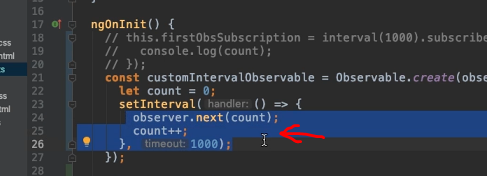
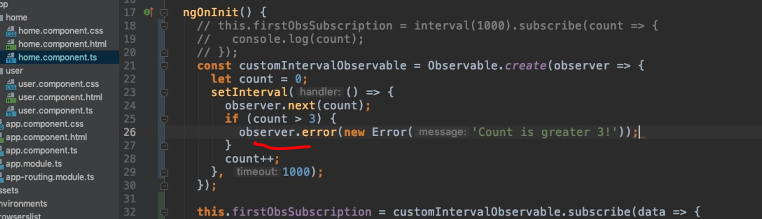
**174. Errors & Completion:**

* -: Emitting new data is arguably the most important thing observables do .
* And in I'd say 99% of all cases when you subscribe you'll pass that first argument where you are interested in the data you are getting .
* Or maybe it's 90% because there is one other important use case too especially when we think about things like HTP requests and *that is error handling* .
* This counter here can't fail .



* There is no error that could occur here but that's a different story if you're sending HTP requests which we will later do .
* **Throwing Error:**
* So here, of course we can also fake an error .
* We can simply throw one and define our own condition when we want to do that .
* Now to do that, we could, for example, check if count is greater than three .
* So if that occurs, then we'll first of all still emits that new data with the next method .
* But then we'll also use the error method to throw a new error here .

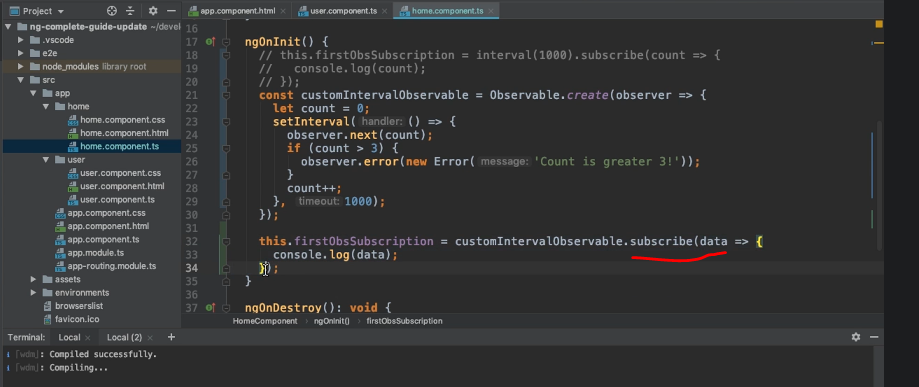


* So here I'll create a new error object, count is greater free, which of course is a horrible event, that definitely is worth an error .
* Now if we do that and I save everything, let's see what happens .
* It starts counting again and as soon as we exceed three we get an error here, right? We get our error that we defined .

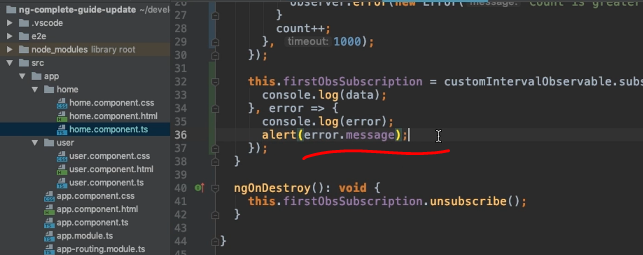
Graphical user interface, text, application

Description automatically generated

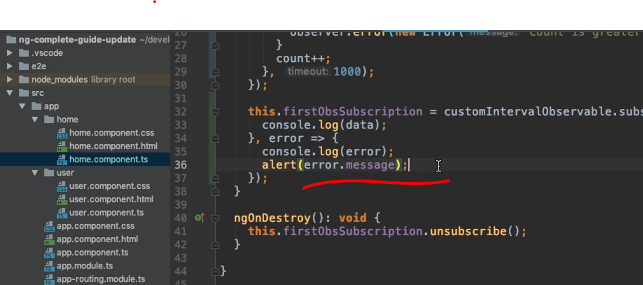
* And you see that the observable now stops emitting .
* *It actually gives us four, which makes sense because that's the last value it emits before this condition is true .*
* ***But there are no other values emitted because whenever an observable throws an error, it cancels, it's done .***
* It will not emit any error values, it dies so to say .
* And therefore in that case you also don't need to unsubscribe .
* You still can unsubscribe .
* If you never get away, you'll not get an error .
* But it's not necessary because, well, it was dead anyways but you might not have known that when you unsubscribe .
* So unsubscribing is not an issue .
* Now this is how you can throw your own errors and of course it's more important to not just throw them that most of the time will be done for you when you're using some built in angular functionality that can fail like HTTP requests .
* **Error as Second Argument:**
* But it's important to know how to handle that .
* And for that you can pass another argument to subscribe .
* Thus far, we pass the first argument which is this data function .



* The second argument would be our function that gets called when an error occurs and will get the error as an argument there .
* Now of course the simple thing we can do here is that we log that error to the console but obviously you could do way more than that .
* You could send it to your own backend and store it in a database there .
* **Showing Alert:**
* You can show an error message and alert to the user .
* Let's actually do that .
* Let's show an alert .



* And let's show the error message here in the alert .
* So that's what we're doing here .



* We're logging it to the console, we're showing an alert and now it starts counting again .
* And as soon as we exceed three, here you go .

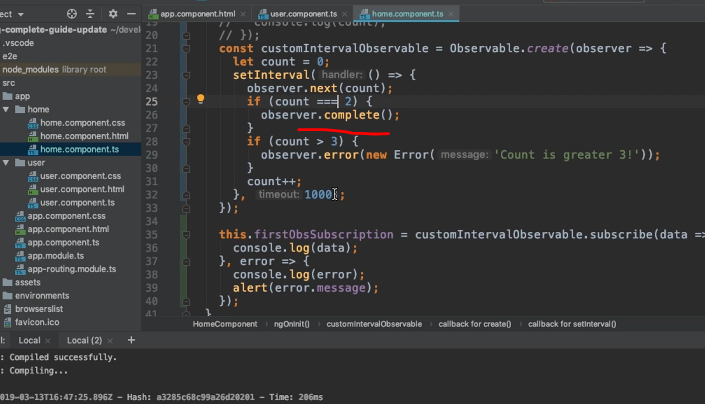
Graphical user interface, text, application, Word

Description automatically generated

* Now we get our alert and you see we don't have that red error here on the right anymore .
* Instead it's now a regular console log because that's what we're doing .
* So now we're handling that error and that of course is also an important part .

**Completing the Observable:**

* Now handling errors is nice but what about completing the observable? I told you that throwing an error actually cancels the observable and lets it die .
* But completing it is something different .
* Completing can be a normal process in an observable .
* *Now our interval by default doesn't complete .*
* It will emit new values until the end of time .
* *And HTTP request on the other hand, will complete .*
* *It completes whenever a response by the server is there .*
* And of course here when we're building our own observable from the crowned up, we can also complete this manually .
* In the end, we are defining how our observable behaves .
* Our event source is the set interval we have wrapped in our observable and the (indistinct) before we throw an error let's say .
* We can also add a narrow condition .
* Say if count is equal to two, well then we simply call observer complete .

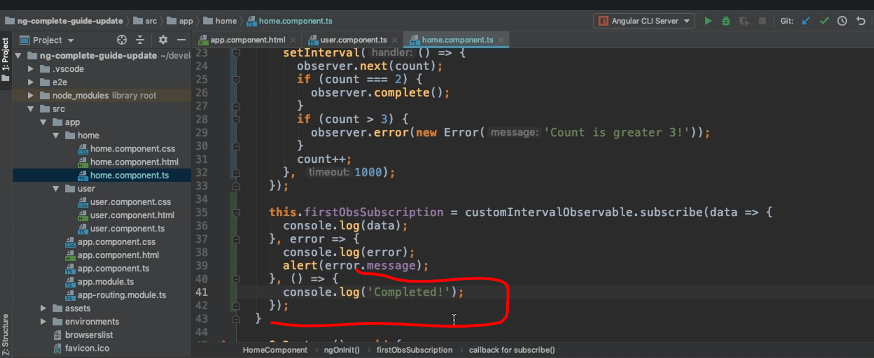


* And there you don't need to pass any arguments .
* Now when you call complete, the observable will really come to an halt and you can see the view .
* Now reload .
* It logs zero, it logs one, it logs two, and that's it .

Graphical user interface, application

Description automatically generated

* Doesn't log anything else .
* Doesn't reach our error condition because we complete the observable before it gets there .
* And that is important to understand and to keep in mind whenever an observable completes, it really is done .
* There are no other values emitted thereafter which kind of makes sense because well it completed .
* **Reacting to Completion:**
* Now if you want to react to that completion *you can add a third argument to the subscribe method* and that is your completion handler function .
* It's a function that gets no arguments because completing doesn't pass any arguments and it's simply a function where you can do some cleanup work or whatever you need to do .
* And here I'll just log, completed .



* ***Also important, you don't need to unsubscribe if your observable did complete, but again, you might not know that here in ngOnDestroy .***
* So you can still do that without getting errors .
* So if I reload now, we see our logs again and after the two we see completed .

Graphical user interface, text, application

Description automatically generated

* Now all the worth noting though, and important to keep in mind, you might think that complete, so that complete function here, all the fires when you're getting an error, after all the observable does finish after throwing an error, right? Well, that's not the case .
* If we change the complete condition here, temporarily, to let's say four, and therefore this to five, let's say and therefore, this will happen after this condition, which actually will throw an error .
* You will see something that might surprise you .
* We see our normal output here and then we get that error and that's it .
* What you don't see anywhere is our completed log .
* So this here never gets logged and that's just one important characteristic you have to keep in mind about observables .
* ***When it cancels due to an error then that's a different thing than when it completes .***
* ***An error cancels the observable .***
* ***It does not complete it .***
* Technically, in both cases, no new values are omitted .
* But regarding the functions that get called here there is a difference .