In this video, let's now create</v> a better and more useful error class, and also do some refactoring. And starting with that error class, let's create a new file in our Utilities folder. So, new file, and I'm going to call it AppError because that's going to be the name of the class, all right? So, class AppError, and we actually want all of our AppError objects to then inherit from the built-in error, and so let's actually extend the built-in error class. So, we used extends Error, okay? So, we did that before somewhere in this course, and again this is just ES6 classes, all right? In this case, class inheritance, where one class inherits from the other, okay? Then, as always, our constructor, and what we're going to pass into a new object created from the AppError class is going to be the message and the statusCode, so just these two. Okay, so, remember the constructor method is called each time that we create a new object out of this class. Now, as usual, when we extend a parent class, we call super in order to call the parent constructor, all right, and we do that with message because the message is actually the only parameter that the built-in error accepts. Okay, so this is basically you're, like, calling error, all right, and then this is here is the usual stuff, we set the statusCode to statusCode, all right, and now we also want to set the status itself. So, remember the status can either be fail or error, and we could pass that into the object, but it's also not really necessary, because the status depends on the statusCode. So, when the statusCode is a 400, then the status will be fail, and if it's a 500, then it's going to be an error, right, and so that simply tests if the statusCode starts with a four. So, in JavaScript, there is a startsWith method that we can call on strings, and so I think that's the easiest way of doing this test, and so that's basically convert the statusCode to a string, and for that I'm simply using a template string with the code in there, so this, or actually we can just use statusCode. So, statusCode, and then startsWith, then here we also need a string, and so if the statusCode as a string starts with a four, well, then we have a fail. And so here, let's use the ternary, okay, and so it's fail when it starts with a four, and otherwise it's an error, okay? So, very simple, and this already saves us from manually have to pass in either fail or error. All right, now next up, all the errors that we will create using this class will all be operational errors. So, errors that we can predict will happen in some point in the future, like for example a user creating a tour without the required fields, right? So that is an operational error, okay, and so again, from now on, we will always use this AppError class here that we're creating right now in order to create all the errors in our application. And so these errors will be operational errors, and so what I'm gonna do now is to actually also create a .is operational property here. So this.is operational, and set it to true. So all of our errors will get this property set to true, and I'm doing that so that later we can then test for this property and only send error messages back to the client for these operational errors that we created using this class. And this is useful because some other crazy unexpected errors that might happen in our application, for example a programming error, or some bug in one of the packages that we require into our app, and these errors will then of course not have this .is operational property on them, all right, and it might sound a bit confusing at this point, but don't worry, it will make a lot of sense when we then implement that part that I just mentioned. So, testing for this .is operational property here. All right, and now just one last step is that we actually also need to capture the stack trace. Now, what do I mean by stack trace? And so, let me actually show that to you, so, show you what stack trace is, and so I will log it here to the console. Console.log and error.stackTrace, so each and every error gets this stack trace, and actually it's just stack, okay. So err.stack will basically show us where the error happened. So let me run this here now, and so let's take a look at this, and so it gives us here the error and then also where it happened, okay? So, in this case here, of course, add app.js at line 32. So, right here. So, that's where we created this error, and so it's now in our stack trace, and it also shows us the entire call stack here, which in the end the originated in this error, okay? So we kind of want to preserve that and also at the same time not add this method here, or this class, to that stack track, okay? And again, that sounds a bit confusing, I know, but in this case it's not really important, all that matters is that we just add this line of code here, which is Error.capturestackTrace, so exactly what you get here, and at first we need to specify the current object, which is this, and then the AppError class itself, which is gonna be this.constructor. Okay, and so this way when a new object is created, and a constructor function is called, then that function call is not gonna appear in the stack trace, and will not pollute it. All right, so let's put it down here, and yeah, so that's our AppError class, and so let's now actually export it from here. So, module.exports is of course the AppError. Great, just one question that you might have is 'Why didn't I set this.message equal to message?' Well, that's just because right here I called the parent class, right, and the parent class is error, and whatever we pass into it is gonna be the message property. So just as I told you before. And so, basically, in here by doing this parent call we already set the message property to our incoming message. All right, give it a save, and now we can close it, we no longer need it, and here we're just gonna import it here. Const AppError, require, and then it's in utils, so Utilities, and appError, okay, and so now let's actually go ahead and use it. Comment out this piece of code, and actually delete the one that we had before, and that will now create the error right here inside of next. So, new AppError, and then the message, which is this one, and the status code. Right, and the fail, remember, will then automatically be figured out, and so let's delete all of this, give it a save here, and one more time, test out this wrong route, and indeed we still get our same error, and we also still get our exact same stack trace. Okay, and finally I actually also want to export this middleware here, okay? So basically, this handler because throughout the rest of the section, we're gonna build a couple of different functions for handling with different types of errors, and so I want all of these functions to be all in the same file, all right? And we can say that all of these functions that I just mentioned are handlers, okay, and so handlers, we also call them controllers in the context of the MVC architecture, and so let's now actually create an error controller file in our controller folder. Okay, and this might sound or look a bit weird because we actually do not have an error resource okay, and so probably some people are gonna disagree with me that this is the correct place, but I personally like to do it like this because at the end of the day, these functions, they kinda are like really for controlling our errors, all right, and so for me at least it makes sense to simply call this function here the error controller, and here I wanted to paste that middleware function, but I guess I didn't copy it, so let's do that again. All right, and so actually I'm gonna export this one here as the module.exports because the other handle functions that we're gonna create later on, I will not export them from here. So they are just kinda just gonna be helpers. So, module.exports equals this function, okay? Let's actually get rid of this console.log here, give it a save, and now back into our app we of course now need to plug in that middleware function here. So, let's again go ahead and import it and I can call it whatever I want, and so let me call it the globalErrorHandler with a capital H, and I'm gonna require controllers and errorController. Put it back here, and now for the final test after this refactoring. Let's see, and indeed one more time, everything works just fine, okay? So, perfect, that was the goal for this video, see you in the next one.