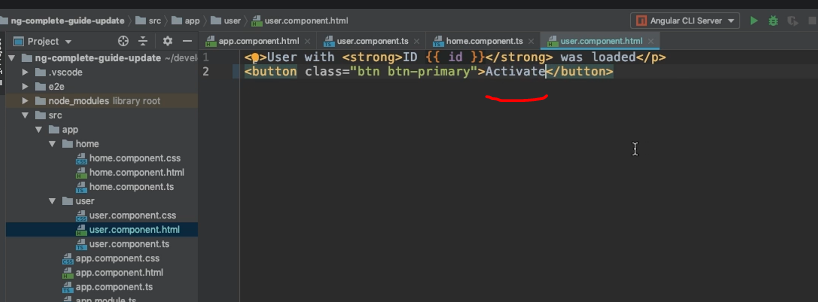
**177. Subjects**

* Now that we learned so much about observables, we're almost done with that section and ready to continue but there is one other important construct that you should not miss and that is the subject .
* **Now what's the subject**?
* Well let's say in our user component here, we have a button which I'll add and I'll give it a class of btn-btn-primary and I'll give it a label of activate .



* Now the idea is that when we click this button, we change something in the app component, so in a different component and we simply print activated here let's say .

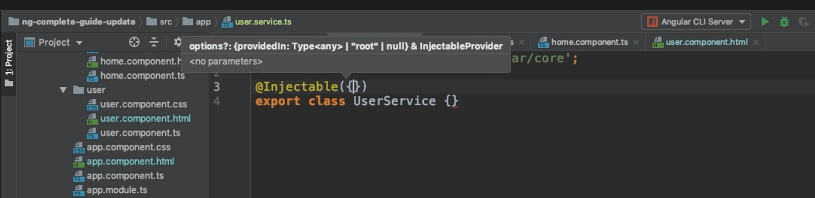
Text

Description automatically generated

* So we add another horizontal line and in-between, we want to print activated but we want to only show this if we clicked activated on the user component .

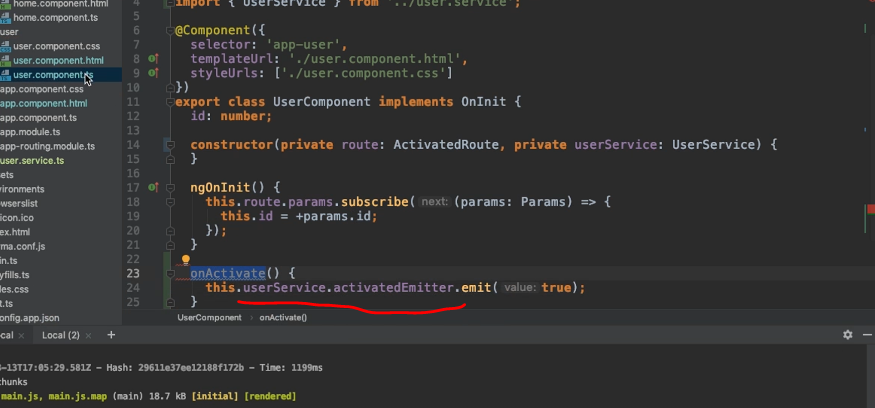
**service and an event emitter**

* Now how can we make this work? Well with a service and an event emitter for example, that is something you also learned about earlier .
* So let's quickly do that, let's add a new file here, the user .
* service .
* ts file, here next to the app component and in there, I'll export a class, user service, nothing too special, it will get the @injectable decorator which is imported from @angular/core and I will use the more modern syntax for providing it, I'll add provided in root here .



**Activated Emitter:**

* Alternatively, you use simply add it to providers in the app module, here which is also fine but this is a shortcut, a shorter and easier way .
* Now we have that service and in there, I'll add a property, activated emitter, which is a new event emitter and event emitter is also imported from @angular/core and that's a pattern I showed you earlier already and that event emitter will emit a boolean and we can now use that event emitter from inside our user component if we inject that service here .



* So the user service is injected here and for that of course, make sure to also import it from that user service file and now we just need to add a new method, on activate and in there, I'll call my user service, reach out to activated emitter and call emit and pass true here as a value .

**Wire up on Activated Emitter:**

* Now the last step is to wire up on activate, here in the user component to the button by adding a click listener to the button and there, I will call on activate .

Text

Description automatically generated

* Thus far, nothing too fancy, that is a normal way of passing the data around or at least, of emitting an event across components .
* **Listening to Event:**
* Now we want to listen to that event from inside the app component and for that, we simply inject the user service here too by using that injection shortcut also where we automatically bind it to a private property named user service and I'll then add a new property here too, user activated which is false initially and in ngOnInit,

Text

Description automatically generated

* we can set up a listener, we can reach out to the user service, activated emitter and subscribe to it and in there, we get our did activate, or whatever you want to call it, boolean and we'll set user, this user activated equal to did activate and therefore whenever we emit true as a value here, user activated will switch to true and in the app .
* component .
* html file, that now means that we can use ngIf here to conditionally show that paragraph as soon as user activated is true .

Text

Description automatically generated

* Let's save all of that and let's navigate to that user page here and let's click activate and indeed, we see activated here .

Graphical user interface, text, application, email

Description automatically generated

* So this is working, which of course is nice .
* **Better Approach using Subject:**
* This is the old approach with the event emitter and this is an approach you could use but there is a better one, a more recommended one and that new approach, the better approach uses a subject .
* Now a subject is something we import from RxJS, SO you import subject from RxJS, instead of event emitter, you now create a subject here .

Text

Description automatically generated

* Other than that, it's pretty similar though, it's a generic type where you define which data will eventually be emitted, in this case a boolean .
* So it's very similar to event emitter thus far
* **Observable Vs Subject:**.
* We also use it very similarly, in the user component we don't call emit however but something you saw before already, you call next because a subject is a special kind of observable you could say .
* We know observables, we can subscribe to them as you learned but they're rather passive .
* You'll learn how to create your own observable but the core idea always is that you wrap a callback or an event or something like that .
* *A subject is different, a subject also is an object you can subscribe to but it's more active because you can actively call next on it from outside .*
* Remember in the observable, we also called next but that was from inside the observable when we created it .

Diagram

Description automatically generated with medium confidence

* So that is a more active observable that is perfect when we want to use it as an event emitter, so if we don't have a passive event source, like an HTTP request or DOM events but if we have something that actively needs to be triggered by us in our application and that's exactly the use case we have here .

Text

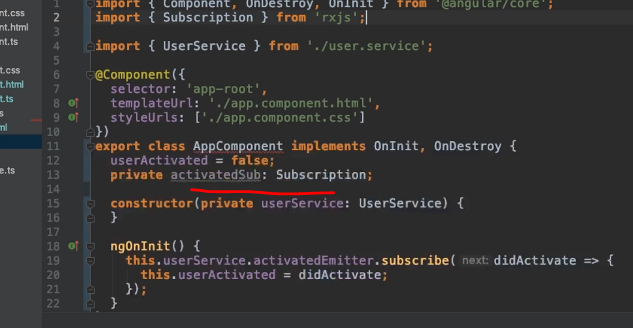
Description automatically generated

* We now can call next here in the user service on our activated emitter, the activated emitter is that subject and in the app component, we still call it subscribe because it is an observable and therefore, we can subscribe .

Text

Description automatically generated

* If it now save that, you click activate again, it still works but now it's a subject and whilst it technically of course works in a similar way, using a subject is recommended way .
* Don't use event emitter,, use subjects they are in the end a bit more efficient behind the scenes, you can also now use all these cool operators because a subject in the end also is kind of an observable .
* So you have that too which is a huge advantage and it's simply something you can and should keep in mind, use subjects instead of observables .
* *One important note, just as with your own observables, you should unsubscribe to your subjects though whenever you don't need them* .
* So add onDestroy here to the app component which you need to import or to any other component where you set up a subscription to your subject .
* Store the subscription, activated sub is a subscription, subscription needs to be imported from RxJS, so make sure to add that import in your file as well .



* Store that subscription then, so here where you call subscribe, you simply set this equal to activated sub so that the subscription is stored here and then you add ngOnDestroy and inside of ngOnDestroy, you simply use your subscription and call unsubscribe and now you have a great application without any memory leaks or anything like that .

Text

Description automatically generated

Text

Description automatically generated

* *Now one important note about subjects as a replacement for event emitters, this only counts if you're using them as cross component event emitters, where you manually call next or previously emit .*
* *You don't use subjects instead of event emitter when you're using @output .*
* So in a component if you're using @output here with your own event, you still use the Angular event emitter, you're not using subject there because the subject is not suitable for that, there you need the Angular event emitter***, you only use subjects to communicate across components, through services so through a mechanism where you in the end subscribe to somewhere, like here in the app component .***
* If you're not subscribing to an event emitter, then it probably is an output, if you do plan to subscribe manually, then it is a subject .