is equal to online, using a ternary expressing here, I will return green, otherwise I will return red. So getColor will return green or red and this is then assigned as the background color with ngStyle. If we now save this, let's have a look at the result.

We see indeed the running server has a green background and the offline server has a red one, so that is working, ngStyle allows us to dynamically update that. If I reload, you see now both are offline so both are red now. And that is the big advantage of ngStyle, it allows you to dynamically update the styles and of course if the serverStatus here did change over time, it would update the styling respectively. So this binding we set up here still is working, if the serverStatus here which is responsible for choosing the right color changes, it updates the style. So that's ngStyle, an attribute directive added like an attribute and in this case, also using property binding to configure it.

**#### 39. APPLYING CSS CLASSES DYNAMICALLY WITH ngClass**

We saw examples for ngStyle and ngIf, now something related to ngStyle is ngClass. Now we learn that ngStyle allows us to dynamically assign a style, now I will also add ngClass here and where ngStyle allowed us to change the CSS style itself, ngClass allows us to dynamically add or remove CSS classes. So let's create such a class and for this, I will first of all add my styles here and I will use back ticks to write this over multiple lines and I will give this a class name of online, in which case I want to “color : white”.

**$ @Component({**

**$ selector: ‘app-server’,**

**$ templateUrl: …………………**

**$ styles: [`**

**$ .online{**

**$ color: white;**

**$ }**

**$**  `]

$ })

So that's all I want to do because I think it's kind of hard to read the black text on the green background. So back here in the p tag, ngClass also, this is the directive, also only works as intended when using property binding. So let's wrap it in square brackets **[ngClass]** too and here we also need to pass a Javascript object and that's just the case for this property of this directive here, of course each property you bind to may take a different value, like disabled which took true or false.

So here it's a Javascript object and for ngClass object works like this. We also have key-value pairs, the keys are the CSS class names and the values are the conditions determining whether this class should be attached or not. So here,

$ **< p [ngStyle] = “{backgroundColor: getColor( ) }”**

**$ [ngClass]=”{online: serverSTatus === ‘online’ }”**

**$ >**

we have the online class, you could also wrap this in single quotation marks if you had a class name with a dash inside of it or something like this and here, well I want to attach this only if the server is online, so only if serverStatus equals online, online here is a string between single quotation marks. Now of course you could also outsource this in a method, you call this serverStatus equal online check but here I will write it inline.

So if this is online, we should attach the CSS class online to this paragraph otherwise this should not get attached. Let's view this in the running application, both are offline so none of the two should have the online CSS class added to it and indeed, I can't find the class on any of the two. Now let's reload and hope that some of them is offline or online, excuse me, both are so let's check it, now we see the online class has been added, actually to both of them because this is how ngClass works. It only adds like CSS class if a certain condition is true and that's an example for another built-in directive, another built-in attribute directive.

**#### 40. OUTPUTTING LIST WITH ngfor**

With all these directives learned, let's now have a look at the last built-in directive for now, an important one though. It's sad that we can click the add server button but we don't actually add servers to our list here right, this list doesn't grow, this list is totally static, we can change this with the “**ngFor directive**”, so let's have a look at how it works. In our servers component here, we right now manually add our app-server component twice, it would be nicer to have an array of servers which adds it dynamically.

**$ <app-server></app-server>**

**$ <app-server></app-server>**

So in the “servers.component” here, I'll add a new property, servers and this is an array and here, I could have my Testserver and my Testserver2 maybe and when we create a new server here( servers=[‘Testserver’, ‘Testserver2’] ), I actually want to access this array of servers and “**push**” a new server on it and this should of course use the server name I set up.

**$ OnCreateServer( ){**

**$ this……………….= true;**

**$ this.servers.push(this.serverName );**

**$ this.serverCreationStatus = ‘ Server………………………………………..;**

**$ }**

Now with this, I have an array of servers, it would now be great if we could replicate the app-server component as often as needed to have a server for each element in the array. So the two initially and then a new one for each server we add.

We can do this. I'll get rid of the second **<app-server></app-server>** here and on the first one, I will place another directive with a star because this is now also a structural directive, changing the DOM itself, is the ngFor directive. The ngFor base syntax looks like this, we define a temporary variable for inside the loop with let, give it any name you like, like server for example and then of servers. Servers here is the property we defined in the TypeScript file and this will now loop through all elements in this array and assign the individual element to this dynamic server variable. So that's just like you might know this loop from a normal Javascript code, the for/of loop.

$ <app-server \*ngFor="let server of servers"></app-server>

This server variable can now be used in the template but here, we don't really need it to be honest, we will soon learn in the next course section how we can pass data to our own components to output it there. So for now, we won't need that but one thing we should see is that if I save this and we go back to our application, we still see two because we start with two but if I add more servers, you see our list grows and we can click this as often as we want.

Of course the content of the individual server is still static because we can't pass the data like the server name to that component but that is something we will learn in the next course section when we dive deeper into components and we will learn how we can create our own properties on components we can then set from outside the feature which would be great to have here. So let's dive into this in the next course section, before doing so though, well let's practice the things we learned in the last lectures and then let's also set up our course project before we then dive deeper into components and data binding again. So see you there.

**#### 41 BASIC PROJECT ASSINGMENT THREE ( project name basics-assignment-three-mysolution )**

-Add a button which says “Display details

-Add a paragraph with any conent of your choice( i.e, Secret password = tuna”)

-Toggle displaying of that paragraph with button created in the first step

-log all button clicks in an array and output that array below the secret paragraph ( simply an incrementing number)

-starting at the 5th log, give all log tiems a blue background ( via ng style ) and white background

**## SOLUTION**

File name “basic-assignment-three-mysolution”

**#### 42 GETTING THE INDEX WHEN USING NGFOR**

In the last assignment, we created this solution with our log here where we could add items which get an

incrementing number and this number then determines whether we have a background color which is blue or this white text CSS class attached to it.

Now in the assignment, I said that it is one option to use an incrementing number in our log, another option might be to use any other content, for example a timestamp but it could be any other text. So I'll first of all change this here in my app.component.ts file and I will no longer push this number, instead I will use timestamp here which we can get with new date, date is a built-in object Javascript ships with,

so there is no need to add any import.

$ onToggleDetails( ){

$ **this.log.push(new Date( ) );**

$ }

Now with that, we're pushing new timestamps on this array, we can demonstrate this with ng serve running by simply clicking this button here but now everything is blue of course because our check here now always returns true, because the string and this is just a string as always, deemed as being greater than 5.

So our check doesn't work anymore, we can add something to ngFor here too. Separate with a semicolon,

you can extract some extra information, the current index or the index of the current iteration. With let, you can bind it to any variable name of your choice like i or index, whatever you like, I will set it to I and then you set equal to index. Now equal to index kind of is also like a reserved expression you could say, it gives you access to the index of the current iteration.

**$ let i = index**

So in the first loop, this will be zero, in the second loop, so in the second element in this array, i will

be one and so on. Now with this information, you can of course update your solution here to no longer check if logItem is greater or equal than 5 but if i is greater or equal than 4 because remember, the index starts at zero.

**$ [ngStyle]=””{backgroundColor: logItem >= 5 ? ‘blue’ : ‘transparent’**

**$ //replace the above with**

**$ [ngStyle]=””{backgroundColor: i >=4 ? ‘blue’ : ‘transparent’**

So let's update both checks like this:

**$ [ngClass]=”{‘white-text’: i >= 4}”**

and now with that, this should work with any content and not just with incrementing numbers, this looks pretty good. So that's a tiny addition, extracting the index of ngFor just like that. With that, enough about the basics, we got a lot of solid knowledge right now, let's use it to set up our course project in the next section before we then dive deeper into components and data binding.

Can't wait to see you there.

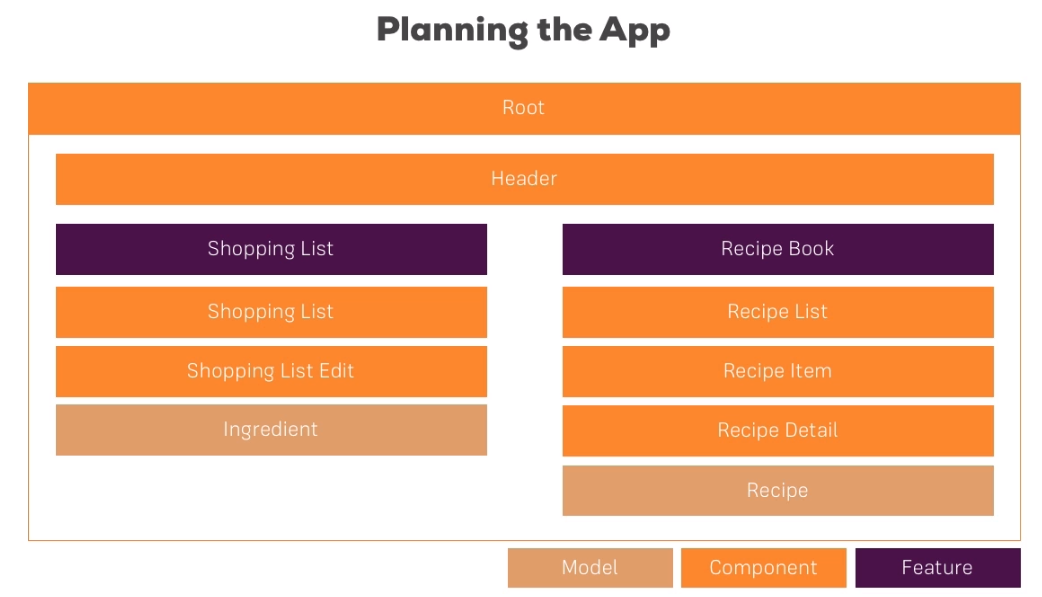
**SECTION 3: COURSE PROJECT – THE BASICS**

**#### PLANNING THE APP**

Time to practice the things we learned thus far in our course project. Now what is our course project?

My goal in this project which we will build throughout the course, to which we will come back after the individual sections of this course, is to build a recipe book and shopping list app, you saw it in the intro video of the course. Basically there, we're going to have two sections, shopping list and recipe book and we will be able to manage our recipes, view them in detail and also to manage our shopping list and even push ingredients from a recipe directly to the shopping list. This application will use a lot of the features we learned throughout the course and therefore I think it's perfect to practice these things and see them come together in a real app. First of all though, let's try to plan the application because before creating the app, you of course need to have a plan and therefore in the next lecture, we're going to think about which components we will need for the different features of this application.

**#### 44. PLANNING THE APP**



The first step in creating an Angular application of course is to lay out the structure of this app and especially plan which components you're probably going to need. Now it's normal that whilst building the app, you decide to add another component which you didn't plan to add at the beginning or maybe you decide that you can merge two components you thought you would have to split but generally, this is still a good approach and this should also show you how I approach this and how you may decide what should go into its own component and what can be merged into another one.

So on this page, I try to lay out the general structure of our app and at the bottom, you can find kind of a legend, so let's start with the features of this app. Well, we're going to have a “**shopping list**” and a “**recipe book**” section as I said, managing either our single ingredients we need to buy or in the recipe book, our whole recipes. Which components will we need for this? Obviously, a “**root component**” holding our overall application, the “**app component**”, every Angular app has this.

Then since we have two sections, it probably makes sense to have some kind of “**header component”** where we can navigate between these two sections. Of course you could also hardcode the headers simply into the app component but since it will contain its own business logic, in the end it will trigger a routing action later and we will also add a dropdown there in the end which allows us to store our recipes on the server and fetch them from there.

So since we will have that logic attached to the header, outsourcing it into its own component makes sense so that we don't have to put all the logic into our root component which really mainly should only be responsible for holding our overall app.

So that is the header, now let's dive into the individual features, let's start with the “**shopping list**”. Here I guess it makes sense to have the overall “**shopping-list component**”, remember the purple box is only the feature so we need a component holding our shopping list and maybe in this shopping-list component,

we also want to have a “**shopping list edit**” part which allows us to add new items, so an input field and a button because that again has its own logic, we will have to decide if we are editing an existing item or adding a new one, we need to submit this, so it makes sense to create its own component for this.

So if we have an overall list component, mainly holding the data of the list and the edit component being responsible for managing this data kind of, though the edit component will be nested inside the list component. I guess that's it for the shopping list, let's have a look at the recipe book.

Here, it certainly makes sense to have, again just like with the shopping list, a “**recipe-list component**“ which shows us a list of all our recipes. Here we might later also put each individual recipe into its “**own item**” though since it holds a little bit more information than just one line of HTML code and a “**recipe-detail”** area would be great too.

So an area where once we select a recipe, we can see information about that. Later in the course, we will also add another component here because we will then somehow need a component which allows us to edit existing recipes or add new ones but since this is really advanced and we would not be able to fill it with much life right now, I will omit it for now and focus on the displaying part.

So a list with items and then the details section which displays the details for the currently selected item. I guess this looks like a good plan, each of these components should hold a significant amount of business logic and I guess with the component sketch out here, we're really having each component focus on one main topic, displaying a list, displaying information about a single item, displaying detail information about a selected item and so on.

So this is generally how I approach this, of course feel free to split this up even more or if you want, merge them together, for example recipe-list in a single item but this will become more clear once we actually implement these components. Now one last thing we should think about, which models will be use in this app? And with “**model**”, I simply mean which “**data**”.

So we certainly will need some representation for our ingredient. We will use ingredients a lot in the recipe book too, each recipe there has a couple of ingredients, so we should define how the ingredient looks like. That is something which your application probably also needs, be clear about the data you're going to use and put it into its own class, so that you have your own type you can use later on and that you have a clear interface or definition of what your data looks like so that you can easily have your components talk with each other, something we'll dive in later in this course so that there is no issue regarding the data we're exchanging. So besides the ingredients which probably is a very simple model, only controlling the name and amount maybe, we also will need a model for the “**recipe**” which contains things like “**title**”, the “**description**”, the “**ingredients**” and so on. And I think that with this, we have a solid plan for now, let's get started, let's implement all these components, fill it with some dummy data for now and take the first step in building our course project.

**#### 45. INSTALLING BOOSTRAP CORRECTLY**

Download bootstrap locally in that project by typing

$ **npm install –save bootstrap@3**

Once it is installed you need to make the changes in the “angular.json” file look under the build you will see script array.

In that array add “**node\_modules/”** right above **src/styles.css.** This node modules is available if the “node modules folder, scroll down to bootstrap / dist folder, css folder, bootstrap.css folder and bootstrap.min.css . so you will add “**nodule\_modules/bootstrap/dist/css/bootstrap.min.css**”

Solution for the above assignment is under the folder “basics-assignment-mysolution”

In this course, we use version 3 of the framework, install it via “npm install - - save bootstrap@3”

🡺3 is important

Additionally, when using a project created with Angular CLI 6+ ( check via **ng –v** ), you’ll have an “**angular.json**” file instead of “.**angular-cli.json**” file. In that file, you still need to add Bootstrap to the “styles[ ]” array as shown in the next video but path should be

$ **node\_modules/bootstrap/dist/css/bootstrap.min.css**

Do not use . . ./ node\_modules

**#### 46. SETTING UP THE APPLICATION**

I'm in a brand new project created with the CLI and the only thing I did thus far is I cleaned up the app component, remove this property we had there and cleaned up the app.component.html file. All the rest is still the same, I didn't change anything else, so the first thing I want to do and you can of course recreate this project by simply “**running ng new**”, any project name you like and then cleaning it up like I did. The first thing I want to do is I want to add the Bootstrap CSS framework so that we have some CSS classes we can work with to give this some nice styling because in this app, I really want to focus on Angular and not on the styling.

So of course, we can add Bootstrap with npm inside this project folder, in the terminal, I type npm install

then --save as it will be a production dependency Bootstrap and this will automatically install the latest version of Bootstrap in this project. Now once it is finished, we get Bootstrap installed, now we also need to inform the CLI that Bootstrap should be included in our final bundle it creates for us because of course, the CLI bundles all our script files and style files and by default, it would not include Bootstrap.

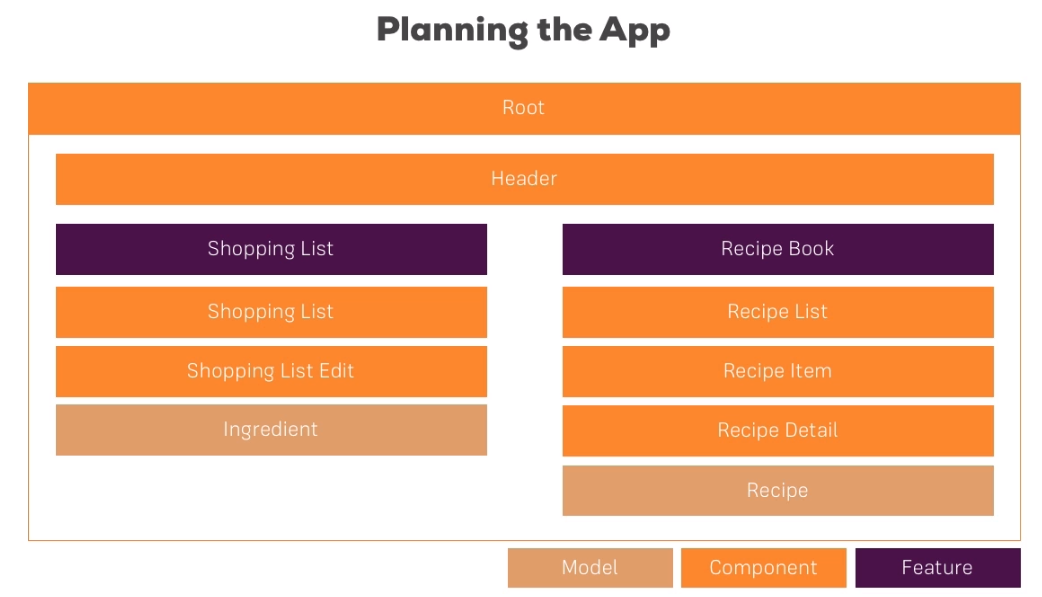
So to inform the CLI, we should go to the “.**angular-cli.json**” file and here in styles, in this styles array, we can add any global stylesheets we want to add to our whole project. Now it already has the styles.css file which is a file where you can find styles for the whole project, here I will simply add the path to our Bootstrap styles. Now we can have a look where that is stored. In a **node modules folder** which manages all our dependencies, if we have a look at the **Bootstrap folder** there, the “**dist folder**” sounds about right for distribution, there we have a “**CSS folder**” and here the **bootstrap.css** file is probably what we need. So we can quickly import this here by adding a path to that and now the important thing is we need to go up one level first because you don't have to see this path relative from the cli.json file here, so this config file we're in but you have to see it relative from your index.html file and that is in the source folder, so it's nested one folder deep into your app you could say.

So here we target node modules and then we can target the Bootstrap folder and now again just to make this really clear where this is coming from, in the Bootstrap folder, we can target dist/css so let's do this, dist/css and here, bootstrap.css, just like that. To be precise, it would even be better to target the minified version which is a file also lying in that folder, so let's add .min between bootstrap and .css, so this will already give us the minified code which is of course smaller.

Now with that, this should be added and now we can run ng serve to get our CLI to build this project for the first time and serve it at localhost:4200 as we did before in this course and that is our running application. Now we don't see very much here, of course because our app.component.html file is completely empty, we can quickly check if Bootstrap styles were added successfully by adding a div for the class container and the shortcut I'm using here is a plugin called “**emmet**” which allows me to just type dot class name and “**hit tab to autocomplete it**”.

It's available for many IDEs, so I'll simply Google for plus your IDE or editor and chances are you will find it and then in there I want to nest a row and now let's say simply to get started, a column for medium sized devices which spans the full width for now and I simply want to output in a h2 tab I'm working. So with this, if we save that m it should recompile and now we're seeing I'm working, this looks like Bootstrap is working too, we can see those on the font size and also if we inspect this here in the styles, all these styles here and down there are coming from Bootstrap. So this is working and with that, we get our application set up. Now let's get started creating all the components we require.

**#### 47. CREATING THE COMPONENTS**



So in the last lecture, we set up our project and we add bootstrap to it, now I want to get started adding the components we laid out here. Here's your challenge, try doing this on your own, now with that I don't mean filling these component templates with life, for now each component may simply hold some dummy text so that we can see that it's there, I mean create the components and try to create a clever folder structure in your app folder. Not every component should have a folder on the root level, you should also nest them by feature maybe and try to create at least one component manually by hand, not using the CLI.

I'll give you a chance to pause the video and then I will show you my approach, chances are that we deviate in our solution but that's a good practice for you, so definitely go ahead and try creating these components without any content for in it for now on your own.

So were you successful?

Let's see how I would solve this, how I would approach this. We got two main areas, recipes in the shopping-list and we get this ***header component***. Now I will “**create the header component**” manually. So for this, we can already start the discussions, should we create a header folder in the app folder where we store this component or should we create a component in the app folder itself? Now I will create a header folder but placing the component directly in the app folder would also not be wrong since it is really only used by our app component.

So here, I will create this new folder named header and in there, I'll add my “**header.component.ts**” file. Now as you learned, a component is simply just a TypeScript class, so let's name it HeaderComponent here (inside the header.component.ts )and right now, that would not be recognized as a component by

Angular, instead here we have to add the **@Component( )**decorator and later, I'm going to use the auto import feature of my IDE.

$ **export class HeaderComponent{…………………………………}**

For now I'll import it manually, we have to import component between curly braces from @angular/core, that's important.

$ **import { Component } from ‘@angular/core’;**

Now with that, we need to pass a Javascript object to this decorator to “**configure this component**”, specifically we need to add a “t**emplate**” and I'll put this in external file, so “**templateUrl**” in this case and a “**selector**” so that we can use this component. So the selector here is “**app-header**” because I want to ensure that I have a unique selector and that I don't overwrite an existing HTML element for example and header would be an existing element.

The templateUrl, here I want to point to the “/**header.component.html**” file and as you've probably recognized, this file doesn't exist. So I can quickly create this by creating a new file in the same folder, in the header folder, here **header.component.html is the full file** name and in this HTML file here, I simply want to output **“<h1>the header</h1>”** for now, I'm going to add some content soon.

So that's my header component and I can already **include this in my app.component.html,** maybe above the container because in the container, I later only want to contain my content component I will say, so here I'll “**add the <app-header>”,** just like this. So with that, we added it here, now let's see if we can see it and I just see “**loading...”** which always is an indicator that there has been an error.

So check the Javascript console in your developer tools to see the error and the error is pretty clear, app-header is not a known element. This is a common gotcha which is why I wanted to show it.

Why is it not known?

We added it here and we set up selector here, right?

Well remember, you have to register all the features you're going to use in the app module and since we didn't use the CLI to create this component, it wasn't added there automatically. So we have to add it manually to our “**declarations array**” and that means we also have to **add the import**. So we should import header component from and now pointing to the header folder and then in there, the ***header.component*** file without the file extension. So that now unlocks the header component in our app and if we save this, now we see the header. Now that's not pretty, we're going to change this soon but we can see that it is working.

Now, let's move on to adding all the other components with the CLI for now. So we had a couple of components we wanted to add and I will use the ng generate command or just ng g as a shortcut and then we want to generate a component or just c as a shortcut and the first component I want to **add is the recipes component** for this recipes feature area.

I'll add spec false to prevent the creation of a testing file which I don't need. So with that, we get a new folder in our app folder, the **recipes folder** which holds our recipes component, that is great. Now this will be our overall recipes feature component and that actually is a component I didn't sketch out here, here we only have recipe-list item in detail. Now you could also just work with these but I want to create a set up where I have a list on the left let's say and if I click an item, the detail component is displayed on the right, so I will need an overall component holding both.

Now that is an optional set up, you can also go with another one where the detail is somehow embedded into the list component, I chose to create this overall component. Now besides that, I will go ahead and create the other components though, so besides recipes, I will also now **create the recipe-list component** as laid out on the slides. Now here, I don't want to create the recipe-list on the or inside the app folder though, which is the default behavior by the CLI if I now would hit enter, instead that should go inside the recipes folder because that is where all the recipe related components should go.

We can easily tell the CLI to create this component in a subfolder by basically passing a path here, so instead of just saying recipe-list, we can say **ng g c recipes/recipe-list** and since you already have the recipes folder, this **will create the recipes-list folder** inside of the recipes folder, that is what I meant with structuring the folders by feature. Now with the recipe-list added, I also want to **add the recipe-detail** here because the recipe detail should be displayed next to the list in my set up here, so I will add it to the recipes folder too, so that now we also have the recipe detail folder here and we also need the component for a single recipe-item, right, ***so recipe-item***.

However, I don't want to add that to the recipes folder, that should go inside the recipe-list folder because it's in the recipe-list where I want to also have that item, so I'll point to the recipe-list subfolder here and now hit enter. “**ng g c /recipes/recipe-list/recipe-item**” And now we get the component inside the recipe-list because again the recipe-item is just one item in this list of recipes. Now again, it's only one possible folder structure, you could remove the recipes component and put the recipe-list component as the main recipe feature component and add details somewhere in this component then, I chose to have this set up here though.

Now let's move on to the shopping-list. Well for this, I'll of course run ng g c again to generate a new component and I **name it shopping-list**, I'll also add spec false here to prevent the creation of this testing file and this will give me a shopping-list folder in my main app folder, so next to recipes and header. Inside the shopping-list, there's only one other component I want to create and that's the **shopping-list-edit** component which will allow us to add new ingredients or edit existing ones, that should now go into the shopping-list component, so just like before with recipes, I'll simply change this to **ng g c shopping-list/shopping-edit** and this will now add it inside of this component.

So now we get all the components we need for now or at least I think we need for now, of course feel free to choose a different set up but with all these components added and since you use the CLI, they were also added automatically to app module. If you did create them manually, make sure to add them here too. So with all that created and added to app module, we can now use them. So let's start using them in the next lecture and let's also start filling them with some life there.

**#### 48. USING THE COMPONENTS YOU HAVE CREATED**

We created all these components, let's start using them. I will start using them in the app.component.html file, here where I say I'm working, here inside app.component.html I want to display my **<app-recipes>** component, so the one we created here in the recipes folder and below it, the **<app-shopping-list>** component, so this component here. If we save this, we see “**recipes works!”** and “**shopping list works**!”, which of course are the default strings the CLI gave us here.

So with these added and working here, let's now also work on the individual components here, starting with recipes maybe. In my **recipes.component.html**, as I said in my set up, I want to have a **recipes list** and **detail next to each other.** So for this, I'll add a **row div** here and in this row, I'll add a div with the class **col-md-5** to have a smaller column on the left and then another one with **col-md-7**, using these Bootstrap columns sizing classes to have a broader column on the right, so two columns next to each other.

Now in the first column, I want to have my **<app-recipe-list>** component, in the second one I want to have my **<app-recipe-detail>** component, just like that. So that should place these two components next to each other, if we save this and have a look at our app, we indeed see list and detail below each other simply because due to the console, the viewport was so small and then it automatically wraps them below each other. So if you have a normal page, they sit next to each other and I actually want them to wrap below each other on smaller pages.

So this is now working too. Now in my **recipe-list** component here, as I said, I somehow want to include that **recipe-item**. So here I will later output a list of items, for now I will simply include the **<app-recipe-item>** here, app recipe item, so only the selector, which should now ensure that we no longer see recipe-list works but recipe-item works because that's the only content of our recipe-list component for now. So that's looking great, now for the shopping list, I'll do pretty much the same. In my **shopping-list component.html,** here I want to display the list and above it this shopping edit area, so I'll add a **row** here too and in this row, so let's say with **col-xs-10** to create a very wide column here, I'll first add my **<app-shopping-edit>** component to have this editing area at the top where we can add new ingredients or edit existing ones and then I want to have a horizontal line here and somehow output my list below this. ( **<p> The list </p>** )

So that's the rough setup I want to have in this component. So it should work like this, that will be the recipe-list component, later on we're going to fill this with more life but for now, I'm very happy with how that looks. Talking about how it looks, the header really kind of destroys our whole layout here. So let's work on the header in the next lecture and let's actually bring it into its almost final form already.

**#### 49. ADDING NAVIGATION BAR**

So we added all the components we created to our templates, now let's fine tune them and add the real content we want to have there, starting in the header. Of course I don't want to have this h1 tag here, instead I will use a couple of Bootstrap classes to create a nice looking header. So I'll create a wrapping **nav** element and on this element, I'll add the **navbar class** and the **navbar-default.** Again, these are just Bootstrap classes which will give me this default Bootstrap header.

In there, I'll place a **container-fluid div** to have my own view container basically in there and the first thing

I want to have here is a header. For that, we get the **navbar header class** we can add to a div and in this div, I want to have a clickable header, so I'll add an **anchor tag** pointing to hash for now, we'll later of course add a route there once we learned how to do this and this should have the class **navbar-brand**, to give this this brand-like text style, I'll say **Recipe book here** since this is the name of our application. So that's the header part of our header, so this main brand part, now let's work on the actual links. For this,

I'll add a div which will receive the **collapse class** and then the **navbar-collapse class.** Now, I won't really use the collapse feature, I won't add a hamburger icon here since we will not have many links on our header anyways but this gives us some nice styles and of course, collapse here should have an e at the end, like that.

So in here, I now want to have a group of list items, so a group of links in the end, this will be an **unordered**

**list** with the CSS classes, **nav** and then **navbar-nav**. So make sure to create this unordered list with these classes and in there, if we add some list items, they will actually sit next to each other, so they will automatically be positioned correctly by Bootstrap. Each list item should hold a link, so here I'll add an **anchor tag**, also pointing to just **hash** for now, one for the **recipes** section and the second one for the **shopping list** section.

Now with that added, I'll add another area in my header here, this will be **another unordered list** which will receive the **classes, nav** and now also **navbar-nav** like before but then also **navbar-right** to position these links at the right and actually, this will only be one link here which will receive a **class of dropdown**, this list item will, because later, we will add our own directive which will turn this into a working dropdown where we will then choose some options to store our data on a server, something we'll also implement later in this course or fetch it from there.

So in the dropdown, I simply want to have my dropdown menu which when using Bootstrap is added by adding an **unordered list with the class dropdown-menu,** so this will hold the items this dropdown and here, I'll add a couple of list items which all hold some links pointing to just hash for now, one for **saving data** and one for **fetching data.** Again this will be filled with working code later.

So that's my header, let's save it and see how it looks. Looks ok, I don't see my dropdown though, the brand looks off, so I'm not entirely happy with that. Let's see what is wrong. The brand is not working because that should be navbar, not have a h in between, so if you fix this, the brand looks all right. Now what's wrong with the dropdown? Well the dropdown simply misses the link we would see to open the dropdown because this unordered class here simply is the content of the dropdown, we of course also need a link to trigger it. So let's add an anchor tag pointing to hashtag here where I will say **manage,** just manage maybe, you could of course change that text and this should have a **class of dropdown-toggle** because this will be the button toggling the dropdown in the end. Therefore it will have a **row of button** to add some accessibility to this and also required for this to work correctly though and with this now if we save this, we see the dropdown. It would be nice to have a little arrow next to it indicating that it is a dropdown.

So next to manage, I'll add a **span** with a **class of caret** which will give us this arrow pointing to the bottom,

this arrow here and now we get a dropdown. It won't work if we click it because we haven't added any code which would open the dropdown, we will do this after the directives section though. So the header is now working, with that let's move on and let's work on our recipes section.

**#### 50. ALTERNATIVE NONCOLLAPSABILE NAVIGATION BAR**

The way we added it, the Navbar will collapse on smaller screens. Since we didn’t implement a Hamburger menu, that means that there’s no way of accessing our links on smaller screens.

You can either add such a menu on your own ( see below ) or replace “collapse navbar-collapse” with just “navbar-default”

Adding a Hamburger Menu:

Alternatively, if you want to make the navigation bar responsive, please replace these lines in

**$ <div class=”navbar-header”>**

**$ <a routerLInk=”/” class=”navbar-brand”> Recipe Book</a>**

**$ </div>**

**$ <div class=”collapse navbar-collapse”>**

With these lines

**$ <div class=”navbar-header”>**

**$ <button type=”button” class=”navbar-toggle” (click)=”collapsed =!collapse”>**

**$ <span class=”icon-bar” \*ngFor=”let iconBar of [1,2,3]”</span>**

**$ </button>**

**$ <a routerLInk=”/” class=”navbar-brand”> Recipe Book</a>**

**$ </div>**

**$ <div class=”navbar-collapse” [class.collapse]=”collapsed” (window:resize)=”collapsed=true”>**

And add this line to header.component.ts

**$ collapsed = true;**

**#### 51. CREATING A RECIPE MODEL**

We added the header in the last lecture, now I want to work on my recipes. So we get the recipes component which holds the recipe-list, now I also want to fill the list with some life. For this, I'll go into my recipe-list component and we'll learn more about ngOnInit later in the next module, in the components module, for now what I want to do here is I want to add my recipes array, my array of recipes which for now is an empty array and here

recipes: Recipe[]

, let's go back to that slide from the beginning, I want to define how a recipe should look like.

We're going to use a recipe a lot throughout this app and therefore we should definitely clearly define how a recipe looks like, so that whenever we use it in any component, we're always talking about the same structure, about the same kind of object.

So for this, I will create a “**model**”. **What is a model then?** A model simply is a TypeScript file. So in the recipes folder because it's going to be the recipe model, I'll add another file, it's called **recipe.model.ts**. Now .model is optional but again you should be descriptive about what's inside the file and with that, it's pretty clear what will be inside.

So I'll add this file, again it's now **just in the recipes folder** and how should this file look like? First of all, I'm going to **export a ts class**, a TypeScript class which is simply **named Recipe**

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

because we define how a single recipe looks like here. Now you could think that we're going to add something like @model here, now we're not going to do this, there is no decorator like this and we don't need to do this because we can use vanilla TypeScript for this. A model in the end should just be a blueprint for objects we create and the TypeScript class does just this job.

A class can be instantiated, so we can create new objects based on the setup we provide here in this class. So we can define how a recipe should look like in this class, so let's do this. A recipe should have a name and I'll add an accessor to be really clear about that this is **publicly available**, so that this can be accessed from outside if using this as an instantiated object. So I'll **add public** in front of it and then **name** as the **property name**.

I'll also assign the type of this by adding a colon and the **type will be string** and that is just how you assign types in TypeScript, you add a colon after the property name and then the type you want to assign, so string in this case. I also want to have a description in each recipe, so I'll **add my description property which also is a string because a description is just a text of course**.

$ export class Recipe {

    public name: string;

    public description: string;

Now maybe we also want to store an **image for each recipe**, so we should have an **imagePath** since we won't store the image itself here of course, that wouldn't work, we can't store files in our code but we want to store the path pointing to the image and we will simply use images from the web here, so this will hold a URL in the end, so that is also a string, a text in the end. That is the basic model for now, the basic blueprint. I'll also **add a constructor to it** so that we can instantiate this with the new keyword and pass the arguments right to the constructor, so here I **expect to receive the name** which will be of **type string**,

the **description ( desc** ), I'll just write it a bit shorter here which will be of **type string** and the **imagePath** which will be of **type string**. In the constructor body and if that's brand new to you, the constructor is simply a built-in function every class has and which will be executed once you create a new instance of this class, I will show you how this works in a second.

$ constructor(name: string, desc: string, imagePath: string){

So inside the body of this constructor, we have to assign the arguments we receive here to the properties of our object now, to the properties of our class. So **this.name,** referring to the **name up here**, to our property **should be equal to name** and now the same for **description, this.description** **should be equal to** **desc** and **this.imagePath should be equal to imagePath**. So with that, we get a recipe model we can use, now let's use it in the next lecture.

constructor(name: string, desc: string, imagePath: string){

        this.name = name;

        this.description = desc;

        this.imagePath = imagePath;

    }

**#### 52. ADDING CONTENT TO RECIPES COMPONENTS**

We added our recipe model, now let's use it. So back to the **recipe-list.component**, < here I added my recipes array ( **recipes = [ ]** ) and now that we created the model, we can also define a type for this. So we can add colon to inform TypeScript that now we will define the type for this property and the title will be **Recipe,** so that will use our own model and not just a recipe but an array of recipes which is indicated by adding square brackets after it.

$ **recipes : Recipe[ ] = [ ]**

So now TypeScript knows the only thing which will ever get stored in this recipes property is an array which holds a couple of recipe object or also no such objects, that would be a possibility too but certainly not a couple of strings or numbers but objects like this, like you find in our model. Now **to inform TypeScript where this type is coming from, we need to add an import**. So I'll import Recipe, that is what we named it in here, right, that's our class name so that's what we have to specify between the curly braces here, from and now we have to move up one folder.

Since we're in the recipe-list folder right now, we have to go to the recipes folder because that is where our recipe model lives, so now I navigated up one folder just like with the cd command in your terminal and then it's in the recipe model file, again without the file extension. So with that, we can now use this model and now we can manage our array of recipes here and let's start with one recipe, one dummy recipe.

$ import { Recipe } from '../recipe.model';

So in this array, I will **create a new recipe** and I can use the **new** keyword to create a new object based

on this recipe blueprint, based on our class and now when executing this like a method, we're actually calling the constructor, so now here we need to pass the arguments we're expecting in this constructor, **so name, description and image path**.

Now you may choose any data you want here, I'll name this **a test recipe**, so that will be the title, the name of the recipe, then description, **this is simply a test** maybe and the **imagePath**, now as I said, I'm going to use images from the web here, so we're going to use a URL. Now let's simply search for an image and I will search for recipe, let's use google image search, I also want to make sure that I may use whatever I find here. So let's find a recipe, maybe this one here, let's see. If we take this and deep linking like of course shall be done in a real app, this is just for testing this.

So if we take this image here and paste the path in here as a string, now we should be able to use this later on once we output this. So with that, we are using our own model, we're using it here in the recipe-list component, of course we won't be able to see anything for now.

recipes: Recipe[] = [

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

so to see that, we should actually do something in the template of the recipe-list component. Here I'm right now only outputting my **<app-recipe-item>** and I want to loop through all these items and use them to display my recipes. For now, what I will do is I will add a Bootstrap row first of all, like that and in that row, I want to ***add a column*** with the class **col-xs-12** to have it span over the full width and here I will later **add a button** or I will add the button do right now but I will later add the functionality to create a new recipe.

So here, I'll simply set up a **button** with the classes **btn** and then **btn-success** where I say **new recipe**, like this. Now again, this will be hooked up later. So I'll add another row below this which should now actually hold my list of recipes, here I'll again use **col-xs-12** to have this span over the full width and keep in mind, this will already be nested inside of this md-5 columns, so it's only that wide to begin with but inside this column, I wanted to span over the full width and now here, we will later output my recipe items.

<div class="row">

    <div class="col-xs-12">

        <button class="btn btn-success">New Recipe</button>

    </div>

</div>

<div class="row">

    <div class="col-xs-12">

<app-recipe-item></app-recipe-item>

</div>

</div>

For now though let's put the code we will later put into **app-recipe-item** into this component here directly because right now, we will have no way of getting our recipe data down to the recipe-item. So that's something I will actually do after the next section once we learned this but so that we're able to see something, I will add the code for a single recipe-item here into the recipe-list directly. A single recipe-item **should be clickable**, so it's going to be an **anchor tag** because later once we click it, we want to select it and here, I will then actually assign another class which will be a **list-group-item** to make it look like a list-group-item in Bootstrap, also **add clearFix**, otherwise it will probably look wrong and then I'll add a div which will receive the class **pull-left** since I want to have this on the left of this list item. In there, a h4 tag maybe with the **recipe name**, this should receive the class **list-group-item-heading**, again these are all Bootstrap classes I'm using here and there, I will later output the name of the recipe.

so recipe name for now, we will soon replace this with dynamic content. Below this, I want to have a paragraph with a class of **list-group-item-text** and here, we will have a **description** of the recipe and on the other side, I'll add a span with a class of **pull-right**, in this span, I want to **output my image**. So here, what I'll do is I'll add a class, **img-responsive** so that Bootstrap will resize to image automatically for me

and again all that content will go into the app-recipe-item component after the next course section.

**$ <a href=”# class”list-group-item clearfix”>**

**$ <div class=”pull-left”>**

**$ <h4 class=”list-group-item-heading”>Recipe name</h4>**

**$ <p class=”list-group-item-text>Description</p>**

**$ </div>**

**$ <span class=”pull-right”>**

**$ <img src=” “ alt” “ class=”img-responsive” style=”max-height: 50px”>**

**$ </span>**

**$ </a>**

One more thing I want to set on the image though, I want to add my own style, I'll simply add an inline style here to set the **maximum height of that item to 50 pixels**, so that we can't go above that and Bootstrap will handle the rest and resize the image appropriately. So that's my set up for a single item,

of course feel free to fine tune this to your needs. Now with that, the goal is to replicate this item, again we will later put it into its own component and then fill these spaces ( **Recipe Name, Description and src** ) here with dynamic data. This is your challenge, we got a recipe-list component with our recipes array, the array only holds one item for now, you of course are free to add more but even with that one item, you should be able to replicate this part here as often as needed to create one section of this code for each item in this recipe array and then **fill name, description and the source attribute here, property, with the right values for this recipe.**

I will do this in the next lecture but definitely try doing this on your own first.

**#### 53. OUTPUTTING A LIST RECIPES WITH NGFOR**

So were you successful using our recipes array here and replicating this part in the code as often as needed and fill it with the data for the recipe?

Now clearly, we need ***ngFor*** for this.

So let’s add ***ngFor(inside the anchor tag that holds recipe name and description)*** , this is the directive which allows us to repeat code in our template and here, I want to loop through all the recipes.

So I will actually create a local **variable named recipe**, this name is totally up to you, that will simply store the recipe of each iteration in our loop and I will **loop through all the recipes**, so through this recipes property we created here in the TypeScript file.

**$ <a href=”# class”list-group-item clearfix” \*ngFor=”let recipe of recipes”>**

Now with that, I’ll also bring this into new lines to make it a bit easier to read(break the anchor tag). With that, everywhere inside this ngFor loop, so everywhere here, we can use this recipe now.

So let’s use it to output the name(replace Recipe Name h4 tag with), we can use string interpolation for that.

String interpolation as we learned allows us to output some text anywhere in our template. Now we got the single recipe but that’s going to be an object looking like our model because remember, the recipes array( **recipes = [ ] = Recipe[ ]** ) simply holds an array of our recipes, so a single item simply looks like this**({{recipe.name}}**) , it has a name description and then imagePath. Well with this information, we can of course access the name here, like this and for the description(replace Description with **({{recipe.description}})** we can replace this with description. So we’re going to use the property names we set up here in the model(**recipe.model.ts**) and important you have to use the ones here at the top of course

    public name: string;

    public description: string;

    public imagePath: string;

because down there (the constructor) these are only the constructor arguments names they are not available outside of this model.

So that’s the name and description, now about that image. First of all for that alt-text( inside the image tag, **alt=”{{ }}”** )here that’s a normal HTML attribute of course. I will use string interpolation to output some text here, the **recipe.name** maybe but regarding the image source itself and im splitting this over multiple lines to make it a bit easier to read regarding that source we can follow two approaches.

You can either use string interpolation to output recipe imagePath ( **src=”{{recipe.imagePath }}** ) and that should work fine or as an alternative you cannot use the attribute plus string interpolation but you could bind ( **[src]=”recipe.imagePath”** ) to the **source property itself** and now here you don’t add curly braces instead just recipe imagePath. Because now we’re using property binding and not string interpolation anymore.

So here is a great example of where you may use both string interpolation with the normal HTML attribute or not using that tribute but directly binding to the source property (src=”” inside img tag) of the image and then just putting in a valid expression returning what this property expects in this case string between the quotation marks.

Now I will go with the property binding approach here and since I cant comment inside the image tag, this out in here without breaking the element im going to remove it but I want to keep in mind that both approaches would work. (remove **src=”{{recipe.imagePath}}”** )

With that lets save and see if we see something and actually that looks pretty good but looks like the image I chose does not support deep linking so that’s nice probably good for them let me quickly pick a different image then. So I ended up with picking this image here so you can see it here and now we get the list working. This is the style I want to have of course it doesn’t do anything if I click it but that’s something we can add in the future. For now the recipe-list component is taking shape, I want to add one more thing though I don’t want to have this button sit directly on top of this list.

So what ill quickly do in my recipe-list component here between the button and the list I can simple add a horizontal line to have a little bit of a spacing in-between and this nice separation. So that’s the list and you should see that if you add more items (inside the recipe-list.component.ts file copy the new Recipe and paste it again ) to that list that it automatically grows, so if I simply duplicate this because im lazy you should see the same recipe twice here.

So that’s working great and now lets work on the details section next.

**#### 54. DISPLAYING RECIPE DETAILS**

In the last lecture, we finished this recipe-list component, now let's work on the recipe detail. Now for the recipe detail here(**recipe-detail component**), thereI want to output the information about the selected recipe. So in here, in this HTML file(**recipe detail html**), I'll add another Bootstrap row and then a column spanning over the full width( col-xs-12) and keep in mind the detail will be added to our component here, in the recipes components(**recipes.component.ts**), so it will be in this column on the right to the list but in there, I want to have a column spanning the full available width and I want to start by displaying the image.

So let's add the image here(**in recipe-detail html**) and let's **add the img-responsive class to it**. Now I will fill this with life and output some data there later. Below the image, I'll have another row, in this row also have a column spanning the full width and here maybe output the recipe name, so **Reecipe name** would be nice to see here. Then maybe another row with another column in there spending the full width again and here what I want to add is, I want to have buttons which allow me to manage this recipe.

So a button which allows us to add it to the shopping-list and then one which allows us to edit it or to delete it and actually, we could use a dropdown here. Such dropdown may be added here by **adding a div**

with the **Bootstrap class btn-group** and again this is just how to create a nice looking dropdown using the bootstraps CSS framework which is the framework we use in this project.

Here, we need a button which will **open the dropdown, this will be of type button** and it will receive a **class of button**, then I'll use the **button primary** look which is this bluish button and very important, it will also receive the class **dropdown toggle** because it will act as such. On this button, I want to say manage **recipe maybe** because we want to allow the user to manage the single recipe and I want to add this span with the **class caret** to have this arrow pointing to the button again and I will distribute this over multiple lines so that it's really easy to see what's going on here.

**$ <div class=”row”>**

**$ <div class=”col-xs-12”>**

**$ <jimg src=” “ alt= “ “ class=”img-responsive”>**

**$ </div>**

**$ </div>**

**$ <div class=”row”>**

**$ <div class=”col-xs-12”>**

**4 <h1> Recipe Name </h1>**

**$ </div>**

**$ </div>**

**$ <div class=”row”**

**$ <div class=”col-xs-12”>**

**$ <div class=”btn-group”>**

**$ <button**

**$ type=”button”**

**$ class=”btn btn-primary**

**$ droopdown-toggle”>**

**$ Manage Recipe <span class=”caret”></span>**

**$ </button>**

**$ </div>**

**$ </div>**

**$ </div>**

That's the button opening the dropdown, below it, I'll have an **unordered list** with the **class dropdown**-menu and that is exactly the same setup as we chose in the **header** here where we also had this dropdown menu, unordered list because that's the nice thing about Bootstrap, dropdowns work the same no matter if you place them here in the header or use a separate button for them and then here, the **list items**, the separate actions I want to have. The first one pointing to just **hash** for now should actually allow us to **add ingredients to shopping-list** maybe, let's see if that is too long of a string.

**$ <ul class=”dropdown-menu”>**

**$ <li><a href=”#”> Add Ingredients to Shopping List </a></li>**

Then I want to have a button which allows us to edit recipe and one which allows us to delete the recipe, maybe let's rename the first one to shopping-list so that it's not that much longer than the other ones.

**$ <ul class=”dropdown-menu”>**

**$ <li><a href=”#”> Add Ingredients to Shopping List </a></li>**

**$ <li><a href=”#”> Edit Recipe </a></li>**

**$ <li><a href=”#”> Delete Recipe </a></li>**

With that, that should be a working dropdown but we're not going to find out before we implement it, something we'll do after the directive section. Now below this dropdown here though, I want to add **another row and in this row here** with another column spanning the full width, I want to output a **description of this recipe.**

Well and then at the very bottom, I'll add the last row for now, also with a full width column, here we'll output the ingredients. We're not storing any ingredients on our recipe now but we'll do so soon and this is a nice place to output it.

**$ <div class=”row”>**

**$ <div class=”col-xs-12”>**

**$ Description**

**$ </div>**

**$ </div>**

**$ <div class=”row”>**

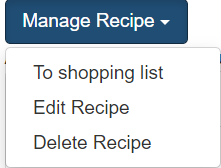
**$ <div class=”col-xs-12”>**

**$ Ingredients**

**$ </div>**

**$ </div>**

So that is the recipe-detail component, having all the logic inside of it which we need to display the details and then having some options to go somewhere else in our app or to trigger something else in our app like editing the recipe, we'll of course add all those features throughout the course and now with that added and set up here, let's see how that looks like. Looks pretty good, we got the name, the description, ingredients, a button which doesn't work of course, we're going to fix this later in the course and we don't see an image at the top because right now, image doesn't have a source or anything like that.



Now of course it would be nice if we could select a recipe here in the list and display it in this recipe-detail component but one issue we have with this is, we can't get our selected recipe into the detail component because the detail component is added here in our recipes component where we also have the list and communicating between the two components, list and detail, is something we don't know how to do right now. We have the same issue in the list component where we had to put in the code for a single item temporarily because if we would outsource it to the recipe-item component, we couldn't pass our recipe in this loop of recipes to that component.

So that cross-component communication is a huge issue right now but no worries, that is something we're going to dive deeper into in the next core section and thereafter, we'll come back to this and enable all that communication which allows us to outsource this code into its own component and make these links clickable so that we actually load the right recipe on the right. Before doing this, let's wrap up this section here, this first part of our project by working on the shopping-list feature.

**#### 55. WORKING ON SHOPPINGLIST COMPONENT**

We finished for now all our recipes components, let's work on the shopping list now. Now the shopping list component (shopping-list.component.ts), if we have a look at it here in our files, here in the HTML file, we already added some content to its template here but instead of saying the list **(<p>the list </p>),** I want to output an **unordered list** of **type list-group** or of the **class list-group** I should say, so using this Bootstrap class and in there, I want to have clickable list items, so I'll **add an anchor tag** pointing to just **hash** for now again to make sure it doesn't reload the page accidentally and then add some Bootstrap classes or one class to be precise, the **list-group-item class** to make this look like an item in this unordered list and I'll even get rid of the anchor tag here, of the ***href*** I should say and therefore set an inline style to enforce a cursor which looks like the pointer.

**$ <ul class=”list-group”>**

**$ <a class=”list-group-item” style=”cursor: pointer”></a>**

**$ </ul>**

The reason why I'm removing this is, I will later use this like a button and therefore, I don't need this ref element there, it would simply lead to unwanted behaviors. So that will be a single element, a single item on our shopping list. Now to display something here, we need an array of ingredients though.(**shopping-list.component.ts**) So what I'll do for now is, I'll **add my ingredients** property here to the shopping list component.ts and this will be an **empty array** ( under the export class) but now just like with the recipe, we're going to use ingredients a lot throughout our app, also in the recipe section, so we probably should also create a model for this. Let's do this in the next lecture.

**$ export class ShoppingListComponent implements OnInit{**

**$ ingredients = [ ] ;**

**#### 56. CREATING AN INGREDIENT MODEL**

We're going to use ingredients a lot in our application, so having a **model** for it makes sense. Now where do we store this **ingredient model**? We store it in the recipe model in the recipes folder because it belongs there feature-wise but where does the ingredient model belong to?

Well, I believe it belongs in a new folder, the “**shared**” folder which I'm going to create in the app folder. Shared is a fitting name for this folder because it will contain features or elements of our app which are shared across different features, like the ingredient which we're going to **use both in the shopping list and the recipes section**. So here I'll create a new file and this file will be named **ingredient.model.ts**. In there, I will **export a class named ingredient** and here, we will basically define how an ingredient should look like.

$ **export class Ingredient{ ………………………………}**

Now how should it look like? It should have a **name**, so **public** available, **which is a string** and an **amount**, so how many items of that type should we have. So maybe **amount here** which would be of **type number**. Now in the ***constructo***r, we of course want to be able to receive that as arguments, **name** and amount and then assign it here as before, so ***this.name*** here equals name and ***this.amount*** equals amount. Now that is such a typical set up, that we receive the arguments we want to assign in the constructor and that instantly assign them, that TypeScript offers a shortcut. We can get rid of the declaration of our properties up here and get rid of the content in the body of this constructor and simply add an accessor in front of the property name here, so public in front of both arguments.

**$ export class Ingredient {**

**$ public name: string;**

**$ public amount: number;**

**$ constructor( name: string, amount: number) { };**

Can also be written as

**$ export class Ingredient {**

**$ constructor(public name: string, public amount: number ) { }**

What that will do is behind the scenes, it will create the same effect we had before, so it will automatically give us properties with the names we specify here as argument names, so name and amount in this case and it will automatically assign the values we receive in this constructor to these newly created properties. So nice and convenient shortcut. With that, we defined how ingredient should look like, now we can use it in the shopping list component. Let's do this in the next lecture.

**#### 57. \*\*\*\* CREATING AND OUTPUTTING THE SHOPPING LIST**

Let's use our newly created ingredient model( **Ingredient = [ ]** ), so just like with recipe, we can now define the type for our ingredients array and this will be of type ingredient array and of course for this,

export class ShoppingListComponent implements OnInit {

  ingredients: Ingredient[] = [

we have **to import** ingredient, so let's quickly do this from and now we have to move up one folder so that we are in the app folder and then we have to go into the shared folder and there, it's in the **ingredient.model file**, without the file extension. ( the imports automatically gets added by the IDE when you type ingredients: Ingredeint[] ) With that, we can now create one first ingredient, so let's use our ingredient constructor and we have to pass name and amount here. So the name could be apples and we want five let's say and then let's also say we want to have some tomatoes, maybe 10.

**$ new Ingredient(‘Apples’, 5),**

**$ new Ingredient(‘Tomatoes’, 10)**

That is our shopping list for now, now we want to output it here in the component(**shopping-list.html**). So we get an anchor tag here which represents a single item and now you could put this into its own component too but since this is going to be such a straightforward easy item, I don't see much sense outsourcing it into its own component, though it wouldn't necessarily be bad to do so but I also don't see a great advantage, there also isn't a lot of business logic attached to this.

So I will place ngFor on this anchor tag to loop through all the ingredients, so ingredient of ingredients and now we can output the ingredient data here inside our anchor tag.

 <a

            class="list-group-item"

            style="cursor: pointer"

            \*ngFor="let ingredient of ingredients">

So it makes sense to output the ingredient name I believe and then maybe in parentheses thereafter and these parentheses here are no Angular syntax, that is normal text here I'm outputting, inside the parentheses, I'm going to use string interpolation to also output ingredient amount.

{{ingredient.name}} ({{ ingredient.amount }})

            </a>

With that if we now save this, we already see our shopping list here at the bottom, the last missing piece I want to add is this shopping list edit section here which should display us an input field which then allows us to actually edit our items or add new ones, at least later in the course, it will add this functionality.

**#### 58. \*\*\*\* ADDING SHOPPINGLIST EDIT SECTION**

So let's work on the shopping list edit component.html. In its template, I want to add a Bootstrap **row** and in there, I want to have a **column spanning the full width**, like that and in there, I'll **add a form element**. Now important, remove the action( **<form action=””>** ) because we're not going to send a request to the server once this is submitted but we will have a look at submitting forms later in this course, we have a whole section about that and that is when we will fill this form with more life.

For now in this form, I want to have a row and this should now maybe have a width of only five on small devices, on very small devices it should span the full width which is why I'm using the ***sm class*** here and it should also have the form-group Bootstrap class attached to it, this div and in here, I'll have a **label** for the name of my items, so I'll say name here as a text and I'll add an **input of type text**, that's correct with ID name so that this label refers to it and then here, this should receive **class form-control** to make it look nice in Bootstrap's world.

<div class="row">

    <div class="col-xs-12">

        <form action="">

            <div class="row">

                <div class="col-sm-5 form-group">

                    <label for="name">Name</label>

                    <input

                    type="text"

                    id="name"

                    class="form-control"

                </div>

Next to it and that is why I chose to not give this column the full width, I want to have another div with let's say 2 **( col-sm-2** ) because it's going to be a very **small input**, then also the form-group class here and in there, another label for the **amount**, so let's say amount here and then an **input of type text here** or maybe even of type number to be honest which gets the ID, amount and then a class form-control.

  <div class="col-sm-2 form-group">

                    <label for="amount">Amount</label>

                    <input

                    type="number"

                    id="amount"

                    class="form-control"

                </div>

            </div>

So that's give us the inputs the user will later ***use to add new ingredients***, below my inputs here, I'll add another row which will now span the full width and here I want to have a couple of buttons which allow me to manage this item creation or editing process. The first **button** I'll add here with the style of **btn btn-success** to make it look green in Bootstrap's world will be of **type “submit”** because this button should actually submit the form later and I will say “***Add***” there because it allows us to add a new item. The **second button** will be of type btn because it should not submit the form, this button should simply ***delete*** the selected ingredient because later we will be able to click an ingredient and it will then be loaded into the input fields up here and if we click delete, well it simply will be removed, so delete item or just **delete** maybe and then a last one which allows us to **clear the form**.

If we loaded the item we wanted to edit and then decided to not do that, I want to have a button which allows me to clear the form, so here I will maybe make this blue with button primary. If we save this, this is actually how I want this form to look. We have an input field for the amount, for the name and our buttons which later will be filled with some life so that we can manage this ingredient.

 <div class="row">

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

                    <button class="btn btn-success" type="submit">Add</button>

                    <button class="btn btn-danger" type="button">Delete</button>

                    <button class="btn btn-primary" type="button">Clear</button>

                </div>

            </div>

**#### 59. WRAP UP AND NEXT STEPS**

So that’s the first basic set up, of course we can improve this app for example I don’t want o have recipes and shopping lists component on the same page. We will fix this soon first temporarily and then with a real good solution later in the course and I also want to have my components communicate with each other so that we can actually store the contents of the items in our recipe-list in their own component that we an select a recipe here and load it in the detail section and so now. For now though that is how far we can go with the knowledge we gathered the blocking thing right now is the component communication and that’s why we’re going to dive into this in the next course section see you there and then see you back in this project later in the course.

**## SECTION 4: DEBUGGING**

**#### 60. UNDERSTANDING ANGULAR ERROR MESSAGES**

Welcome to another course module. By now, the basics should be set and we had a closer look at what makes up an Angular app. Now chances are you already encountered your first error in your app, we even had some errors together in the apps we built. Now fixing these errors is of course very important and not always straightforward, that is why in this section, we're going to have a brief look at the tools we have at our disposal for fixing errors in our Angular app.

Here's a little example app attached (**debugging.zip file** ) to this lecture which looks nice but won't work as expected. We are on our servers dashboard you could say but if I click add server to add a new server, watch what happens, we get an error. Now you might get such errors a lot and generally, you might not always have your developer console open, though for developing that's a good idea but if something doesn't work the way you expect it to work, like here when you click the button and nothing happens, opening the console, the developer tools and there, the Javascript console is a great idea.

Now here, we clearly get an error message and error messages have gotten better in Angular, so chances are the error message is indeed helpful. So let's have a look at the error message, you should find it right at the top and here it says that in app component, that's the first important information piece, in the inline template, then this would be the line number and then the column where in the line, so line 4, now let's see if that is helpful, that there the error was caused because we cannot read the property push of undefined. Well, we get a couple of useful information pieces here, so let's go to the app component then and let's have a look at it.

If we go there, well the line 4 argument is only kind of helpful because line 4 is neither the problematic code in our template nor in the TypeScript file. The report says line 4 because in the end everything gets merged together and rebuilt, so that is where probably the code then is triggered in line 4 in the final code but not in our code. Still, we get one important clue here, property push of undefined. So we try to call push on something which is not defined at the point of time we do call it. So if we have a look at our code here, we see that there is only one place where we call push and that is of course here in the onAddServer method which makes sense because this is the method which gets triggered once we click the button.

So that would have been another way of finding where the problematic code has to be as this error occurred when we click the button, it probably is somewhere in the method which is then executed. So here there's something wrong with pushing, though not with pushing itself, remember it said push of undefined. So the thing we're calling push on is undefined, so that would be this servers. How can these servers be undefined, we do declare it up here, right? That's exactly the issue in this case. We do declare it here but we don't define it, we tell TypeScript, there's going to be a property called servers but we don't initialize it.

It doesn't have any value assigned to it, it's not an empty array, it's not a string, it's nothing. So to fix our code, we simply have to make sure that before trying to push a new item on servers, we set servers to an array because otherwise, it could be anything, it's undefined in the end, so pushing is not supported here. So what we can do is we can simply initialize an empty array at the beginning and with this little change

in place, this should now work as expected once it reloads and we click add server, now we can add our servers to this list.

So this was a brief example for a potential error message, how to read it and how to try to get to the root of the error. Now of course, it's impossible for me to show all possible error messages and what can go wrong and there are some more tricky error messages but it's always great to not panic but have a look at the message, read which file it points to, read the exact message, there are even error messages in Angular which are much clearer than the one I just showed you. And with that, you should be able to narrow down the issue and hopefully, solve it.

**#### 61. DEBUGGING CODE IN THE BROWSER USING SOURCEMAP**

**Project name ( debugging )**

Sometimes reading error messages is not enough or you don't even get an error message, like here. I get my servers here right and if I click on them, I can delete them and it seems to work until I click the last one or if I deleted all of them, the only one. So deleting the last server is not working, I can click it as often as I want, it's not going away.

Now if we have a look at our code, at the template first, we see that on each list item, I call onRemoveServer and I do pass i, the index of the current iteration to that method. Now of course we could simply dive into the TypeScript code here and have a look at this and there we see, well we get the position of this item and then we simply call splice to remove the item, the element in this array at this position.

this.servers.splice(position, 1);

So you might say well that looks all right, sometimes it's just not super easy to debug a message like this because we're not getting an error message but we get a logical error in our app. Now debugging is supported by developer tools in a browser though, so it would be great if we could simply check at runtime why this is not working. In Chrome, you can simply go to sources and here, you will find all these sources imported on this page.

So in this case, all our script bundles and there are a couple of bundles as you'll see, the important one for us is of course the main bundle and if we have a look at the **main bundle**, yes that's not super easy to debug, right. Now theoretically of course, we can have a look at it and there, let's find, this seems to be our template here with container and row and maybe we can find the line responsible for splicing, here and now maybe we want to debug here and if you try to place a breakpoint here which you of course can do in your developer tools, you'll see something changed, it jumped to the app.component.ts file.

Now remember, TypeScript is not what runs on the browser, so there the Javascript, these bundles here are used, how can we now access TypeScript? Because these Javascript files support source maps.

Source maps are a little addition the CLI kind of adds to our bundles which allow the browser to translate our Javascript code to TypeScript or to simply map the Javascript code to our TypeScript files. In development-only, these source maps will be stripped out for production of course. So this is a great feature because now we can place breakpoints here in our TypeScript code and if I now click somewhere, you see it indeed pauses and now we can check and even see that ID currently is 2 which makes sense because I clicked this item here which has an ID of 2, just to show this again, here now with this item, now it's ID 1 because again the middle item which is ID 1 in this case because we started 0 for the first item and then we see that position is undefined right now.

Now we can move on a step, now we see position is 2 and now we can also check servers and in servers, we see position 2 of course is the last server in the list. So if we let this continue, it looks like the middle one was removed but actually, the last one was removed and we can confirm this by checking this if I click on the last one which was the thing which was not working, we see ID is 1, makes sense, it's the second element of 2 and the ID starts at 0, so ID 1 makes sense but now position is 2 because we add one to the ID. So now we try to delete the element at position two in this servers array and clearly we don't have that element here, we only have two elements with positions 0 and 1.

So the logical error in our code is that we add 1 to the ID here and that is how we could debug that

with the TypeScript code here, with that source map, that of course would allow us to now fix this bug.

Now correctly, you might say that's awesome, that's great but if I have to scan my main bundle to then click somewhere to open the TypeScript file, that's not really a great tool because if that bundle grows bigger, it gets even harder to find the right spot.

Therefore you can directly access your TypeScript files, under **webpack** here, you should find a couple of subfolders and if you open up the **dot folder** here and then **source,** here in the **app folder** and so on, you find all your TypeScript files, so here is where you can directly access your TypeScript files in the same structure as in your project to easily find the file you want to dive into and you want to debug with the Chrome developer tools. So that is another great tool at your disposal, besides reading error messages, using the debugger with source maps, with TypeScript files to either of course dive into errors for which you have error messages but also into logical errors, like here.

**#### 62. USING AUGURY TO DIVE INTO ANGULAR APPS**

So in this section, we learn about how to read error messages and how to use the debugger with source

maps. There is a third tool you can use when working with Angular apps, it's called augury, so you can simply google for this, Angular Augury. If you click on this, you should get taken to the official webpage where you can click install which should take you to the Chrome store since this is a Chrome extension.

Now I already added it, you can add it by clicking up here on add to Chrome and once you added it, you can open it in your developer tools, maybe you need to open and close your developer tools to see it and then pick augury here. Now Augury is a nice little tool which allows you to analyze your Angular application.

So if we reload our app here, we see for example all our components and here we only have one and on this component, you can jump into the state and see the properties of this component and also see the elements of that and for more complex applications, you can have a look at the injector graph and injection is something we will cover in one of the later modules but basically which will allow you to see dependencies between your components and your services in your app, again this will be taught in a later module. When we come to routing, you can also have a look at your routes here and how they work together.

You can check your NgModules like the app module, what it imports, what it declares, what it offers

you. So that is a nice little tool to understand your app at runtime, to see the app dependencies and jump right into them and to be able to analyze them. So with these tools, reading the error messages, using source maps to easily debug and using augury to understand the application and your dependencies, you've got a lot of great tools which should really help you debug your applications and create great Angular applications therefore.

**#### 63. MODULE INTRODUCTION**

Welcome to a new module. You learned the basics and we set up our first project, additionally we had a look at how to find and fix errors. Now I want to dive deeper into components and data binding and how you can connect both concepts even closer to now make the next big step in your application.

Let's start with an example project, you can find it attached to this in the next lecture and enhance this project to take advantage of components and data binding to make it a better app.

**#### 64. SPLITTING APPS INTO COMPONENTS**

That's the project you can find attached to this lecture here(**cmp-databinding-start.zip, but look for cmp-databinding-start-mysolution**), it allows us to define a server name here(**server name**) and then some content(server content) for the server and then we can either “add a server” or a “**add server blueprint**”

and as you can see, both look pretty similar but the server is bold and red or has some bold and red text,

the blueprint has some italic blue text.

So that's the difference here.

Now this app is all running in one component as of now, in the app component.html , here in the template, we got a lot of code and we can do better, we can split this up because that's not ideal having all that logic in this component here, maybe splitting it up makes sense because that would also allow us if we build a bigger app to reuse parts of that.

So how could we split this up? We could “create a new component” for this whole ***cockpit*** here where we enter a name and content and click the buttons and of course we could also create component, a component for an individual server and with that, we should have a much leaner app.component.html file and we would bundle our business logic better into individual components.

Now we learned how to create new components, I will use the CLI, you could of course also do it manually and I will create “***new components with ng g c cockpit –spec false***” and then let's say the first component could be this cockpit here, so I will simply name it “cockpit component”.

Now as an addition, we can type “--spec false” to make sure this spec, this testing file is not getting created, I deleted it all the time anyways and then I will repeat the step for another new component which I will simply name “***server-element***”, server-element because it could either be a server or a blueprint

in this example.

So with that, we get two new folders in our app component, so two new components, thus far nothing you don't know. Now we can of course take our code here from the “app.component.html” file from this template, so for example this “first bootstrap row here”( div class=row), cut it here and add it to our cockpit, like this. Now of course if we remove the template part here, we also have to remove or we have to move the methods we're calling here(app.component.ts), so “**onAddServer** and **onAddBlueprint**”. So in the TypeScript code here, I will copy both or I will cut both elements here and move them over to our cockpit and we'll soon also come to ngOnInit and what this does as a side note.

Now of course my issue is that I try to push these two arrays or to an array which doesn't exist here, serverElements, that array is still in the app component. We could of course move that, oh and maybe I should also move that closing curly brace. We could of course move this serverElements array but then we would no longer have it in our app component and we will need it there too because there we still will need to replicate the serverElements to create the servers we output here.

So as a first step, what I can move of course, cut and paste are the two properties(**newServerName**, **newSErverContent**) which are used here with two-way binding, so I can **move that to the cockpit** but we will have to see what we do about this serverElements array.

For now let's go back to the “app.component.html” file and take a single server here, so this **panel div** here. So with that, all we have here now is the row and the column and the other part will go into the HTML file of the server element because that is a single server.

Now here of course, we have this loop**(\*ngFor**) and that's not what I want to have in here, so I will get rid of it there because I still want to manage my array of serverElements in the app component because we somehow need to be able to manipulate from the cockpit and then replicate it in the app component to create multiple instances of this server element component which is only a single server.

Keep this in mind, this is not the list of servers. Of course you could set this up differently but this is the approach I will go with and this approach won't work right now, we also tried to access an element property now inside of our server element component which we also don't have.

So a couple of things to fix, I like the general way we split up our app and that we now can add our “<app-cockpit></app-cockpit>” here (app.component.html) where we before had the full HTML code we moved over there and that we can add our “<app-server-element></app-server-element>” component here and that we could also replicate this with ngFor, ***let serverElement of serverElements***, (<app-server-element \*ngFor=”let serverElement of serverElements”></app-server-element>”)so we can of course also place ngFor on a component as you already learned before.

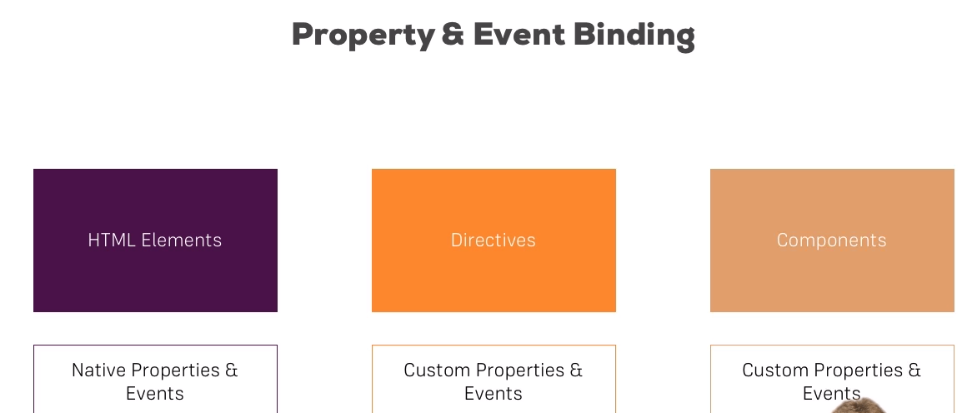
So I like that but I don't like that the app is broken and if we have a look at it, that we simply get an error that a couple of properties are unknown. So we somehow and that is a key part, need to get the new server we add in the cockpit to this array of servers in our app component, so we kind of need to inform our app component that one of its child components, the app cockpit changed or that something changed there to which we should react, a new server was added and then we also need to get the data we got from the cockpit, so this new server which is managed in this array of servers in the app component, we need to get to individual server of a single loop iteration here down to the app-server-element. So we need to pass data between components and that of course is a super important task we haven't touched yet but we will dive deeper into right now.

**#### 65. PROPERTY AND EVENT BINDING**

In the last lecture, we split up our app into multiple components which is great that is why we use components but one issue is that we need to pass data between these components.

Now in the basics sections, we learned about property and event binding and there we used it to, for example bind the disabled property of a HTML element.

So we kind of passed the data to that element, we passed the information that it should be disabled, that this is set to true to this HTML element. The same with event binding when we click this HTML button, something happened. This buttom emitted an event to which we are able to listen so it send us some data. The same for the input where we even used that data with $event if you remember this. there, we get data from the input element. This is exactly the behavior we need now with our own components, we need to be able to send data into a component or receive data, receive an event and Angular of course gives us agreat tools to implement this flow.



We can use property in event binding not only on HTML elements and their native properties and events as we did thus far we can also use it on directives and we also did this with ngClass and ngStyle. There we use property biding but and that is important. We can also use it on our own components and bind to our own custom properties and custom events.

We can emit our own events that is why I’ll dive into in the next lecture starting with custom property binding.

**#### 66. BINDING TO CUSTOM PROPERTY**

So we learned that we can use property binding to bind to our own properties, properties of our components, now this is what I'll start with.

So to temporarily prevent my app from crashing, I will simply comment out the code in the cockpit (comment out **onAddServer() and onAddBlueprint()** )so that this doesn't give us any errors anymore for now, we won't be able to use it but it will not crash. Now in the server components, server-element component, if we have a look at its HTML file, you'll see that we try to access the “**element**”, the single element, the single server element and therefore, we should create a property in this TypeScript file, element which represents our server.

We can even assign a type by adding a colon ( **element: { } ;** ) and then simply defining the type which will be a Javascript object hence the curly braces and just because it's important, this is not the value, this is TypeScript syntax for defining the type to make sure that element may only have this type and we then know that an element will have what we're trying to access here, a type, a name and a content.

So we'll have a type and type on its own will be of type string ( **type: string**) , we will then have a name which will be a string ( **name: string** ) and we'll have a content which will be a string ( **content: string**). All these go inside the element: { } .

**$ element: {type: string, name: string, content: string};**

So that is all now our type definition for a property we want to use in our HTML code, we use here in all these places. Now still this element, this property is part of this component only, of the server element component, we can't access it from outside.

Now it would be great if in our app.component.ts where we manage our server or our array of serverElements I should say, if we could somehow access this property because let's say just so that we can see something, we start with one server here which is of

type server(**serverElements=[{type:’server’}]**, has a “**name: testserver**” and a “**content: Just a test**” and

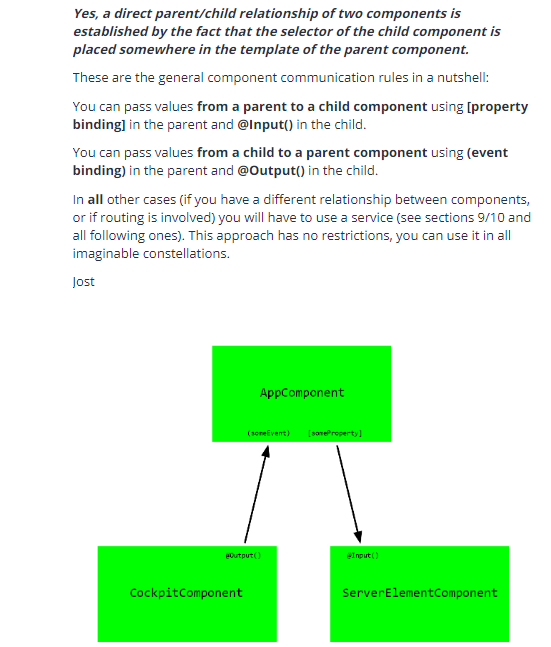
**$ serverElements = [{type: ‘server’, name: ‘Testserver’ , content: ‘Just a test!’}]**

just to be super clear, these colons here are not type definitions, we're on the right side of the equals sign, so here we are assigning a value and the value here simply is a Javascript object and in Javascript objects, we simply assign key-value pairs or we create key-value pairs by having the key, then the colon and then the value, so normal Javascript syntax here.

So now, we get our ***serverElements array***, we're looping through this array here on the **app-server-element** in the app-server-element, we have our element property, so a property in that component. Now we want to access this component from outside, so it would be great if we could bind to it, just like we were able to bind to disabled. So maybe add square brackets, enter the name of the property in between,

so the name of the property in the component, here this ***name( “element” )*** and then assign the serverElement, so simply the element we store for each iteration in this variable here ***( [element=”serverElement”]).*** It would be great if we could do this and if we try this and our app reloads, we see that we get an error that we can't bind to element ( ‘app-server-element’ )because it isn't a known property of app-server-element.

Now you would say that is not true, it is a property of server element. Here we defined it as a property, it's even public right ( public element: ), we didn't add private in front of it or anything like that, so why can't we access it? ***Simple answer, because by default, all properties of components are only accessible inside these components, not from outside and that generally is a good thing.*** You don't want to make all your properties bindable from outside. So you have to be explicit about which properties you want to expose to the world so to say. So if you want to allow parent components to be able to bind to this property, you need to add something to that element property, you need to add a decorator.



Remember, I said that decorators are not only available for classes, here we're going to add onto this property, the decorator we need to add is **@input** and here important, you need to execute it, it's like a function in the end, so add parentheses( **@Input( )** ) and input now needs to be imported from **@angular/core. ( import { Component, OnInit, Input } from @angular/core** ). With this in place, now we are successfully exposing this property to the world, so now any parent component, any component hosting our server element component, so implementing it through its selector like we do here, is now able to bind to element and I can demonstrate this if we save this, you now see just the test here, you see our component because now we replicate this component for each element in the array and we successfully can pass the element, the property down to that component and bind to the element property in that component. And this is how you can set up custom property binding on your own events, exposing your own properties to the components where you implement these components through the selector.

**#### 67. ASSIGNING AN ALIAS TO CUSTOM PROPERTIES**

In the last lecture, we learned something crucial, we learned how to bind to our own properties with @input parentheses in front of it, super important.

Now you can even configure this a bit more, sometimes you don't want to use the same property outside of the component as you use inside of it. So inside this component, you might say element is exactly the property name I want to use because it makes the most sense but outside, so here(app.component.html [element] ), you maybe don't want to bind to element, maybe you want to bind to **srvElement** to make it clear it's a server element, anything like that. Now that would of course not work because now you would try to bind to a property, Angular doesn't know that it would be bindable because there is no server element property in this component with @input in front of it.

You can assign an alias, you can pass an argument to @input with the property name as you want to have it outside of this component, so srvElement could be added here ( **@Input( ‘srvElement’ )** ) and now from the outside, so from the component where you implement this component, you now have to target srvElement if you want to bind to this property(@Input(‘srvElement’) ). Element will no longer work, it has to be srvElement now which is why now with these two places changed in the app component and in the server element component, we see the same as before but now binding through the alias. And just again to be really sure, to make this really clear,

now you have to use s**rvElement, element will no longer work**. So that's a little bonus, a little alternative in case you want to bind to some other property name outside of the component, this is how you assign an alias.

**#### 68. BINDING TO CUSTOM EVENTS**

Great! In the last lectures, we learned something crucial, something we will use a lot in our demo app in this course and probably in any app. How to pass data from a component down to another component which was implemented there.

Now about the other direction, what if we have a component and something changes in there and we want to inform our parent component? So the component which implements the other component?

For example here in app.component, we’re implementing **<app-cockpit>** and in this child component, in app cockpit something can change because here we got two buttons and when we click the buttons, we want to do something and right now the code which would normally get executed was simply commented out ( onAddServer( ) and onAddBlueprint( ) these two are commented out )

So here, we want to inform our parent component the app component in this case, that a new server or a new blueprint was created.

So in our app component, we maybe still want to implement these two methods(onAddServer( ) and onAddBlueprint( ) are the 2 methods ), so lets simply add them here (inside the app.component.ts) .

However I will adjust the names and say “onServerAdded (previously onAddServer() )” because these methods will kind of only be executed after this button has been click, so not once the button is clicked but thereafter, once we really already are done creating the server so to say. So “onServerAdded” sounds like more appropriate name here in the app component.

Then however, we can un comment out this code:

**$ onServerAdded( ){**

**$ this.serverElements.push({**

**$ type: ‘server’,**

**$ name: this.newSErverName,**

**$ content: this.newServerContent**

**$ })**

and still create a new server or new blueprint. However this will not work as expected here because we’re referencing “**this.newServerName**” and “**this.newServerContent**” which are not available in the app component. But that is exactly information we kind of want to pass from the cockpit to the app component because the app components is the central place in our app for now which manages this array of servers.

So how can we emit an event our own event, how can we inform our app component? It would be great if it would look something like this, on the app cockpit. So in the app.component.html file where we implement app cockpit it would be great if we could listen to

“**serverCreated”(<app-cockpit (serverCrated)>** any names you like but that would be a fitting name for the kind of event we want to listen to.

Of course this event doesn’t exist by default so just like we listen to click now we want to listen to some event like this, serverCreated and once a server was create we execute some code. So just like in any other build in event like click here on the right side of the equal side between the quotation marks, we want to call “**onServerAdded**” the method we’ve prepared for this

**(<app-cockpit (serverCreated)=”onServerAdded()”></app-cockpit>).** and we even expect to get some data with **$event (onServerAdded($event)** )this is how you could catch this data, that could be some object which gives us the required information about that server, like the name and the content.

So in on “**onServerAdded( )”** now method, we would expect to get event data or lets say we would actually expect to get the **serverdata(onServerAdded(serverData: { }) )** and that could be of type lets say JS object which has a name filed with a string or lets be more explicit. Lets name it serverName which is of type **string({serverName: string})** and “s**erverContent**” which is of type **string({serverContent: string}).** So that would be the type of object we expect to get in this method and with that we could use it here(replace **name: this.newServerName** with **name: serverData.serverName**) and use server data, serverName to assign it as a name for our newly created server, the one we push onto our array here and the same of course down here for the content

**$ (replace name: this.newServerContent with name: serverData.serverContent).**

So that is how it should work. We get this data in “onServerAdded” and we call “onServerAdded” once our custom event occurs on the cockpit component and this event even should give us this data we expect to get and we catch it with $event. Now obviously we can also replicate these for the lets say “blueprintCreate” event, here we would execute “onBlueprintAdded” and of course you could alos kind of merge these two methods(**bluePrintCreated)=”onBLueprintAdded($event)”** ).

I keep them separate to be really clear about what’s happening here and there again we also expect to get our blueprint data in this case:

$ **onBlueprintAdded(blueprintData: { serverName: string, serverContent: string})**

**$ onBlueprintAdded( ){**

**$ this.serverElements.push({**

**$ type: ‘server’,**

**$ name: serverData.servername,**

**$ content: serverData.serverContent**

**$ })**

So blueprint data maybe and there we also would receive the information we require like the serverName and then of course also the content this is how it should work. Now in the cockpit component w actually need to emit our own event. We are waiting for events anme “serverCreated” and “blueprintCrated” maybe with a lowercase p.

So in the cockpit component I kind of want to crate two new properties here properties that’s important serverCreated and blueprintCreated.

**$ export class CockpitComponent implements OnInit{**

**$ serverCreated; // properties**

**$ blueprintCreate // properties**

**$ newServerName = ‘ ‘;**

**$ newServerContent = ‘ ‘ ;**

Now before we placed @input in front of them to mark them as properties you can set from outside now we want to do the opposite we want to make sure that both properties kind of are events we can emit and to make them events we first of all have to assign a new value, new eventEmitter and eventEmitter needs to be imported from @angular/core

**$ export class CockpitComponent implements OnInit{**

**$ serverCreated = new EventEmitter;**

So you should add this import here at the top. eventEmitter is a generic type which is indicated in Typescript by using this smaller than and greater ( < > )than sign and in-between, you simply define the type of event data you’re going to emit and here. Well we know this would just be data we expect in the app component so this will be an object with serverName and servercontent so we can pass this in between the smaller and greater than signs.

**$ export class CockpitComponent implements OnInit{**

**$ serverCreated = new EventEmitter <{serverName: string, serverContent: string}> ;**

And important add parentheses in the end to call the constructor of eventEmitter and crate a new eventEmitter object which is now stored in serverCreated.

**$ export class CockpitComponent implements OnInit{**

**$ serverCreated = new EventEmitter <{serverName: string, serverContent: string}> ( ) ;**

I’ll do the same for **blueprintCreated.** Now we get two event emitters and that’s the first step, event Emitter is an object in the Anuglar framework which allows you to emit your own events. What we can do with that is in onAddServer, we can now call serverCrated (this.serverCreated) and here we can call the emit method(emit( ) ).

This will, well like the name implies emit a new evetn of this type so of the serer created here using this eventemitter. Well and here we simply want to create the object or pass the object we kind of said we would pass here( emit( { } ), an object where we have a server name which of course is our new serverName so it would be whatever we save here and then a serverContent with the value of newSErverContent as a value

I can split this over two lines to make it a bit easier to read.

**$ onAddServer( ){**

**$ this.serverCreated.emit({**

**$ serverName: this.newServerName,**

**$ serverContent: this.newServerContent**

**$ })**

**$ }**

So this is what we emit in “onAddServer” and lets do the same for “onAddBlueprint” here however of course we use blueprintCreated, the second event we set up there.

**$ onAddBlueprint( ){**

**$ this.blueprintCreated.emit({**

**$ serverName: this.newServerName,**

**$ serverContent: this.newServerContent**

**$ })**

**$ }**

So with that, we’re emitting our own events pretty awesome we’re passing the data one piece is missing though. Remember we added @input to make a property bindable from outside, now we need to add something to serverCreated and bluepringCreated to make it kind of listenable from outside. That something is another decorator and its not input because we’re not getting something passed into this component its @output with parentheses because we’re passing something out of the component, we’re passing our event out of the component

$ **@Output() serverCreated = newEventEmitter<{serverName: string, serverContent:strng}>( )**

and the same for blueprintCreated.

$ **@Output() blueprintCreated = newEventEmitter<{serverName: string, serverContent:strng}>( )**

Make sure to import output from @angular/core. With this in place, now if we save this and have a look at our application, if we create a new server some test content and we click add server you see it gets displayed down there so it successfully emits this event. Lets check the same for the blueprint that looks pretty good too.

So with that we get our app working again but now split up into several components which can communicate with each other and that is the key building block because component communication is so important.

**#### 69. ASSIGNING AN ALIAS TO CUSTOM EVENTS**

In the last lecture, you learned how to use @output to enable other components to listen to your

own custom events which you could create with eventEmitter.Now just like on @input, on @output you also can assign an alias but here for blueprintCreated, maybe you want to name this bpCreated:

$ **@Output(‘bpCreated’) blueprintCreated = new……………………………………..**

Now with this, this is the event you can listen to from outside. So in the app component where we try to listen, we now need to rename it here to and listen to be bpCreated, which is now just like on @input,

the one and only event you can listen to when it comes to this blueprintCreated event.

**$ <app-cockpit**

**$ (serverCreated) =”onServerAdded($event)”**

**$ (bpCreated)=”onBlueprintAdded($event)” //replacing blueprintCreated**

**$ ></app-cockpit>**

The other name, blueprintCreated here will not be exposed to the outside anymore, so it has to be

bpCreated now.

All the rest stays in touch and works just like before.

**#### 70. CUSTOM PROPERTY AND EVENT BINDING SUMMARY**

Let me quickly summarize what we've learned thus far because it really is important. This component communication is such a key feature and with @input and the ability to make your properties bindable from outside, from the parent component using this component and the same for @output which allows parent components using this component to listen to your own events which you created with new eventEmitter, these are such important features in your app.

We will use it in the recipe book in the next section because this really allows you to make your application

very dynamic, use many components and have them talk to each other. Now one issue you could say with that approach is that if you want to talk between two components sitting

next to each other, it can get kind of complicated to emit an event in one component, change something

in the parent component and then pass this new data down to the child component and these chains of inputs and outputs can really grow more complex.

So while this is absolutely fine here and this is a feature you will use a lot because it's not bad at all,

there are some use cases where the distance between two components that should talk to each other is

so great that building a chain of outputs and inputs is not very practical. That is why in the services section of this course, I will then show you another approach of having components talk to each other, though and that is important, this will not be a better approach, it is just a better tool for a specific use case.

A use case like this here is typically handled with input and output like we are doing it here, it's the perfect tool for this job and it's so important, you will see it getting used a lot throughout the rest of this course

**#### 71. UNDERSTANDING VIEW ENCAPSULATION**

I'm not sure if you noticed it in the last lectures, with our working application, if I add servers and blueprints, something changed. Our new server down here in the blueprint as well as the paragraph right at the top no longer has a blue color.

Now it was colored blue before because in the app.components.css file, we define that paragraphs should have a blue text color but both paragraphs were removed, they are no longer part of our app component here. The first paragraph is part of the cockpit and the second paragraph, that is inside of our server

Element but isn't that strange? Here on this CSS definition, we just say paragraph, so it should affect all paragraphs in our whole app.

Now you might say, well no it's in the app.components.css file so clearly it only belongs to this template

but is that so clear? Because the way CSS works, it doesn't really care in which CSS file you define a rule,

it simply is applied to the whole document normally. So this actually is a behavior enforced by Angular which is not the default behavior of the browser.

So whilst of course this CSS files have the goal of encapsulating styles for the component they belong to, this again is not a default behavior, Angular gives us this behavior and it's a great behavior because with that, we can make sure that whichever styles we define in a CSS file will only get applied to the component they belong to. However here of course, this means that now we would have to copy this blue color( p{color: blue; } here to all our other components where we want to use it. Well I never wanted to use it on the cockpit anyways but I will copy it to the server element and if we now change this and let it reload and enter something here and add a new blueprint, now you see this is blue again.

Now it's interesting if we inspect this text here in the developer tools, you see that if you have a look at the styles, clearly here we assign a blue color but do you see that? The selector changed, before we had just p here, just paragraph, now it's paragraph and then this very strange attribute, \_ng-content-ejo-2 .

p[\_ngcontent-coa-c2] {

 color: blue;

}

Well I don't know about you but I didn't add it. You can see this attribute here on the paragraph, that is why this style gets applied here because this style definition says make every text blue which is in a paragraph where the paragraph has this attribute, something which is true for the paragraph holding this specific text.

Now the other paragraph we have in app cockpit here in the div, this paragraph does have a different attribute, \_ng-content-ejo-1, down here we have ejo-2. That is why the styling gets not applied to this paragraph and all these strange attributes which you can see all over the place here, all these strange attributes are applied to their specific elements by Angular. Angular as I told you enforces this style encapsulation and it can't do that magically, so the way it does it is it simply gives the same attribute to all elements in a component.

So all elements in this cockpit component have ng-content-ejo-1 attached to it as you can see and it does this for each component with different unique attribute names, ejo-1 and ejo-0 and ejo-2 and with that, it can make sure that once it changed your style selector, it automatically adds this attribute selector to all your styles defined for a component, that these styles get only applied to elements of that component. This is how it enforces this behavior. It kind of emulates the shadow DOM, the shadow DOM is a technology not supported by all browsers where each element has its kind of own

shadow DOM behind it, where you then could assign styles to each element but as this technology is not supported by all browsers, this is how Angular emulates it and that is the default behavior of view encapsulation in Angular.

**#### 72. MORE ON VIEW ENCAPSULATION**

In the last lecture, you learned how Angular encapsulates your styles, now you can override this encapsulation though. If you go back to your code, to the server element component, you can add something to the @component decorator, it's called encapsulation, this property and there as a value, you can access ViewEncapsulation (encapsulation: ViewEncapsulation.Emulated) which needs to be imported from @angular/core, so make sure to add this import and then you can choose between three modes.

Emulated is the default, so you don't need to add this, none if you add this would make sure that if we have a look at our component again and we inspect our server element, that now in here, we don't see these strange attributes being added to our elements. Therefore, this component now does not use view encapsulation. The other components still will use it, there you still see these attributes but if you now define any styles for this component in the CSS file of this component, they will actually get applied globally and I can demonstrate this by going into this CSS file and if I change the label color and the label is in the cockpit, not in this component, to red, now you will see the label there is overwritten whilst this label of course still has its custom unique attribute, it still is a label in the end and in this server element component, we disable encapsulation.

So there our selectors aren't changed by Angular, they don't receive their unique selector, therefore they are enforced, they are used application-wide, also affecting other components. Might be the behavior you want, chances are that this is not the case but it's important to know that you can change this behavior.

Now besides none(viewEncapsulation.none) , you also could choose native( viewEncapsulation.Native) and native uses the shadow DOM technology. This should give you the same result as before with emulated but only in browsers which support it which is why for most cases, you want to choose emulated but be aware that you could switch to none or native here too.

So I will go back to emulated which again you wouldn't have to add, it's the default anyways, with that our labels should no longer be red now. So that is how you can change it and how view encapsulation works, a nice tool ensuring that by default, only your component receives the styles you defined for it

but as you learned, it can be overwritten

**#### 73. USING LOCAL REFERENCES IN TEMPLATES**

Back in the application we built throughout the first lectures of this course, with our cockpit and the

server-element component, with using output and input to pass data around our own custom property and event binding. Now in the cockpit, right now I'm using two-way data binding to get the serverName and content.

Now there is nothing wrong with that but maybe we don't need to use two-way data binding. Since I only

want to save or use the data at the point of time I click the button, the add server or the add server blueprint button, it would be enough to get the value of the input at this point of time and there is a nice way we can achieve just that.

So I'll duplicate this and comment out the old solution:

<input type=”text” class=”form-control” [(ngModel)]=”newServerName”>

and now I'll structure it out over a couple of lines and remove the two-way binding( remove ngModel) , there the other option we would have is to place a local reference on that element. A local reference can be placed on any HTML element, so not only on an input element, on anything you see here in the template and you add with a hashtag and then a name of your choice, like for example server name(#serverName):

<input type=”text”

class=”form-control”

#serverNameInput

*[(ngModel)]=”newServerName( this one got commented out ) ”>*

since this is what this reference will hold, a reference to this element, so maybe serverNameInput. Now important, this reference as I just said will hold a reference to this element, so not to the value we entered there, to the whole HTML element with all its properties. We can see this if we pass this now as an argument once we click add server, onAddServer.

Here:

<button class=”btn btn-primary” (click)=”onAddServer(serverNameInput)”>Add Server</button>

I can pass serverNameInput because the other important thing you need to know about references besides how to create them is that you can use them everywhere in your template but important, only there, not inside your TypeScript code, only in the template.

But we do call the method here inside the template of course, so here we can use serverNameInput and that is a way how we can pass it to the TypeScript code because in onAddServer, we now know that we will receive the name input,

onAddServer(nameInput){

console.log(nameInput);

this.serverCreated.emit….

……….

})

}

you could also name it serverNameInput and if we log this, we can analyze what this actually means, what we actually got here. So let's save this and let our app reload and now if I test something and enter something here and I click add server, of course this was broken here(you will see empty box) , this is broken because we're not fetching the data correctly right now, you see we actually get the input element itself. So this is the element we got here(when you inspect the element), that is what the local reference gives us, the element with all its properties:

<input \_ngcontent-lls-c1 class=”form-control type=”text”>

So here we could also output value,(console.log(nameInput.value) the value of this input since we know that an input element has a value and of course this will depend on which kind of element you placed your local reference on, not all elements have the same properties.

So with this if we try this again, you'll see we now print test because we access the value of this input to which we got access through the local reference. So local references, a very nice feature to get access to some elements in your template and then use that either directly in the template, you could also output serverNameInput.value here(under the <label>Server Content</label> )or you can pass it on, like we do to use it in the TypeScript code, a nice and handy feature. Now with that being passed, we can use it here to not only log our data to the console but instead here once we define it or once we create the server, we could access name input value:

onAddServer(nameInput){

this.serverCreated.emit({

$ serverName: nameInput.value ( *replaces this.newServerName*)

$ serverContent: this.newServerContent

$ })

$ }

And we should also be explicit about the type, we know that this will be a HTML input element:

onAddServer(nameInput: HTMLInputElement){

so we know that this value property will be there. Now the same can of course be done on the blueprint where we can always replace this newServerName with name input and then value.

onAddBlueprint(nameInput:HTMLInputElement){

this.serverCreated.emit({

$ serverName: nameInput.value ( *replaces this.newServerName*)

$ serverContent: this.newServerContent

$ })

$ }

Now with that, we can get rid of newServerName, we don't need it anymore, I will comment it out so that we still have it there for reference but the new approach now only uses this local reference. Now for this to work, we of course also need to pass this reference to our onAddBlueprint method.

<button class=”btn btn-primary” (click)=”onAddBlueprint(serverNameInput)”>Add Server</button>

So these are local references.



**#### 74. @ViewChild( ) in Angular 8**

In Angular 8 the “@ViewChild( )” syntax which you’ll see in the next lecture needs to be changed slightly:

Instead of:

$ @VieChild(‘serverContentInput’) serverContentInput: ElmentRef;

Use:

$ @ViewChild(‘serverContentInput’, {static: true}) serverContentInput:ElementRef;

The same change ( add {static: true } as a second argument ) needs to be applied to ALL usages of “@ViewChild( )”( add also @ConentChild( )which you’ll learn about later) If you plan on accessing the selected element inside of “ngOnInit( )

If you DON’T access the selected element in “ngOnInit” ( but anywhere else in your component) set “static:false” instead!

This is a temporary adjustment which will NOT be required anymore once Angular 9 is released!

**#### 75. GETTING ACCESS TO THE TEMPLATE & DOM WITH @VIEWCHILD( )**

In the last lecture we learned about local references. Now there is also another way of getting access to local references or to any element actually directly from within our typescript code. Right now we're passing the reference here when we serve and we call a method and that works fine. But sometimes you want to get access before we call the method and there is a nice old decorator we can use in typescript to get this access. Let's do the same server content so we're able to quickly duplicate:

Duplicate:

<input type=”text” class=”form-control” [(ngModel)]=”newServerContent’>

And comment out the second one

that input and comment outa old solution using two way binding and the new solution will not use two way binding( [(ngModel)]=”ngServerContent”) but instead this will now also have a local reference these server content input maybe.(#serverContentInput)

Now in the cockpit component I will also comment out the old new server content(newServerContent =’ ‘) here and I will add a new property which I'll name serverContentInput now server content input like this is just a property but we can add at view child in front of it. This decorator with parentheses and we need to import view child from at angular core. Now view child like this won't work. We need to pass an argument here and this argument actually is. Lecture or off to element how we want to select the element and select or now not like a CSX selector. But for example we can simply pass as a string the name of a local reference so server content input could now be passed as a string. If you don't want to pass a string but want to select a component you could simply pass to component type here. So in the cockpit we don't use any other components but in the app component we could pass cockpit component so to type like this not like a string to get access to the first occurrence of this cockpit component in the app component.

$ ViewChild(‘serverContentInput’ {static:true}) serverContentInput;

But here I'll switch this back to getting access to the local reference which probably is the use case you will see the most often. And with that we now get access to our server content input. Now what does this actually hold. Then let's quickly comment out (onAddServer( ) ) this codes that it doesn't break and let's log this server content input so our property here which is kind of set through view child. So with this if at all to comment this out down here(onAddBlueprint( ) ) so that we can run our app and if I now safe this and we go back to our application and I enter something here and I click ad server we see that this is of type element ref. So unlike the local reference we pass directly from the template why the method which was the element itself. This is of type element ref so we can set this type here:

$ @VieChild(‘serverContentInput’) serverContentInput: ElmentRef;

Element ref a reference to that element element ref needs to be imported from at the angular core it's

the angular type. So this element ref however has a useful property we can use and that steam native element property we can use server content loops. This server content input here( server:Content: this.newServerContent) and then native element to get access to the underlying element which now has a value because this will now be the input element.

$ serverContent: this.serverContentInput.nativeElement.value;

Now we can do the same of course for the on app blueprint method. And with this we get direct access to elements in our Dom in our template through at view child. And you should see that this now works again if we Andrews am content here and collect the buttons works perfectly.

Now without two way binding but with local references past two methods or local references fetched through view child. Now one important thing to watch out for is you should not change the element fruit is because of course you could use our server content input the native element and then set value to something right. You can't do this. And if we actually do this like this and I click ads or a blueprint you see something was now added here. I strongly recommend not doing this:

$ this.serverContentInput.nativeElement.value = ‘something’ // not recommended

You should not access to DRM like this angler offers you a better way of accessing the Dom and you will learn about this in the directives section. Generally you should use the other tools like string interpolation and property binding if you want to output something in the DOM and not directly mess with any element. You can get even though if you have gotten them through angular.



**#### 76. PROJECTING CONTENT INTO COMPONENTS WITH NG-CONTENT**

So we learned a lot throughout this section. We learned how to pass data around, how to get access to elements in our DOM, how to use local references, so our application now is much more dynamic, we got much more tools to interact between the different pieces we have in our app.

Now there is another way we can pass data around, the last way for now. In our

server-elementcomponent here,

$ *<strong \*ngIf=”element.type === ‘server’* “ style”. . .”>{{ element.content }}</strong>

$ *<em \*ngIf=”element.type === ‘blueprint’”>*

right now we check if we have a type server or a type blueprint server, right. Nothing wrong with that but sometimes, you have complex HTML code like this which you want to pass into a component from outside. So imagine you don't want to set this up inside the component here(server-element.component.html), I cut it, instead in the app component, you somehow want to add this here between the opening and closing element of your own component(<app-server-element>) and there, we simply would have to replace element with server-element and we should expect this to work, right.

So if we save this and let it reload and we add a new server, we don't get an error but the content is also not there and that is the default behavior. Everything you place between the opening and closing tag of your own component is lost by default, is simply removed from the DOM, Angular will not take care about it but you can change this. There is a special directive and it is a directive, even though it looks like a component but it doesn't have its own template ,a special directive you can add in the server-element components template here,

$ <div class=”panel-heading”>{{element.name}}</div>

$ <div class=”panel-body”>

$ <ng-content></ng-content>

$ </div>

in the place where I want to render the content, I can add ng-content, opening and closing again. It still is a directive, just using this element like selector and this serves as a hook you can place in your component to mark the place for Angular where it should add any content it finds between the opening and closing tag here ( the content inside the p tag that got moved to app.component.html) , so this content in this case. With this tiny addition, you should see that once you save this and let it reload, if we add this and we click add server, you now see the content is back.

It looks exactly the way it looked before but technically, something totally different works or happens here. Now we add this via the ng-content hook, we add it between the opening and closing tags and therefore it will be projected into your component, projected into your server element component. A nice feature, especially if you think about building re-usable widgets, like a tab widget where each tab will have a content which probably comes from some other source and which you don't want to pass through property binding which always would be an alternative but if it's more complex HTML code, property binding really is not the best solution because Angular will escape HTML tags there to prevent cross-site scripting attacks from happening and you could kind of work around that but really, ng-content then is how you want to display this and that is a great tool to have at your disposal.

**#### 77. UNDERSTANDING THE COMPONENT LIFECYCLE**

We're nearing the end of the section, there is one thing you might have recognized before where you

weren't sure what it does.

What's up with this ngOnInit method:

$ ngOnInit( ){

$ }

we have in new components created through the CLI, what's it doing? ngOnInit is a lifecycle hook and Angular supports a couple of lifecycle hooks, let's take a closer look.

If a new component is created in Angular and of course Angular is responsible for creating these components when it finds one of our selectors for example, it will instantiate a new version of that component and add it into the DOM. So once a new component is instantiated, Angular goes through a couple of different phases in this creation process and it will actually give us a chance to hook into these phases and execute some code.

We can hook into these phases by implementing some methods Angular will call if they are present.

The first phase, the first hook we can hook into is “ngOnChanges” ( called after a bound input property changes) and this may actually be executed multiple times, it's executed right at the start when a new component is created but thereafter, it's also always called whenever one of our bound input properties changes and with that, I mean properties decorated with @input, so whenever these properties received new values.

Now the second hook is “ngOnInit” ( called once the component is initialized), this method gets executed once the component has been initialized. This does not mean that we can see it, it has not been added to the DOM yet so to say, it has not been displayed yet but Angular finished the basic initialization, our properties can now be accessed and initialized for example, so the object was created you could say and if you're interested, ngOnInit will run after the constructor.

Then we have “ngDoCheck”( called during every change detection run) , that will also run multiple times, actually this method will be executed a lot because this will run whenever change detection runs.

Now change detection simply is the system by which Angular determines whether something changed on the template of a component or inside of a component I should say, so whether it needs to change something in the template. So whether some property value changed from 1 to 2 let's say and that property is output in the template, well of course Angular needs to re-render that part of the template and ngDoCheck is a hook executed on every check Angular makes. Now important, on every check, so not just if something changed, a lot of times ngDoCheck will run because you clicked some button which doesn't change anything but still it's an event and on events, Angular has to check if something changed because how else would it know? You don't tell it, right, so it has to check on certain triggering events like you clicked somewhere or a timer fired or an observable was resolved and on these occasions, it will check your code and ngDoCheck will be executed.

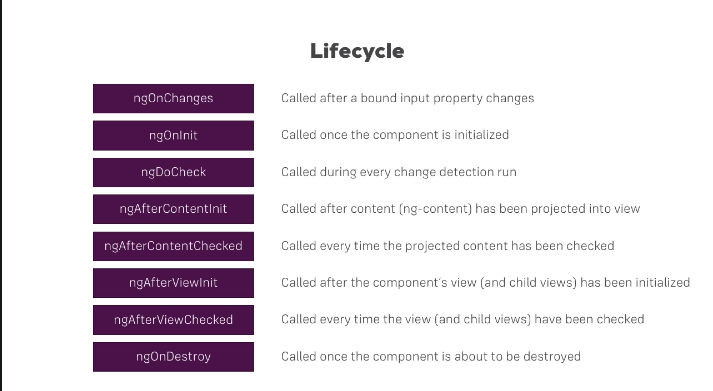
Now whilst this might sound very inefficient, Angular does this in a very efficient way, so change detection on Angular works pretty great and doesn't cost a lot of performance. ngDoCheck is a great method to use if you want to do something on every change detection cycle, like maybe manually inform Angular about some change it would not be able to detect otherwise, though that is a very advanced use case.

Well then, we reach “ngAfterContentInit” (called after content (ng-content) has been projected into view) , this is called whenever the content which is projected via ng content has been initialized. So not the view of the component itself but instead you could say the view of the parent component, especially the part which will get added to our component through ng-content. And ngAfterContentCheck is executed whenever change detection checked this content we're projecting into our component.

ngAfterViewInit( called after the component’s view ( and child views ) has been initialized ) is then reached once the view of our own component has been finished initializing, so once our view has been rendered you could say. And the same, ngAfterViewChecked, well that is called whenever our view has been checked, so once we are sure that either all changes which had to be done were displayed in the view or no changes were detected by Angular.

And finally, if you destroy a component, for example if you placed ngIf on it and this gets then set to

false and therefore it removes it from the DOM, “ngOnDestroy” ( called once the component is about to be destroyed ) is called and here's a great place to do some clean up work because this is called right before the object itself will be destroyed by Angular. These are the hooks, nice to see them in theory, let's see them in practice in the next lecture.





**#### 78. SEEING LIFECYCLE HOOKS IN ACTION**

So we learned about all these hooks, now let's see what they actually do. In the server element component, this is the component I will choose for demo purposes, we already get the constructor in ngOnInit.

So in the @constructor, I will simply log constructor called ( console.log(‘constructor called!’) )so that we can see when this happens in the console and I'll do the same for ngOnInit but of course here I'll say “ngOnInit” called( console.log(‘ngOnInit called!’) ) . Of course I will add the other hooks then too but let's start with these two and if we reload the application here, we see right at the top ( in the console ), constructor called “ngOnInit” called and for every new server we add, we see this again of course because a new instance of this component was created and therefore all these hooks are executed again because it's a brand new instance, doesn't have anything in common besides the class it was created of with the first instance of that component here.

So that is constructor in “ngOnInit”, now I told you that “ngOnChanges” would run first.

$ ngOnChanges( ){ }

So let's implement “ngOnChanges” and this would be enough, though it is a good practice to implement the onChanges interface( add in exports ) too and that is true for all hooks. Implementing the method alone will do the trick but it is a good practice to be very explicit about which interfaces your component users or which methods your components will have. So therefore implementing these interfaces is a really good practice, it's very clear then which methods may be called in your component and which methods you won't have.

Now here, we're getting a lot of imports which is why I will split this up and as you can already see, you have to import all these interfaces, onInit and onChanges. So “ngOnChanges” runs first as I told you,

so let's say “ngOnChanges(){ console.log(‘ngOnChanges called!’)” here and this hook here actually receives an argument. It's the only hook that does, here we receive a changes argument which is of type SimpleChanges which also needs to be imported from @Angular/core.

$ ngOnChanges( changes: SimpleChan ges){ console.log(‘ngOnChanges called!’

So I will simply log changes here( console.log(changes) inside ngOnChanges ), so that we can have a look at it and see what's inside. So let's save this and here( in the console) are the top again, we see constructor called “ngOnChanges” called as mentioned before, “ngOnInit” and then we see the “SimpleChanges”.

That's an object which has an element which is of type SimpleChange and element of course is our bound

property, here it's this property name, that's the name we see reflected here and then Angular simply gives us some information. What's the current value? Well that's the object with just a test, test server and of type server. Is this the first change? Yes it is, we just initialized it. And what's the previous value?

There is no value because we didn't didn't change this before.

Now to see that ngOnChanges actually gets fired more than once, I'll add something to this app just for demo purposes, I'll add a button, change first element to my app component template above this list of server elements, give it some bootstrap styling here but that's not really important. More important, give it a click listener, onChangeFirst( ) would be a name you could choose and then in the app.component.ts file, I went to change the name of the first element.

$ ngChnageFirst( ){

$

Now for this, I also need to change the way we pass this element to the server component. Right now, we receive the full element, the full object and I will leave this here for demo purposes,

(@Input(‘srvElement’) element:{type:string, name:string, content:string}; )

so that we can reference this code but now I only expect to get the name because the name is the only thing I'm outputting here anyways. (div class=”panel-heading”>{{element.name}}</div>

So I'm duplicating this and I will comment out the old code and now we only output the name because all the other element information was projected via ng-content into this element anyways, into this component. So we only get the name now, this name will simply be a string, we added input so that it can be bound from outside and we're outputting name here.

So in app component, when we loop through this, I will now actually bind the name too and I will bind to server element name and you wouldn't need to bind server element here

<app-server-element……………………………. [name]=”serverElement.name”

anymore because that is not usedbut I will leave it here for reference.

So this is now how we pass the data, how we passed the name. In the app.component.ts file here in $ onChangeFirst,(){

$ this.serverElement[0]

I will now access my server elements, the first element and therefore this will fail if we were to empty

this array and set the name to changed. So again, I did these changes so that we can really see when ngOnChanges gets fired and we couldn't use the element as an object because objects are the reference types, so we passed the object via input, therefore both properties in the server element component,

this one here and the object in the array of server elements here in the app component, they both point to the same place in memory and therefore if we change the name there, it will update in the child component, in the server element component but it will not trigger ngOnChanges because technically, the property we're binding to, we use here with @input, that didn't change the value of that didn't change it, it's still the same object in memory.

So it simply this difference between reference and primitive types and you should look this up if

that's all brand new to you, it's a Javascript concept, has nothing to do with TypeScript or Angular.

So that is why now I'm binding to a primitive with name which is just a string and therefore, ngOnChanges should get fired again if I click the change first element button,

let's do that and indeed, it does. Note that the constructor and ngOnInit weren't called but ngOnChanges was and here now we also see the previous value and the new value and therefore, this hook could be interesting to you if you want to react to any changes and then do something with the old values, store it

before it gets dumped or something like that, that's ngOnChanges.

We get a couple of our other hooks, so let's move on. Besides ngOnChanges and ngOnInit, we get do check

and as I said, this also needs to be imported from @angular/core. This will give us a method which is executed on every change detection run. So ngDoCheck is the method we need for this and here we can simply console log ngDoCheck called.

so let's see when this gets called. If this reloads, you see it gets called right at the beginning,

then one more time here but that is only because we're running in development mode, there Angular has one extra change detection cycle but it also gets called if I click this button as you can see and if I click this button(ngOnChange button which isn’t working for some odd reason) a couple of times here(add server button).

So ngDoCheck as I told you gets called whenever Angular checks for any changes and there are a couple

of triggers which trigger this method and event was called by clicking or a promise gave us back some data, so a lot of triggers for ngDoCheck. So that is not something where you want to run amazingly powerful code in because that would cost you a lot of performance but the fact that change detection runs as often as it does on its own is not a problem and this can be a great hook if you want to check, if you do need to change something manually because Angular didn't pick it up or something like that.

So the next hook is afterContentInit, this needs to be imported from @angular/core too, this interface

and with that we need to implement ngAfterContentInit. Now let's call this console log ngAfterContentInit called. If we do this here, you will see that here it is, there it is called after do check and it is called only

once because it doesn't get initialized again. Remember, content is the thing we projected into this through ng-content, so this here will be our content in the end.

Let's move on. The next hook then is and this is getting long here so I will split this up over multiple lines too, all these interfaces I'm implementing, so the next hook will be AfterContentChecked, like this. This also needs to be imported from @angular/core and it requires us to implement a method, ngAfterContentChecked and well now I'm just going to copy the code from ngAfterContentInit and replace init with checked. If we save this and have a look at it, you see here it is called and here it is called again after do check, which makes sense because it is called after each change detection cycle.

Well now I'm going to copy these two here(ngAfterContentInit and ngAfterContentChecked) and also I will add two new interfaces, the first one being AfterViewInit and then AfterViewChecked, both need to be imported from @angular/core too. Now this require me to have ngAfterViewInit as a method and ngAfterViewChecked and I will replace the text here in the console log so that we can see when these get called.

So if we log this, you see it happens after the content has been checked. Well and with that, we're almost

at the end, the remaining piece is ngOnDestroy and onDestroy is called, well right before this gets destroyed, this component gets destroyed. So let's add it here on the implements list, on the interface list I should say and make sure to add the import from @angular/core and then you need to implement ngOnDestroy and here I will say ngOnDestroy called.

Now to be able to actually see this being destroyed, I will add a new button, button destroy first component and give it a class of button and then btn-danger to make it red and here I'll add a click listener, onDestroyFirst. I will copy that, add my method here in just onDestroyFirst of course, just the name. We'll add it here to my TypeScript code and there I will something call server elements, the splice method splicing at the first element and then just one element therefore removing it. So with this, we should now see all the hooks called, view checked and so on and if I destroy the component, you see ngOnDestroy was called because we removed it from the array, therefore ngFor reran and didn't render this first element because it was removed, therefore it was removed from the DOM, hence this component hook was called.

So these are all the lifecycle hooks in action,

you rarely need to use all of them but some of them can be very helpful for your application to hook

into a specific piece, a specific face and run your code.



**#### 79. LIFECYCLE HOOKS AND TEMPLACE ACCESS**

Before concluding this module one more thing related to our hooks here. We learned about at Vue child which we can use here in the cockpit to get access to element from our while Dom from our template.

Let's do the same here in the server-element.component.HTML file where we have all these lifecycle

hooks.

Let's say I want to get access to this heading:

$ <div class=”panel-heading” #heading>{{name}}</div>

So I would simply place a local reference named heading on it therefore I'll add a new ad view child

property.Make sure to import view child from angular core. And with that added here all stored is in the header property which is of type element for F as we learned which also needs to be imported from angular core. And I want to select the heading local reference with a chest placed on this day of year.

So with that we get access to it and now we would expect that we can't use stairs before we reached

after view in it. So let's try it and energy on in it. Let me actually console log this header and there the native element. And now let's access something which is definitely a whale on a div and that is the text content if we do this and I save this.

Let's view the console. If I reload the application here you see this empty line here. This is where I would have expected text content can make this even clearer by outputting text content and then while rendering the text content if there were any there is no text content in here as you can see an empty space. If I copy the exact same code and placed as an after view in it here. So after the view should have initialized now you will see that here when it's called the second time after viewing it we see test server.

So that is the difference between Wilder points of time where these hooks run after view and it gives

you access to the Template elements. You can then access them and use their values and so on before does Hook has been reached. You can't do that you can't check the value of some element in your dorm because it hasn't been rendered yet.

**#### 80. @ContentChild( ) in ANGULAR 8**

In Angular 8, the @ContentChild( ) syntax which you’ll see in the next lecture needs to be changed slightly:

Instead of:

$ @ContentChild(‘contentParagraph’) paragraph: ElementREf;

Use:

$ @ContentChild(‘contentParagraph’, {static:true} ) paragraph: ElementRef;

The same change ( add { static: true } as a second argument ) needs to be applied to ALL usages of @ContentChild( ) if you use the selected element inside of “ngOnInIt( as we do it in the lectures )

If you DON’T use the selected element in “ngOnInit” set “static:false” instead.

**#### 81. GETTING ACCESS TO NG-CONTENT USING @CONTENTCHILD( )**

In the last lecture we learned that we have view child and how we can access it in our life cycle hooks

that there is one ever nice little addition in app component where we add our content which we then

project into our server element. Let's say we also want to place a local reference on this paragraph.(inside the app.component.html)

$ <p #contentParagraph>

So the content paragraph maybe. But now we want to use that in our server element component which is where this content will end up in the end.

Now in the app component which is where we currently placed us in the template we could use at The View child because it's in the template of the app component even though it will be kind of passed on to

the server element component later. Since we know that it will be there it would be nice if we could also access it from the server element component.

But at the view child won't work because it's not part of the view. It's part of the content which is why we also have separate hooks your content in it and view in it. Great thing is we don't only have at view child.

We also have at content child which also needs to be imported from @angular core. There's not also allows us to pass a selector so the reference name content paragraph would be the selector we chose here in the app component HDMI file and then just like with @view child we get we can store this in some property which will be of type element ref like this and now we can use it and just like

with view ref we of course can't access to value or anything before we reached content in it.

So that's something I just wanted to add at contentchild to get access to content which is stored in

another component but then passed on via energy content. Now let me locked is and I will log it here and on in it to demonstrate that this is not there. Here I will simply output text content of paragraph

ngOnInIt() { console.log(‘text content of paragraph: ‘ + this.paragraph.nativeElement.textContent)

whoops paragraph and access this paragraph native element text content and then I will copy this to

ngAFterContentInIt. Now once I save this reload the application you see that the first time we call it it's empty here( in the console it will look like “Text Content of paragraph: ) after on in it but there after ngAFterContentInIt( )after content in it or after this has been called once this hook has been called we see just a tests now it was initialized. That is the missing piece. Understanding how we can get access to elements in our dorm and how we can work with these lifecycle hooks.

**#### 82. WRAP UP**

So throughout the section, you learned a lot about components, we really had a deep dive on them. You learned how to use data binding with custom properties and events, how to pass data around, how to access elements in your DOM, how to use local references, how to really create a dynamic app and how to use lifecycle hooks to run some code at specific points of time.

With that, a lot of core features of Angular have been set now, have been taught, now it's time to of course practice them in the recipe book in the next module where we will enhance it as with our new knowledge and of course to play around with that knowledge in your own apps. So let's move onto the recipe book before we dive deeper into directives.

#### ASSINGMENT

Practicing Property & Event Binding and View Encapsulation

Time to practice the things you just learned! In this assignment, you’re going to bind to your own events and properties and also use view encapsulation to style everything just the way it should look like.

-1.Create three new components: GameControl, Odd and Event

-2.The “GameControl” component should have buttons to start and stop the game

-3.When starting the game, an event( holding a incrementing number) should get emitted each second (ref=setInterval( ))

-4.The even should be listenable from outside the component

-5.When stopping the game, no more events should get emitted ( ClearnInterval(ref) )

-6.A new Odd Component should get created for every odd number emitted, the same should happen for the Event Compoennt ( on even numbers )

-7.Simply output Odd – NUMBER or Even – NUMBER in the two components

-8.Style the element ( e.g. paragraph) holding your output text differently in both components

-Step 1:

-Create a project using ng new “basics-assingment-five-mysolution”

-install bootstrap using npm install –save bootstrap@3

-add the source into “angular.json” for the bootstrap to work

-then start ng serve with a port if 4200 is already in use ng serve –port 4300,4301,4400

-create components using CLI

-in the file app when you are in “src”

-type ng – g c “gamecontrol”

-type ng –g c “odd”

-type ng –g c “even”

-Step 2:

-Create two buttons

-<button class=”btn btn-primary”>START</button>

-<button class=”btn btn-warning”PAUSE</button>

-add the app-gamecontrol component into the app component in order to view the buttons

-Step 3

-now we need to add a click listener inside the gamecontrol by adding a click button and method onStartGame() – (click)=”onStartGame( )” inside the button tag

-in the gamecontrol. Ts file write the onStartGame( ) that increments

-set an interval and set it as property later access it and clear it later, under the exports

-set that interval, this.interval to setInterval and to 1000 milliseconds

-this.interval = setInterval( )

-inside the set interval write a function that gets executed on each tick

-create a new event emitter as “intervalFired”

-this eventEmitter should be imported to angular core

-this eventEmitter will hold a number as a value and set it to generic type <number>()

-now call that emitter inside the onstartgame method. By this.intervalFired.emit()

-create lastnumber equal to zero and call that lastnumber inside the emit

-add this lastnumber by 1 and then keep incremementing it

-Step4

-to make this listenable from outside

-so we need to add @output( ) infront of the intervalFired under the export class

-and output needs to imported in the angular core

-in the app.component inside the app-gamecontrol we can listen to the “intervalfired”

-<app-gamecontrol><(intervalFired=”onIntervalFired()”)</app-gamecontrol>

-isnide the app.component write the method onIntervalFired( )

-and console.log the last number

-add $event inside the onIntervalFired($event), $event gives the access to the data which is the number in this case.

-in the app component we know we are going to receive the number onIntervalFired(number)

-when you click the start game button in the console you should see the number incrementing every second

-Step 5

-now we need to get the pause to work

-inside the button tag create a function click event and assign it a onPauseGame()

-write the onPauseGame function isndide the gamecontrol component.

-inside write the clearInterval and pass the reference this.interval

-when you reload the application, and click the start game, the numbers increase by a second and when you click the pause game the numbers will stop incrementing

-Step 6:

-inside the odd.component.html we will create a p tag with “odd – {{number}} </p> string interpolation

-number is a property which doesn’t exist right now so inside the odd.component.ts we will create one

-will create a type number with property number under export class

-this is not initialized and this should be passed from outside

-so we need to add @Input infront of the number @Input( ) number: number;

-import this at angular core

-now we are able to pass the data from outside into this component

-inside the app.component.html we need create a component which will be the app-odd component

-inside the app component we need to create an array of empty odd numbers to store

-you can add type by adding oddNumbers: number [] = []

-duplicate this for eve numbers

-now inside the OnInterfired we want to chedk if the fire number is odd or even

-write and if and else statement to see if the “firednumber” is odd or even and then push the firednumber

-now in the app component write \*ngFor and create a variable let oddNumber of oddNumbers, looping through the arrays

-now bind the “number” to oddNumber by: [number]=”oddNumber”

-Step7

-for even copy the odd template and make it even

-we need to add @input infront of the number @input number: number

-import this at angular core

-now we are able to pass the data from outside into this component

-inside the app.component.html we need to create a component which will be the app-even component

-inside the app.coponent we need to create an array of empty even numbers to store

-you can add type by adding evenNumbers: numbers[] = [ ];

-you will need to loop through \*ngFor for evenNumber of evenNumbers

**##SECTION 6: COURSE PROJECT – COMPONENTS & DATABINDING**

**#### 84. INTRODUCTION**

So now that you learned more about components and data binding and especially how you can communicate between your components and how lifecycle hooks work and so on,

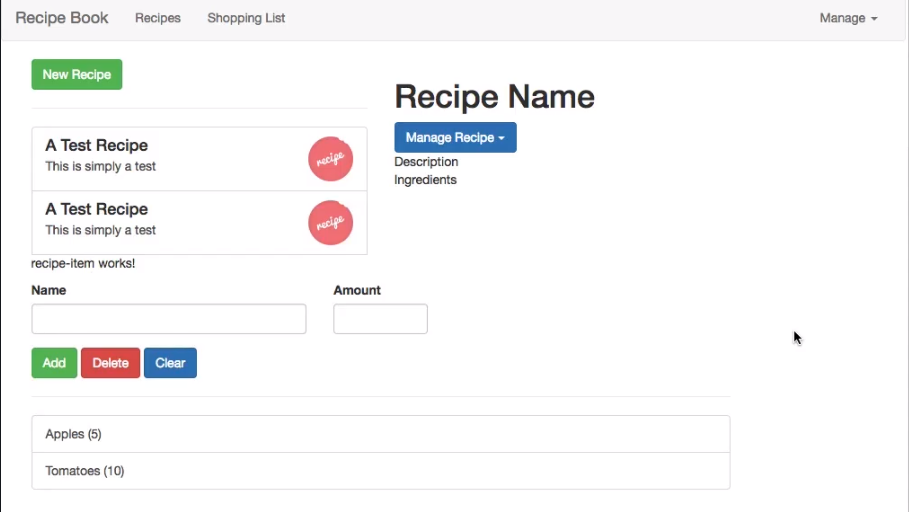
you gained a lot of new knowledge which is super important for most projects you are going to build, including our course project.

So in this section, what we're going to do is we're going to fine tune this, we're going to make sure

that we can navigate between recipes and shopping lists, though our solution we'll add right now will

not be the final one, we will improve this even more later in the course and we will make sure that we

can select recipes here and load them in the detail section and generally, that we have a working communication between our components. So let's dive into this.



**#### 85. ADDING NAVIGATION WITH EVENT BINDING WITH NGIF \*\*\*\* VERY IMPORTANT \*\*\*\***

The thing with which I want to start in this course is to hook up navigation, to either load recipes or

the shopping list. Now we will later learn a way to do this in the way it is meant to be done but the solution we will agree is now also is very creative and not bad.

I will use ngIf to only load one of the two sections at a time, so somehow in my app component probably because that is where I include both, here:

$ <app-header></app-header>

$ <div class=”container”>

$ <div class=”row”>

$ <div class=”col-md-12”>

$ <app-recipes></app-recipes>

$ <app-shopping-list></app-shopping-list>

$ </div></div></div>

I will need to manage which one should be displayed and I should determine which one is displayed in the header component, so I should kind of pass the information which link was clicked there to my app component, so that I can then switch a property there to only display one of these two components.

$ <app-recipes></app-recipes>

$ <app-shopping-list></app-shopping-list>

I will admit that this task involves multiple steps, It doesn't cover anything you haven't learnt thus far though, so definitely take this as a challenge to implement such a navigation on your own, emitting some event in the header which allows us to determine which component should be displayed here

in the app component. Now's your chance to pause the video, we'll do it together thereafter. So were you successful?

Here is my solution, how we could get this to work the way we want it to work. I'll start in the header component template, here:

$ <li><a href=”#”>Recipes</a></li>

$ <li><a href=”#”>Shopping List</a></li>

we got these two links and I want to attach some click listeners to these links. Here I will say onSelect and I will pass a string here( (click)=”onSelect(‘recipe’) ), recipe maybe, we can of course pass our own data to the methods we execute here and here, I will select another one or add another click listener and I will say onSelect shopping list.

Now you could of course also work with numbers there or fire two totally different methods.

So that is my onSelect method, now in the header component, I of course need to implement onSelect and we get an argument here,we pass it here right, we get a string on what to select. So maybe let's name it feature and that will be a string (onSelect(feature: string), so let's assign the type string here by adding a colon and string thereafter and now as I said, I want to emit my own event.

Hey, we learned how to do this, first of all I'll need to add a new property which I'll name “featureselected”, you can of course choose another name if you want. This should be a new event emitter and now remember, event emitter is imported from @angular/core, that's a generic type and we will pass string here, the feature which was selected in the end and we have to add parentheses to instantiate this, so to create an object based on the event emitter class. That is our feature selected event, now I want to use this property(this.featuredSeleted) which now holds this event emitter as a value to emit an event( emit() ) whenever we click one of the buttons and I will then emit feature.

so this string we receive from these method calls here in our template. Well with that in place, the

missing piece is to enable this event to be listened to from outside of this component.

So we have to attach something to this property to make it listenable from the parent component,

the app component in this case, that something is the @output decorator, don't forget the parentheses

which also needs to be imported from @angular/core, @output enables us to use this event or

to listen to it from the parent component.

So in the app component which is the parent component which adds app header here, I will listen

to the “featureselected” event and then here, I will execute onNavigate let's say ( (featureSelected)=”onNavigate() )and of course, I'm interested in the data I'm receiving from feature selected, so I can get access to it with $event, this protected name we could use in the template to get access to any data an event passes to us.

Now just because this might be confusing, if you don't specify $event here but “other-data”

like we did here right(\* (click)=”onSelect(‘recipe’)” ), we kind of overrode $event, that doesn't actually override it, you simply determine which data you want to pass to your own method and $event may also be used as a second argument here(‘recipe’, $event) , $event will always refer to the event data anywhere between these quotation marks here( “onSelect(‘recipe’, $event) )” ), so no matter which place it has in the method argument list here.

So I just wanted to make sure that this doesn't lead to any confusion. So let's turn this back to $event to get access to the data we actually pass in our own event and now in the app.component.ts file, I will add onNavigate and here, I will receive the where should we navigate to information. So you may choose any name you want, I'll go with feature again, this is the feature we want to navigate to and now we should make sure that we actually store this feature.

So maybe here, we should say “loadedfeature” because we store which feature should be displayed

and initially, that might be our recipes and maybe to make sure that we always use the same string, let's

check it here, recipe is the name here, recipe feature, so let's go with recipe here too and then in onNavigate, I will set this loaded feature equal to the feature we receive here as an argument. And you could even omit this method and directly set loaded feature equal $event here but I will go for the longer approach to be very explicit about what's happening here.

So with that, we're getting this loaded feature, now we can use ngIf to determine which of the two components should be displayed. App-recipes should be displayed if loaded feature equals recipe and shopping list should be displayed whenever loaded feature let's say does not equal recipe, so that will also catch cases where it somehow is set to something else. So with that, we're using ngIf, now let's see if that works.

We only see the recipe now and if I click on a shopping list, we see the shopping list and the other way around. So now navigation is kind of working, again we will later improve this but for now, that is a great solution. Now let's move on to making the list items here real list items living in their own component.

**#### 86. PASSING RECIPE DATA WITH PROPERTY BINDING**

So we add navigation in the last section, now I want to work on this list of recipe. Right now if we have a look at its component here(recipe-list), we see that the code for a single list item is this one here.

 <a

            href=""

            class="list-group-item clearfix"

            \*ngFor="let recipe of recipes">

            <div class="pull-left">

                <h4 class="list-group-item-heading">{{ recipe.name }}</h4>

                <p class="list-group-item-text">{{ recipe.description }}</p>

            </div>

            <span class="pull-right">

                <img

                [src]="recipe.imagePath"

                alt="{{ recipe.name }}"

                class="img-responsive"

                style="max-height: 50px;">

            </span>

            </a>

We do have a separate recipe item component but right now it doesn't hold any template because we had to outsource this code and put it into this recipe list component temporarily because otherwise, we would not have been able to display our recipe data.

Now we know how to pass data to a component though, so we can cut all that code here(the above code) and put it intothe “recipe-item” component template, now the goal of course is to get this ngFor loop here and remove it from inside the item and move it back on the item here and then pass down the individual item, the individual recipe of each iteration to that component so that we can still output recipe name and so on here.

So that is your chance to do this on your own again, pause the video and try to loop through the component itself, so get ngFor out of this component and then pass down the selected recipe or the recipe of the current loop iteration, in a few seconds you'll see my solution.

So were you successful? First of all, I'll cut the ngFor loop here( from the anchor tag inside recipe-item) and I'll go back to the recipe list component and add it to the app recipe item selector, so that now the whole component is replicated for each recipe.

Now the issue of course is that we don't have the recipe which we access inside the recipe item component anymore. So we should probably add a recipe here( inside recipe-item component, under export class, add recipe), the recipe of that single recipe item component. It will be of type recipe of course, so we have to import this from our own recipe model, going up two folders and then it's there, right in the folder we then moved to and initially this will not be defined, there won't be a value assigned to it because I want to get this recipe from outside.

So to get a recipe from outside or any data from outside, we need to add something, a decorator, @input with parentheses which needs to be imported from @angular/core. @input allows us to bind this component property from outside and of course I want to bind it from my recipe-list component here, so in this list where I replicate the single recipe item a couple of times.

Now here we can now bind to our recipe ([recipe]=”, referring to the property in the single recipe item and bind the recipe of the current loop iteration. Now I'm not happy with the naming here because that can be very confusing, we're having recipe three times here, so to make super clear what each name here refers to, I'll rename the recipe here in our loop to “recipeEl” (recipe-list component) for element and that also has to be replaced here then because we're passing the single recipe element as a value, therefore on the right side of the equal sign, to the bindable recipe property.

And keep in mind, the recipe here([recipe]=”recipeEl”) between the square brackets simply is the recipe property in the component we're placing this on. So in the recipe item component, this recipe here, that is what that refers to and actually to make this super super clear, I can split this horizontally to you have the receiving component, the recipe-item component and the sending one, here template of the sending one, the recipe list component above each other, so that is what we're binding to here.

So let me close this. With this if we save that, we should see the same as before which makes sense,

it's a good thing that we see this, it means it didn't break and it makes sense because now we're successfully passing down the data which keep in mind is stored in the recipe list component here in this array, we're successfully passing this down to the recipe item component. Now let's work on a feature that once we click an item here( recipe list with image next to it) , we actually select it and then we load it in the recipe detail component. So that is what we'll work on next

**#### 87. PASSING DATA WITH EVENT AND PROPERTY BINDING COMBINED**

So with all these features added, let's make sure that we can click on a single item here and load it in

the recipe details section. Again this is the challenge for you, it only uses tools we learned thus far,

you somehow need to emit an event on the single recipe item component, that it was clicked and then get this event somehow to the recipes component which then also holds the recipe detail to pass down the event data, which recipe was selected to that component. Now you're going to have to work with property and event binding for this, again only things you learned thus far and if you're getting stuck or something like this, of course we will solve it together in a couple of seconds, here's your chance to pause.

So were you successful? Now the approach we're going to do now, we're going to follow now is only a temporary approach. We'll later learn how to do this more elegantly and easier but it's also a great approach because it uses the things we learned thus far and it gives us an impression on when a different approach might be useful. So what's our goal? Here in the single recipe item component, here we got this link and we want to make sure that whenever this link( anchor tag inside recipe-item component) is clicked, we emit an event informing our parent component that this recipe was selected.

So for this to work, I'll add a click listener to this link(inside the anchor tag) and I will say onSelected( ) here (inside recipe-item) , of course you can choose any method name you want.

I'll implement onSelected( ) here( inside recipe-item component) and there I want to emit my own event, so I will name it recipe selected and this will use the event emitter as you'll learn, again make sure to import it from @angular/core. The event emitter here won't pass any information, so I'll set the type to “void” because it will contain no information and we have to add @output so that we can listen to this event from outside, @output also needs to be imported from @angular/core.

In onSelected( ), I then want to trigger this.recipeSelected and call emit ( this.recipeSelected.emit( ) ), not error, emit. Again we don't pass any error because here we specify that we won't, we don't need to pass any. You could argue that you want to pass the recipe on which this was emitted but you don't need to do this because what is the parent component listening to that? That of course is our recipe list component, that is where we use the single recipe item.

Now if we implement our own event here(inside recipe-list) and listen to recipe selected, well we already got information of which recipe this will be because we have it here from the ngFor loop.

Keep in mind, this will replicate this element, so for each element, it's absolutely clear which element this is. In recipe selected, we again need to emit an event though because we're not there, where we want to be, we have to go up one level because we want to reach our recipes component, we're just the one implementing the recipe list.

This is probably where you see that as a kind of complex to simply inform another component that we

clicked something. For now it's the only method we have and it's a great practice but no worries, we will improve this in the future. So in recipe list, we need to emit another event, so here I'll execute onRecipeSelected( ), of course choose any name you want and that method of course has to be implemented here. Now here, we need to get some extra information, which recipe was selected and therefore, I will pass the recipe element here( (recipeSelected)=”onRecipeSelected(recipeEl)” ), so that is the recipe which we did select.

Now with that we have that information, we know here(receipe.component.ts) we will get the selected recipe which will be of type recipe( onRecipeSelected(recipe: Recipe) ), we are already importing the model therefore we can easily use the type and then we, as I mentioned, want to emit another event. So here we need to find a fitting name, before we used recipe selected, that would be a fitting name here too but to make it super clear that this is a different event because custom events don't propagate up so we can't listen to the event of a child of a child,

I want to choose a different name, “recipeWasSelected “maybe, which also uses the event emitter, so make sure to import it from @angular/core and here I will pass my recipe as a type. Here I will pass the recipe which was selected because that is the information the recipes component will need in

the end. ( recipeWasSelected = new EventEmitter<Recipe>( ); )

Now as before, we need to add @output here and import this from @angular/core, so that we can listen

to this event from outside and then here in onRecipeSelected( ), we can use recipeWasSelected emitted

and pass the recipe as data( recipeWasSelected.emit(recipe) ) Now with that, we can go to our recipe component, our main component for this feature area and on the list(<app-recipe-list>), we can now listen to recipeWasSelected, the custom event we created a second ago and here, I now want to store this selected recipe.

So in my TypeScript file(recipes.component.ts), I will create a selectedRecipe property which will be of type “recipe”, make sure to import recipe from your model file then and initially, this is undefined because I'm not assigning a value. I will assign a value here once this event here occurs, so then I will have set selected recipe equal to $event and of course you could also put this into a method but since this is all I want to do, I can put it into the template here. Generally though, you should try to put as little logic as possible into your template.

So here, I'm referring to the selected recipe property of my recipes component and $event as you learned refers to the data we passed with an event, now we can pass this information to the recipe detail component. First of all though, I will add ngIf to this detail component(<app-recipe-detail>) to determine if we want to display it because I only want to display it if selected recipe is set.

So here, I will check if selected recipe and if it is undefined, that will resolve to false and if it is not set, I want to say a dummy text, so I will use ng-template for this to define a paragraph where I say please select a recipe and ng-template will receive a local reference of info text ( <ng-template #infoText> and now with that placed on <ng-template>, we can use this ngIf else syntax where I add a semi-colon and then else info text, so that else the content of this template should be displayed

( \*ngIf=”selectedRecipe; else infoText” ). If we save this, we see please select the recipe here, so that it's working as it should, now if we select a recipe by clicking here, we see the recipe detail.

Now of course we don't see the detail data because we're right now not passing this recipe down to the component but at least our if check is working and we're passing the recipe, this event successfully.

Now as a final step, let's pass the selected recipe to the app recipe detail. For that we have to go to that component and add a property for this, here I'm going to name this recipe again which will be of type recipe and make sure to import this from your recipe.model file.

It's not set initially but I will add @input so that we may set it from outside, input of course as always has to be imported from @angular/core and with that, we can now go back to the recipes component and bind to that recipe and of course we want to bind our selected recipe and keep in mind, this whole component will only be rendered, only be attached to the DOM if this condition is true, so if selected recipe is not undefined. Therefore once we pass this down to the recipe detail component, we know that we can safely output the recipe name here( inside recipe detail component, change the h1 tag to {{recipe.name}} ), so recipe name, keep in mind recipe is the property name in this component, the one to which we're binding to from outside and this will never fail because recipe will always be set otherwise this component wouldn't be rendered due to ngIf.

Here( <img> tag inside recipe-detail component) I also want to use property binding to display my image, so recipe imagePath then and here, I'll use string interpolation to display the name ( alt=”{{recipe.name}} )if we can't load the image for some reason. The buttons are fine

but below in the description , I will set the recipe description.( {{recipe.description }} )

Now with this, if we save that and we click on the recipe, we see the recipe here. Now the image is a little bit too big, so maybe on this image, let's simply add some styling to it, an inline style where we set the maximum height to 300 pixels maybe, like that. Now with that, that looks better. So now we got this working, looks like nothing changed because we get the same text here, so let's quickly go to our recipe list component and name this another test recipe so that we can actually see if we correctly switch

and we do. So that is working, the missing piece now is the shopping list where I want to make these input fields here work, at least in a basic way.

**#### 88. MAKE SURE YOU HAVE FORMS MODULE ADDED**

One quick note: In case you’re hitting an error in the next lecture make sure you have “FormsModule” added to your imports[ ] in the AppModule. Also look at this threat: <https://www.udemy.com/the-complete-guide-to-angular-2/learn/v4/questions/4924644>

**#### 89. ALLOWING THE USER TO ADD INGREDIENTS TO THE SHOPPING LIST**

We added a lot of functionality in this core section. Our recipe book is really taking shape. Now of course there are also some features we haven't added yet which we will add later. For now I want to make this shopping list input fields here are working so that we can add new items to our shopping list.

We will overwrite this later when we learned about forums but it's a great exercise to work with local

references and at view child. For now so what I want to do is I want to work on this shopping-edit- component here. Here we got a couple of input fields and we get the Add button which submits our form later delete and clear.

Now I want to allow the user to add new items. So what I'll do is I will restructure both inputs to use multiple lines simply to make it easy to read and understand. And on the first input for the “name” I will add a local reference named name( #nameInput ) input on the second one I'll add one named amount input.

Now with that setup your challenge would be to get this Add button to kind of use these local references

preferably by either passing them as an argument or more advanced by selecting them with at view child

so that once you click the button you use the values of these inputs to create a new ingredient and

add it to your array of ingredients.

Here is your chance to pause a value and try this on your own. Then we'll do this together so where are you successful.

Let's do it together. We already have our local references here. So all I have to do for now is add a (click) listener and a name onAddItem and I could pass the local references or even already local reference store to value here as arguments I will use to add view child approach though.

So in the shopping list added component all add to on at item method and then at the top I'll add two

properties the name input reference ( nameInputRef ) and that will be of type elementref.

As we learned. Make sure to import this from @angular core. And it will get a value by using @Viewchild which also needs to be imported from @angular core. And here we pass the name of the local reference so name input for this first input field.

$ @ViewChild(‘input’, {static:false}) nameInputRef: ElementRef;

Well now I'm going to duplicate this and simply renamed as to amount input and amount input ref. That's the local reference we put on the second input here and with these two inputs added in on ADD item I now want to emit a new event where I pass this data to the parent component which is the shopping list component which manages my array of ingredients.

So we learned how to emit our own events so quickly go through that again. Here I'll name it ingredient added which will be a new event emitter ( ingredientAdded = new EventEmitter ) . Of course you could choose another name for your event. I will pass an object here as an argument which will hold the name which will be a string and the amount which will be a number({name: string, amount:number}). And that here you're also close to parentheses.

That here is simply just a type definition. So that type will be a javascript object with two properties name and amount where the first property name is a string and the second one is a number.

So that's not a value. That's just the type definition.

$ ingredientAdded = new EventEmitter<{name: string, amount: number}>( )

Now of course for this exact type definition we get our ingredient model so we can replace live this with just ingredient.

$ ingredientAdded = new EventEmitter<Ingredient>( )

Make sure to import ingredient from the shared folder and then the ingredient thought model file though.

Well and with this and onAdditem we can create a new ingredient here (inside onAddItem( ) )with the new keyword and then ingredient and we simply passed a name input ref native element value here and that will simply be the value of the input element and the same for the amount.

$ const newIngredient = new Ingredient( );

But since this is really long Ultra Q new temporary constants so to say or constants to make this code

a bit shorter.

So the ingredient name will be retrieved from the name input reference from the native element which

is the input element and then from the value field and the same for the ingredient amount here.

I will simply get this from the amount input ref and with these two constants here I will simply pass

IngName and ingAmount here.

So I will create my new ingredient with that data.

$ const newIngredient = new Ingredient(ingName, ingAmount);

Important the constants here are used or I use constant instead of let because we're not going to change

these variables but they won't be variables anymore than as well we only assign these values one time and you should declare something as a constant. If you don't plan on changing it so with this I can now admit my own event here and past in new ingredient data.

Well and with this all we have to do is we have to add at output in front of our own event here so that

we can listen to it from outside. Make sure to import output from angular core and now in our shopping list component which implements your shopping added component.

We can listen to our ingredient added event here and there. I want to execute on ingredient added and we will get some data which is always stored in dollars an event which I want to pass to my on ingredient added method. Let's implement this method here. So on ingredient added I know I will receive an ingredient here of type ingredient that is what our event emits the data we set up there and now we can reach out to our ingredients array and push a new ingredient.

The one we got here is data on it. And with that we should be able to indeed add some whips here add some bread to it. One click add and you see it on the list. So we got a half way working shopping list.

Of course the lead and clear is not working we also can select items here. That is something I'll work on later once we learned more about forms which give us some nice tools to manage them.

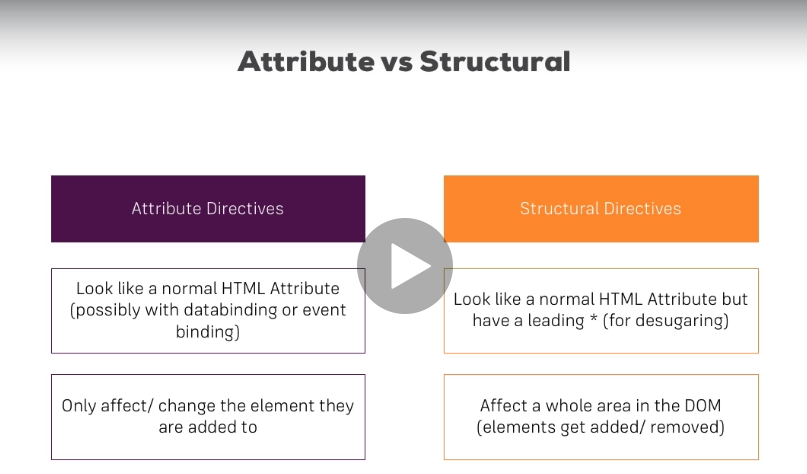
But we greatly enhanced our app in this section. We added some kind of working navigation. We are able to select a recipe here and we are able to add new items to the shopping list already. Even though some of these behaviors will get overwritten later in the course it still is a great start. Now let's move on and learn more about directives so that next time we come back to this project we can make our dropdown buttons here work. See you there.

**##SECTION 7: DIRECTIVES DEEP DIVE**

**#### 90. INTRODUCTION TO MODULE**

Time to take another look at directives and not only repeat quickly what we've learned thus far but also

learn how to build our own directives and really dive into the difference between attribute and structural directives and what this star means on the structural directives.Well, I already said it right, we get attribute and structural directives. Attribute directives are called like this because they sit on elements, just like attributes and structural directives basically do the same but they also change the structure of the DOM around this element. If you have ngIf on a paragraph, if that condition is false, this paragraph is removed from the DOM, so the overall view container is affected, whereas on the attribute directive, you never destroy an element from the DOM, you only change properties of that element, for example the backgroundColor.



So that is why we have attribute directives only affecting the element they sit on and structural directives which also affect the whole DOM or the view container, the area around the element which they were placed on. That is the difference, now with that let's briefly have a look at the directives we already know and how to use them before we then dive into building our own ones and learn more about structural directives.

**#### 91. RECAP \*ngFor AND \*ngIF**

**( basics assignment six was created for this )**

I prepared a little project which you as always can find attached to this lecture here

and it's a super simple project. We get a button where we can toggle to only see odd numbers but right now, we don't see any numbers because whilst I do have an array of numbers here in my app.component.ts file, I'm not outputting it here.

However I did set up a list ( li ) .So with this, why don't we go ahead and simply add \*ngFor with a star

since it is a structural directive to this list item( class=”list-group-item) and then we could simply say “**let number of numbers**”, we already learned this syntax and we could also extract the index if we wanted but we don't need that here.

So that is our single number and that of course allows it to now output it here ( {{ number }} ) . We can output it anywhere inside this element we placed ngFor on, so on any nested elements too. With that, we're outputting the number and now we see 1, 2, 3, 4, 5, the button still doesn't do anything though.

The reason for this is that right now we do something upon clicking, we set onlyOdd, which is this property, to what it currently is not, so if it was false, we set it to true and the other way around but we're not using this information. Let's use it.

Now we could try placing ngIf on the same element we already have ngFor on and check if number modulo, so the percent sign too, if that is equal to zero,( \*ngIf =”number % 2 == 0” ) which would mean now we only display even elements, right?

If we do this we get an error because we can't have more than one structural directive on the same element, that just doesn't work but we don't need that, we can change it differently, we can simply make a tiny change to our main app.component here and implement odd numbers which is 1, 3 and 5

and then of course even numbers and that is just a demo app for directives, right?

So that might not be the best solution for your app where you really need this behavior, later in the

pipe section, we will learn how to dynamically filter an array for example. So here, even would be 2 and 4. With that, I'm going to comment out the first numbers array and instead here, I'll loop through all my odd numbers and output odd here ( ngFor = let odd off oddNumbers ) and of course we can do the same for the even numbers now, so even of even numbers.

And now to display only one of the two, we can simply wrap them in let's say a div, so let's put the list item inside of the div and on the div, we could place ngIf and now the first one should only be displayed if onlyOdd is true. Now let's do the same for the second one here, let's place them in here and we place ngIf on that too where we check if not onlyOdd and maybe also replace even here. We now see we can toggle between the two lists, so this is now working with ngFor and ngIf. Nothing new, something we already learned. So these are the two structural directives we learned about and how we may use them and that you can't combine them on the same element. Before we dive deeper into this, let's also have a look at the two attribute directives we learned thus far, two of the most prominent ones.



**#### 92. RECAP \*ngClass AND \*ngStyle**

**( basics assignment six was created for this )**

There are of course many attribute directives and throughout this course, we will see different ones,

two we already learned about are ngClass and ngStyle. Now just to quickly refresh how they work,

let's add a class to our CSS file here, let's maybe name it odd ( .odd { } ) and here I simply want to set the color to red. (.odd{color:red} )

Now in my template, I want to assign this class if it is odd. So here of course, I could always assign the class since we already made sure that this list item list is only displayed if onlyOdd is checked. Still, we could also use ngClass here, like this and make sure that we assign the odd CSS class, therefore we define it here as a property name, only if the number we currently look at divided by two with the modulo sign equals something else than zero, so not equal to zero.([ngClass=”{odd: odd % 2 !==0}] )

That means we have a remainder for this division which means it's odd. Now we can copy this to the even list and again, I know that this will never be odd here( inside the even list group item) because we already make sure that this is only rendered if we are not displaying odd items, we don't have any odd items in the even numbers array. So just to show how this works, of course we have to check even here, now you see even numbers are black, odd numbers are red.

Again, not super surprising but just a brief refresher on how ngClass worked and important, I mentioned it before, the square brackets here indicate that we are binding to some property on our ngClass directive.

Now ngStyle was the other famous directive we had a look at thus far, ngStyle allows us to also pass an object to some property which is also named ngStyle on the same

Directive and here, we could simply set the backgroundColor and we could set it to red like this.

Now it would always be red but of course, we can also set a dynamic value here, so we could simply repeat our check from up here if it's an odd number, in which case I want to give this let's say a backgroundColor of yellow, otherwise it's transparent. ([ngStyle]=”{backgroundColor: odd % 2 !==0 ? ‘yellow’ : ‘transparent’}

Now I'll quickly copy this over to our other list, here of course I check if even is basically even and guess what, I think it will be. And now the even number is, still no styling at all and the odd numbers are really ugly now to be honest but that means our directives here are working. Now let's build our own ones because that is something we can do too, let's start with our own attributes directives.

**#### 93. RECAP \*ngClass AND \*ngStyle**

**( basics assignment six was created for this )**

So we already use attribute directives like ngClass and ngStyle, it's even easier to understand what to do once we've built our own one. So I want to build a directive which simply highlights an element I hover over, let's say. Of course I could achieve the same with normal CSS styles but this is just a good demo.

So I will create a new folder and I will name it “basic-highlight”, like this because in there I create

a new file, “basic-highlight.directive.ts”, so before we always had component files, now this is a new element, it's not a component, it will be a directive hence the file name. Now in this file, I'll **“export”** a class which I'll name **“BasicHighlightDirective”**, again to be descriptive about what this class is and to make it a directive, just like with a component where we added @component, here we have to add **@directive** and directive needs to be imported from **@angular/core**.

so make sure to add this “import” and we need to pass an object to this decorator to configure this directive. Now how do we configure a directive? The one thing our directive absolutely needs is a **“selector”** because remember, we do place directives in our template to attach them to elements, so we need to have some way to give Angular that instruction and that is the selector. Here that should also be a unique selector.

So typically, here you use a camel case notation to give it” appHighlight” as a selector or here maybe

**“appBasicHighlight”,** like this. Now that would select it by element. Now I want to have this attribute style, so I'm going to wrap this in square brackets( **[appBasicHighlight ]** ) which means this will now be recognized whenever I add appBasicHighlight without square brackets to an element. Now to show this, we need to do something which gives us some visual clue that this is working.

So the basic thing is, the most basic use case is we change let's say the background colour of the element

where we attach this directive. For this we need to get access to the element the directive sits on and

the cool thing is Angular gives us this access. We can inject the element the directive sits on into this directive and injection is something we'll take a closer look at in the next course module, in the big next course module which is about services, I can say that much.

It basically is an easy way to get access to some other classes without having them to instantiate on our own. We do inject by adding the “constructor( )” which every TypeScript class has, we don't need to write anything in the constructor body for now but here on the list of arguments, we list a couple of arguments we want to get whenever an instance of this class here is created and of course Angular is responsible for creating these instances.

So therefore if we tell it to please give us a specific type of argument, this is what injection is, Angular will try to create this thing we need and give it to us. Well this thing we need in this case simply is a reference to the element the directive was placed on. So an **“elementRef”** and this name is totally up to you but the “type” is important, the type has to be **“EleventRef”,** you might recognize this from @ViewChild, there it also was a reference to some element, here it is too. Now to be able to use this data in our class here, everywhere in the class, we can use a TypeScript shortcut of adding **“private”** in front of it which will make this both a property of this class, so property element ref and automatically assign this value, so this instance we're getting to this property. ( constructor(private elementRef: ElementRef) )

Now with that, we got access to the element, now we can use it here in our constructor for example, access the native element and then do something ( elementRef.nativeElement ) with it. Though a better place than the constructor is onInit and just like the component, the directive also has the “ngOnInit” lifecycle hook.

So here, I can therefore add ngOnInit and in there, we could access elementref, that's the shortcut `which automatically gave me this property, access the native element and there access the style and maybe the background color and set this equal to green so that we can see something changed.

**$ ngOnInIt( ){**

**$ this.elementRef.nativeElement.style.backgroundColor = ‘green’**

**$ }**

So what we're doing here is we're getting access to the element the directive was placed on, getting access to that exact element and then we're overriding the style of this element, now let's use this directive. To use it, we have to do two things.

First of all like for a component, we have to inform Angular that we have a new directive. Just like with components, it doesn't scan all our files, so it doesn't know. So we have to go to app module and here in **“declarations”,** we have to add our **BasicHighlightDirective** and also add the **import** pointing to the basic-highlight folder and there to the BasicHighlightDirective file.

**$ @NgModule({declarations:[AppComponent, BasicHighlightDirective], ………..})**

Now with this, we can use the directive in our app since we informed Angular, now let's use it in our app.component.html file and here below all our lists( li ), I will simply add a new paragraph, “style me with basic directive”, something like that and now to this paragraph, I can add “appBasicHighlight,” my own selector.

**$ <p appBasicHighlight >Style me with basic directive! </p>**

We don't need to set any value and importantly, we don't use square brackets because as I already emphasized, the directive name is just a selector we set up here and the square brackets here are not part of that name, it's part of this selector style telling Angular please select it as an attribute on an element and that's just how we add it here, like an attribute of the paragraph. Now with that if we save this, we indeed see a green paragraph below our list because our directive is doing its job here, it's styling it, so that's our first basic attribute directive. We can enhance this though.



**#### 94. USING THE RENDERER TO BUILD BETTER ATTRIBUTE DIRECTIVE**

In the last lecture, we created our first basic directive and it's doing its job, we learned how to create a selector, how to use that selector, like an attribute in this case since we set up an attribute selector and how to register it in app module, we also learned how to get access to the element the directive sits on.

This however is not the best way of changing that style because as you might recall from some earlier

lecture in this course, accessing elements directly like this is not a good practice, you should use a different tool which I'll show you in a second because Angular actually is also able to render your templates without a DOM and then these properties might not be available. It could do this when using service workers, so basically some advanced use cases but nonetheless, it's not a good practice to directly access your elements.

How should you access them then?

Well there is some other helper you can inject, it's the renderer. So let's do thisbut actually let's not do it

in our BasicHighlightDirective, let's create a new one and for this, I will create a new one with the generate command or just g as you learned with the CLI and then directive or just d and then any name of your choice and I'll name this better-highlight. ( “**ng g d (directive) better-highlight”, this should created in the app folder, or else it will throw errors in the terminal, could not find an NgModule** )

So this will give us a new file or two new files. I can delete the testing file and we can also put this into its own folder of course, it's not the default but let's do it here, better-highlight, you could also create a central shared or directives folder where you store all your directives. For now just for demo purposes and to have a clear separation and make this easy to see, I'll put them into individual folders.

So better-highlight directive here, in the app module, I will need to adjust my “path” since I just moved it to the better-highlight folder, so make sure to point to the right path in the right file then. In there we get a selector, appBetterHighlight, now let's inject this better tool I was referring to. It's the renderer which is of type Renderer2 and this needs to be imported from @angular/core.

**$ constructor( private renderer: Renderer2{ }**

Now with this injected, we can use it and I will use ngOnInit again:

**$ export class BetterHighlightDirective implements OnInIt {………………**

to still stick to this best practice of doing initialization work there. Make sure to import ngOnInit or just onInit here from @angular/core and now let's implement ngOnInit here:

**$ ngOnInIt( ) {……………}**

and there, I can now use the renderer. I do this by calling the property which I created here, the private renderer and there we get a couple of helper methods we can use to work with the DOM basically.

Now one important method I'm interested in here is the setStyle method, this method allows us to, guess what? Set the style of some element.

**$ ngOnInIt( ) { this.rendered.setStyle( )……………………}**

Now the problem here is for that, we need to have the element for which we want to set the style

and of course in Angular, there are different ways of getting such an element, here in the directive,

it's exceptionally simple. We already saw how to get one in the basic directive, the BasicHighlightDirective,

we can simply inject the element reference. So I'll add another private property here or argument which

will automatically be bound to a private property I should say which I'll name elRef and this will be of type element ref which we then also, as before, need to import here, element ref from @angular/core.

**$ constructor ( private elRef: ElementRef, private rendered: Renderer2){ }**

Now with this import added here, now I can use this element ref here by calling this elRef and here I then want to access the native element and that's important, we can't pass the reference itself, we need to get access to the underlying element and pass this as a first argument here to setStyle. Now setStyle takes a couple of other arguments, we defined which element we want to style, now we have to define which style we want to set and that would be the background color in our case. With the style property defined,

the third argument we pass to the setStyle method is the value we want assign for this property, now we could set the background color here to blue. The fourth and final argument is a flags object, here we can set a couple of flags for this style and this is optional, I will leave it out here.

**$ ngOnInIt( ){ this.renderer.setStyle(this.elRef.nativeElement, ‘background-color’,’blue’, )**

What you could set here are things like if you want to add an important tag, so this exclamation mark,

important notation for a style to override other styles or something like that. Now this however does it for my cases here and now if we save this with our better-highlight directive already added to app module, we can go to the app component.html and add it here too.

I will simply duplicate this paragraph here and use appBetterHighlight here, ( inside the p tag ) our new directive, again which is available through the selector since we also added it to the app module and then here, I want to style this with a betterdirective.

**$ <p appBetterHighlight> Style me with better directive! </p>**

Now if we save this and let it recompile and reload the browser, we already see the blue background here

behind this paragraph. So our new directive here, the better directive is working and now we're using the renderer which is a better approach of accessing the DOM.

Now why is it a better approach?

Angular is not limited to running in the browser here, it for example also works with service workers and these are environments where you might not have access to the DOM. So if you try to change the DOM as you did here in basic highlight by directly accessing the native element and the style of this element, you might get an error in some circumstances.

Now to be honest, in most circumstances you probably don't and you probably also know if your app is

going to run in the browser or not, still it is a better practice to use the renderer for DOM access and to use the methods the renderer provides to access the DOM. You can learn more about the renderer in an article after this lecture here if you are interested. Just another note, I had to rerecord this part here because the renderer type here changed from RendererV2 to just Renderer2 as did some arguments you could pass to setStyle. Therefore in the other videos in this section, you will see that this code looks slightly different, not by much though.

Don't get confused by this, the code you see here is the code you should use and the code which will work, all the other code shown in the section of course is not changed, it's still the same, it's only this renderer setStyle method and the renderer type which is now Renderer2 and not RendererV2 as you will see in the other videos. With that, let's move on.

**#### 95. MORE ABOUT THE RENDERER**

In the last lecture, we used the Angular Renderer class to change the style of a HTML element. As explained in that lecture you should use the Renderer for any DOM manipulations.

Of course, you can do more than simply change the styling of an element via setStyle( ). Learn more about the available Renderer methods here: <https://angular.io/api/core/Renderer2>

**#### 96. USING HOSTLISTNERS TO LISTEN TO HOST EVENTS**

So we saw two examples for directive now and the better approach definitely is a good approach but

it's not very interactive, right?It always gives us a blue background and I want to change this. I only want to style this blue or give this a blue background if I hover over it and if I move my mouse away, it should go back to transparent.

So let's improve the better-highlight here a bit. We need to react to some events occurring on the element the directive sits on and a quick and easy way to do this inside of this directive is to simply add a new decorator and this is the **@HostListener** decorator which needs to be imported from @angular/core and add it to some method we want to execute.

so the method here could be mouseover() **( @HostListener( ) mouseover( ){…………………………}**

.

Now I'll move this from the top of the file below ngOnInit here, so now this can be triggered whenever some event occurs and that event is specified here as an argument, as a string. @HostListener here takes the argument name as an input and that would be mouseenter let's say**( @HostListener(‘mouseenter’) ).** That is one of the events supported by the DOM elements this directive sits on, so you have basically all events available you could also use event binding before. So that's my @HostListener targeting this event

and we could also receive the event data here, so event data of type event would be passed to us here, so that works. **( mouseover(eventData: Event ){………………………} )**

You can also listen to custom events here and retrieve that data, so that's just like the method you execute when you add a click listener or whatever your event is and then pass the method between quotation marks, so that's happening here, @HostListener is just a convenient way of listening to events on that element.

So with that, we listen to the mouseenter event, we get the event data which I don't need here

but what I want to do in this case is I want to change the backgroundColor of the element. So what I can do is, I can copy this code from ngOnInit and comment it out and now set the style here on mouse enter and now I can quickly copy this and add another method which I'll name mouseleave maybe where I will

listen to the mouseleave event, another officially event I can listen to and then I want to set the backgroundColor to transparent maybe. Now with this in place, we should have a reactive directive.

So now, no background color and if I hover over it, it gets blue.

So that is working as intended, now with @HostListener reacting to user event or to any events.

**#### 97. USING HOSTBINDING TO BIND TO HOST PROPERTIES**

So we learned about @HostListener and this set up here is absolutely fine for what we're doing but we got another decorator we can use which then allows us to not use the renderer. There is nothing wrong with using the renderer but we get an even easier way of simply changing the background color if that is all we want to do in the directive, again using the renderer is not wrong though. The decorator I'm referring to is called **“@HostBinding “**which also needs to be imported from @angular/core and since we got a lot of imports, I'll quickly restructure this to cover multiple lines.

So @HostBinding, that's the new decorator and I will come back to what we pass here in a second. First of all, we need to bind this to some property which value will become important, so that could be a backgroundColor property, a new property I create here which is of type string.

**$ @HostBinding( ) backgroundColor: string;**

Now in @HostBinding, we can pass a string defining to which property of the hosting element we want to bind. Now properties of the hosting element, that is simply what we also access here in the BasicHighlightDirective, style would be such a property and there then the backgroundColor property to be precise.

Therefore we can simply say style. and now camel case backgroundColor and that's all. Camel case

is important here because we're accessing the DOM property which doesn't know dashes.

**$ @HostBinding(‘style.backgroundColor’ ) backgroundColor: string;**

So with this, what we're telling Angular is on the element this directive sits, please access the style property which pretty much any input has on other directives which access something like the value, you of course have to make sure that they are only added to inputs which have this property but here again, the style property should be available on any element.

So here we access the style property and then there a sub-property, the backgroundColor and we set this equal to whatever background color is set to here. Well and we can simply change background color here when we mouseover it, we can set backgroundColor to blue in this case and set it to transparent in this case here, transparent and I will comment out the renderer code here, again not wrong to use it but no longer needed it in this example.

**$ // comment this code out**

**$ this.renderer.setStyle(this.eleRef.nativeElement, ‘background-color’, ‘lightblue’)**

**$ //use this for hosbinding**

**$ this.backgroundColor = ‘lightblue’**

Of course we also have to set some initial color so that we don't get an error before we mouse over it the first time, so here:

**$ @HostBinding(‘style.backgroundColor’) backgroundColor: string = ‘transparent’**

I will set this to transparent initially, give this an initial value. With this it's transparent, no error, looks good and if we hover over it, it works the same way as before but now solely driven through @HostListener and @HostBinding, a great way for working with the element inside of a directive. And of course here on @HostBinding, you can bind to any property of the element you are sitting on.

**#### 98. BINDING TO DIRECTIVE PROPERTIES**

Our directive is looking really great, we're almost there but there's one other thing I want to add,

one functionality I want to add, right now it's dynamic in a way that we can mouseover and then move

the mouse away but we can't decide which colors get used.

Now if that were a directive we want to ship with, like say a third-party package we're offering, the

user, the developer using this directive should be able to dynamically set the value. Maybe we even want to do this on our own app because the color we want to set changes depending on some other parameters in our app.

Right now, the color is hardcoded in there, transparent by default and blue if we mouse over it. So that is something we can improve and we can improve it with a tool we already learned, custom property binding. As a side note, custom event binding also works in directives but you probably won't use that that often. So how can we use custom property binding then? Let's add two properties to which we bind. I'll add it with input here and I'll name the first one defaultColor and set it equal to a string and even assign a default color which is transparent.

$ @Input( ) defaultColor: string = ‘transparent’;

Now @input of course needs to be imported from @angular/core. I'll duplicate this and I'll set this to highlight color, that will be blue by default.

$ @Input( ) hightlightColor: string = ‘lightblue;

So we do have some default values to use but that can be overwritten from outside. So now by default here,

$ @hostbinding(‘style.backgroundColor’) backgroundColor: string = ‘transparent’

$ //change the above to

$ @hostbinding(‘style.backgroundColor’) backgroundColor: string = ‘this.defaultColor

I will assign my highlight color for initially setting this and in my host listeners here, if I mouse over it, I will assign my highlight color, excuse me, here it should be my default color at the top of course, so default color when we initialize this and then highlight color once we mouse over this and then default color again once we move the mouse away, so here, default color.

$ this.backgroundColor = lightblue;

$ //change the above to

$ this.backgroundColor = this.highlightColor;

$ this.backgroundColor = ‘transparent’;

$ //change the above to

$ this.backgroundColor = this.defaultColor;

So with this, it should still work just the way as it did before but now we can actually bind this from

outside. So in our app component where we use the “better-highlight directive”, we can now bind to default color and maybe set this to yellow, should be a string though, yellow and we can bind to highlight color and pass a string which could be red.

$ <p appBetterHighlight [defaultColor]=” ‘yellow’ “ [highlightColor]=” ‘red’ “ >

And now if we save this, notice that appBetterHighlight, the directive itself, the name is not enclosed

in square brackets, now you see it's red and yellow here, though we also detect a bug, initially before we mouse over, it's white.

The reason for this is when we assign default color here, it's just not set here. So to prevent this from happening, what we can simply do is we initialize it here in ngOnInit instead, that also before anything has been renderer but after our values here are available.

So here, we can set this background color to this default, this default color and now it should work fine,

now we have yellow by default. So this now works and now we are able to overwrite this, there are a couple of interesting things we can observe regarding the way we pass down these values though.

The first interesting thing is that we have two extra directives-like looking things on the paragraph and that is just property binding. Now how does Angular know if we want to bind to a property of paragraph, which of course doesn't have a default color or to a property of our directive?

The answer is it just figures that out, it simply checks our own directives and so on first before it reaches the native properties of elements. That's an important takeaway, we can bind to properties of our own directives by simply placing them on the same element, these properties I mean, enclosed in square brackets of course.

Now for ngClass, you'll see that somehow the directive itself is enclosed in square brackets and that's a typical use case especially if you only have one property to bind or at least one main property, then you can provide an alias and we can do this here for, let's say the highlight color and set this equal to your directive selector, so appBetterHighlight in this case.

$ @Input(‘appBetterHighlight’) highlightColor: string = ‘lightblue’;

If I set this as an alias for the highlight color, now this assignment here won't work anymore, so I will remove it, instead now I can enclose my main directive into square brackets and set this equal to red.

$ <p appBetterHighlight [defaultColor]=” ‘yellow’ “ [highlightColor]=” ‘red’ “ >

// remove [highlightColor=” ‘red’ “

$ <p [appBetterHighligh]=” ‘red’ “ [defaultColor]=” ‘yellow’ “ >

So now we should still see yellow and red but now we're having this same style of enclosing the directive itself. And it's important to understand that this is only an option, you can set such an alias but that's not something you have to do and by default as you saw before, the directive name is not enclosed in square brackets, that really only happens if you want to bind to a property which has the same name or alias like your directive selector. One other thing about how we pass down data, one other thing which is true about property binding in general. If you pass down a string, like now we use square brackets and then single quotation marks, while you can take a shortcut, you can remove the square brackets and remove the single quotation marks and that is a special case.

$ defaultColor = “yellow”

if you are passing down a string. You can do, you can add property binding without square brackets if you then also omit the single quotation marks between the double quotation marks and that should work as it did before, no error. You can use this, you will see me use this later in the course with official Angular directive, be careful that if you use it, it's really clear that this is property binding, that no one thinks that this could be a real attribute existing for the element you placed it on, then there is absolutely fine to be used. And this is our directive finished, with some extra features like @HostListener, @HostBinding and dynamically setting values from outside.



**#### 99. WHAT HAPPENS BEHIND THE SCENES ON STRUCTURAL DIRECTIVES**

So now that we had a closer look at attribute directives and created our own one, before creating our

own structural directive, let me explain why this star is actually required ( \*ngFor ). This star indicates to Angular that this is a structural directive, the question is why does it need to know? Because structural directives with that star actually are just a nicer way for us to use them basically. Behind the scenes, Angular will transform them into something else because there is no star in Angular syntax when using directives or when using property binding or anything like that, there is only property binding, event binding, two-way binding and string interpolation and there is no star operator or something like that.

So behind the scenes, Angular needs to transform this ngIf usage into something where we end up with

these tools, properly binding and so on and it does, I can write the same list here differently. So if I try to write this only Odd block differently.

I would end up with the following, I would have an <ng-template> element, that is provided by Angular as the ng indicates and inside this element, we have the content we conditionally want to render.

So in this case here, we would have a div inside of this and in this div, our list item.

       <ng-template>

         <div>

          <li

          class="list-group-item"

          [ngClass]="{odd: odd % 2 !== 0}"

          [ngStyle]="{backgroundColor: odd % 2 !== 0 ? 'pink' : 'transparent'}"

          \*ngFor="let odd of oddNumbers">

          {{ odd }}

        </li>

         </div>

       </ng-template>

Now you probably see that it's basically the same but that it's wrapped in ng-template, the reason for this is that ng-template is an element which itself is not rendered but which allows us to define a template in the end for Angular to use once it determines that this template, some element needs to be rendered because this condition is true in this case. So on the ng-template, we place ngIf and now not with the star because this is the form to which it will get transformed due to the star but instead with the tool we already know, with property binding and here( <ng-template [ngIf]=”!onlyOdd”> ), we simply bind not onlyOdd, so the condition.

So now we're back to the tools we know without the star and that's why we have the star because obviously that's the more intuitive syntax to use, directly placing it on the element we want to conditionally render but that would be the real content behind the scenes to which it gets transformed.

Now you don't need to use that but I find it super important to be aware of this. Now therefore if we save this, we see the list twice and we see it disappear and appear twice because it works exactly the same,

it is exactly the same code, behind the scenes this is what the star transforms it to.

**#### 100. BUILDING A STRUCTURAL DIRECTIVE**

Now that we understood what the role of the star is, we can write our own structural directive. So let's create a new directive and I will use the CLI for this, I'll name it unless, so I will basically create the opposite of the ngIf directive. **( ng g d unless )**

This directive here will attach something only if the condition is false, ngIf does it if the condition is true.

So here, I'll again delete the spec file here and in the unless directive, I get my selector which is fine.

Now here, I need to get the condition as an input, so I'll add @input and remember in the end, we will use property binding with the square brackets because Angular transforms it for us, we need to of course import input from @angular/core and then here, I want to bind to a **“property named unless”,** which kind of simply is the condition we get but whenever this condition changes, so whenever some input parameter here changes, I want to execute a method and therefore, I can implement a setter with the set keyword.

**$ @Input( ) set unless( )**

This now turns this into a method, though technically and that's important to understand, this still is

a property, it's just a setter of the property which is a method which gets executed whenever the property changes and it of course changes whenever it changes outside of this directive, so whenever the condition we pass changes or some parameter of this condition. Unless therefore needs to receive the **“value”**, the property we would normally get as an input and we know that this will be a boolean because it will be our condition in the end.

$ **@Input( ) set unless( value ){………………………….}**

**//if you don’t give parameter here value, it will throw error in the console.**

**$ @Input( ) set unless (condition: Boolean){……………………………..}**

so we could also name this condition. Then we can check if the condition is not true, which is the case

in which I want to display something because unless is the opposite of ngIf and if the condition is true,

well then I want to display nothing. So that is how we get the condition, how we use it, how do we display something? Keep in mind that our unless directive here in the end will sit on such an ng-template component because that is what it gets transformed to by Angular if we use the star.

So we can get access to this template and we also need to get access to the place in the document where we want to render it, both can be injected. The template can be injected by adding “templateRef”, any name you like but the type is of type template ref, so just like element ref gave us access to the element the directive was on, template ref does the same for a template and this is a generic type, you can simply pass any here and we need to import template ref from @angular/core. The second information piece we need is the view container, so where should we render it?

$ **constructor ( private templateRef : TemplateRef<any>, private ) { }**

The template is the what, now the question is where. So I'll name it vcRef for view container reference and the type is view container reference or view container ref which is also imported from @angular/core. That marks the place where we placed this directive in the document, Angular marks this place and you can see this if you inspect it in the developer tools actually. So with these two tools available, we can use

$ **constructor ( private templateRef : TemplateRef<any>, private vcRef: ViewContainerRef ) { }**

the vcRef whenever the condition changes, to call the createEmbeddedView method which does just what the name implies, it creates a view in this view container and the view simply is our template ref. So this template we created there is exactly this reference to the template there, is exactly what we need.

Well and if the condition is true in this case, so if its not what we were looking for, then we will simply call the clear method to remove everything from this place in the DOM.

export class UnlessDirective {

  @Input() set unless(condition: boolean){

    if(!condition){

        this.vcRef.createEmbeddedView(this.templateRef);

    } else {

        this.vcRef.clear();

    }

  }

With that, our own directive is set up, of course we need to make sure that it is added here(app module.ts), the CLI did this for us and now in the app component, we can use our own directive, maybe to replace ngIf here,

<div \*ngIf="!onlyOdd">

          <li

          class="list-group-item"

          [ngClass]="{odd: even % 2 !== 0}"

          [ngStyle]="{backgroundColor: even % 2 !== 0 ? 'pink' : 'transparent'}"

          \*ngFor="let even of evenNumbers">

          {{ even }}

        </li>

        </div>

so I'll comment out all this stuff here and only copy the original div down there, comment this in of course.( the entire div that contains boht the \*ngIf=”onlyOdd” and \*ngIf=”!onlyOdd” ) So here instead of using ngIf, I'll use \*appUnless and the star is important because it still is a structural directive, otherwise we would have to manually write it with this ng-template syntax you learned before.

So here of course, we don't want to check if onlyOdd is false because keep in mind, the unless directive

will check for the opposite already, so here we have to pass just onlyOdd. And with this in place, we get that we can't bind to appUnless because it's not a known property. Why do we get this? Can be really hard to track, we get this error because what we're trying to do is we have property binding here, custom property binding with @input and we're binding a property named unless.

export class UnlessDirective {

  @Input() set appUnless(condition: boolean){

    if(!condition){

        this.vcRef.createEmbeddedView(this.templateRef);

    } else {

        this.vcRef.clear();

    }

  }

Now keep in mind, the star automatically transforms this in this ng-template syntax where we then try to property bind to the directive name which is appUnless. So we have to make sure that our property here shares the name of the directive, appUnless, exactly the same, the same as the selector. Now with this, it works fine and as we can see if I toggle here, we get the same behavior as before even though I commented out the ngIf block and use my own appUnless directive instead. So this is our own custom structural directive built.

**#### 101. UNDERSTANDING NGSWITCH**

I want to conclude the section with one other built-in structural directive which might be handy from time to time, **“ngSwitch”.** Imagine the case that we have a value like this which is simply 10 ( value = 10 ), anything you want. Now we got a place in our app where this value changes and we get a couple of different messages we want to display for each of these values. For this, we can use ngSwitch, so let's add a new <div> below our paragraphs with our own attribute directives and “ngSwitch” is used like this.

We are bind to ngSwitch with property binding, so no star at this point and we bind to value.

$ **<div [ngSwitch]=”value”>**

So this is our condition basically, what we want to check and then you might be aware that switch has a couple of cases we can now cover. So let's say we have a paragraph where we say value is five, then we have the same for 10, then we have the same for 100 and maybe for default.

<p>Value is 5</p>

<p>Value is 10</p>

<p>Value is 100</p>

<p>Value is default</p>

Now we need to add something to these paragraphs to control which paragraph gets shown because it should only be one of these at a time, that something is **“ngSwitchCase”** and we simply pass the value forthis case here as an argument, for example 5 and we need the star at the beginning, that's super important. So here, the star comes into play again because behind the scenes, this is done also transformed by Angular.

<p \*ngSwithCase=”5”>Value is 5</p>

So now we can use this here for the other use cases too, with 10 and with 100 and finally for the default case, we have ngSwitchDefault, like this. Now here, we also need to add the star of course. With this in place, if we save this, we only see 10 here in the browser and if I go to my component and I change this to 5, like this, then of course we see five. So this is ngSwitch, super easy to use, might be useful in some cases where you find yourself creating a lot of ngIf conditions, ngSwitch might be the better solution in such cases.

So that is structural directives finished and that is directives overall finished. You learned a lot about them and I hope this knowledge allows you to create awesome directives for your applications in cases where the built-in ones aren't everything you need even though these already cover a lot of the default use cases to be honest. So let's move on, let's add a directive to our recipe book in the next section and then let's move on in the course.

See you there.