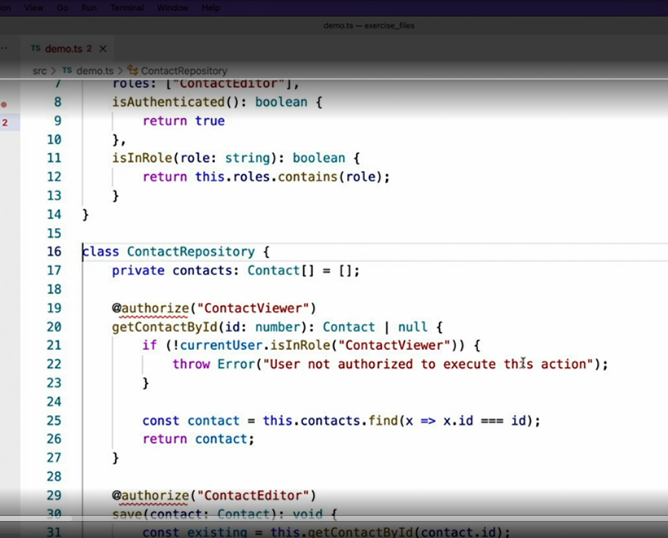
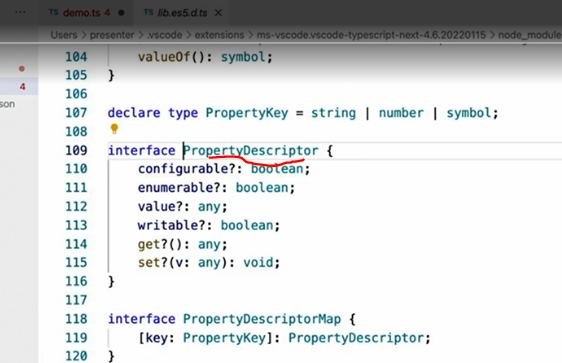
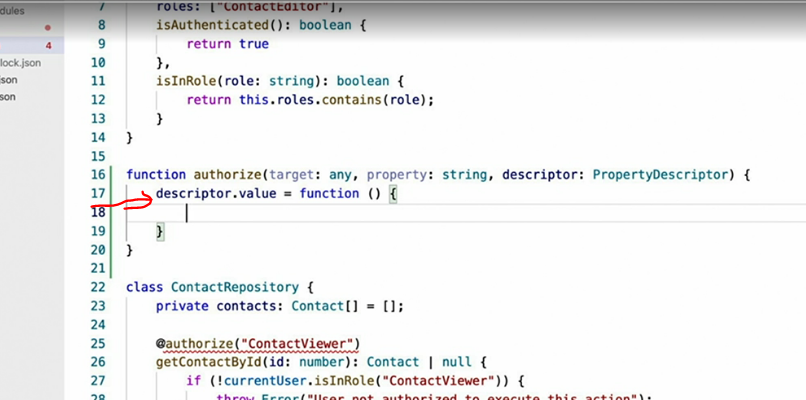
* - [Instructor] The best way to learn how to create a decorator, in my opinion, is with a method decorator.
* As I find it the most straightforward to understand and implement.
* As its name implies, **a method decorator is a decorator that you apply to a method** on line 19 and 29.



* Implementing a method decorator is ***as simple as writing a function*** that looks like this.
* This function **gets executed at runtime and accepts three parameters.**
* A)The first parameter **target** is the *object that the decorator is being applied to.*
* In the case of a method decorator, it is ***the instance of the object that the method belongs to***.
* B) The second parameter **property** is the **name of the property that the decorator is applied to.**
* C) And finally, the **third parameter descriptor** is an *object containing the current metadata about the property.*



* Now, this descriptor object is actually the same for any property of any object, regular fields, getter setters, and methods.
* In the case of a method, however, this value property will be the one we're most interested in because its value is the function that is executed when the method is called.
* When creating a method decorator, you have **two options to modify the behavior of the method.**
* A) The first option is to *simply edit the descriptor* in place like this.
* This approach leaves the existing descriptor in place and just modifies its properties.

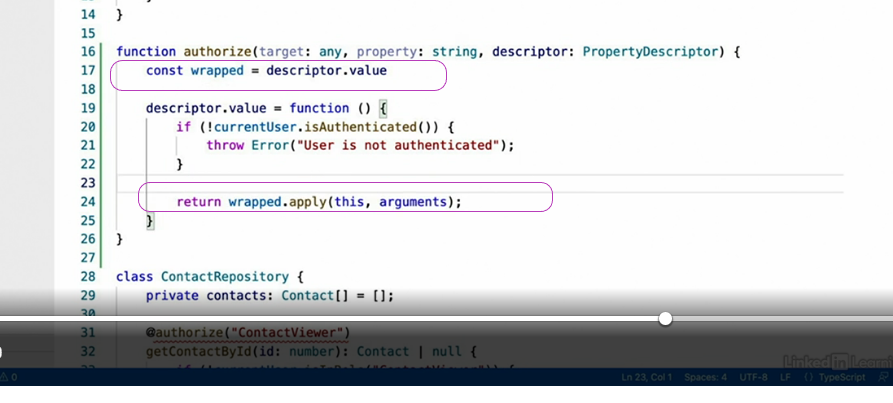


* B) The second option to *return a brand new descriptor* object like this.
* Once returned, type scripts will replace the method's current descriptor with this new one.

Graphical user interface, text, application

Description automatically generated

* Now, neither approach is inherently better than the other.
* It's mostly just a *matter of style* and the current problem you're trying to solve.
* Now, let's get back to our example, implementing this authorized decorator, which throws an exception when the user is not authorized to access a method.
* To do this, I'm going to replace the *descriptor's value property with a new function.*
* And I'll start this function by checking to see if the user is authenticated.
* And if not, we'll throw an error.
* This is the new logic that we want to add to the method, but keep in mind that this new function completely overwrites the current method.
* So we still need to execute the original logic.
* To do this, I'll make a copy of the current method before I overwrite it.

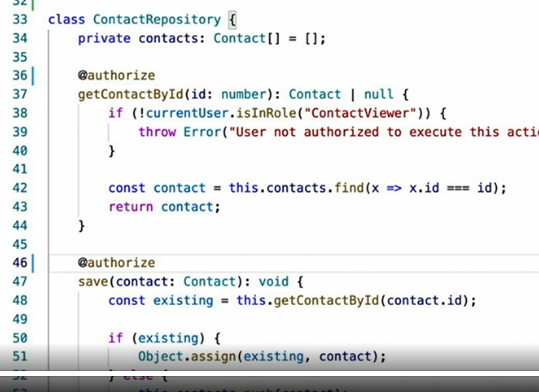


* And then, I'll simply call that function from within my new function, using the apply approach.
* And that's it, we've wrapped the methods logic with our own.
* Here is where you can really your imagination run wild with what you can do in this code.
* For example, you can wrap every method call in a try-catch block, and automatically log every exception that occurs, but still not affect the way the application behaves.

Graphical user interface, text, application

Description automatically generated

* Now that the decorator is ready, we can apply it to whatever methods we like using the syntax I showed in the previous video.
* With this in place, our decorator now guarantees that users are authenticated in order to access these methods.
* Otherwise, an error is thrown.



* However, you may notice that this is not actually a drop in replacement for the code that I had here before.
* This code checks not only that the user is just authenticated, but that they actually belong to a particular role.
* In order to implement this behavior, we're going to need to add a parameter to our decorator so that we can pass in the role that should be checked.
* Head onto the next video, where I'll show you how to do exactly that.