**Lecture 306**

**Types of Errors and Error Handling**

So how bad are errors? Now errors are not necessarily the end of your app, you can recover from the errors, you can inform the user that something went wrong and that he should try again for example, you just need to handle errors correctly that is the key takeaway and there are different types of errors. We can have technical or network related errors where you have very little influence on at least if you're not the system administrator, we have so-called expected errors and this is not an official term, that is something I came up with and I'll explain what this is in a second and we also have bugs or logical errors in your code. Now for technical errors, our mongodb server might be down and therefore any interaction with the database will fail. In such a case, there's not that much we can do, the best thing might be to show some error page to the user to let the user know that something is wrong on our end, that we are sorry and that we're working on fixing the issue. We also might want to behind the scenes send an e-mail to the administrator or anything like that. We also have these expected errors as I like to call them. There are certain operations, let's say we are interacting with a file or with a database that can fail, not very often and of course it's not really expected for this to fail but that can happen. Maybe because there are too many simultaneous requests to a certain file, anything like that. Here informing the user and giving the user the ability to retry might be a good solution. For example the validation errors which we also implement in earlier module, these would also be expected errors, users will input valid and invalid data and for invalid data, we want to inform the user and give the user the chance of retrying and of course we also have errors in our code where we interact with a user object in a place where it just can't exist or at least not in all circumstances. We should fix such errors during development, we should test our code and we should fix such issues of course, these are not errors we should handle at runtime, we should not show a message to the user because these errors are not the users or the network's fault, they are our fault. So how can we work with the different types of errors then? We have to differentiate. There are errors where an error is thrown, an error is a technical object in a node application. So there is a built in error object which we can throw that's also a javascript language features, basically all programming languages have such a feature. We also can have scenarios where we can't continue with our code but there is no technical error. An example would be that we try to log a user in but the email address does not exist, this is not really a technical error, there is no error being thrown but we know we can't continue and so we want to check for this scenario as well and handle it appropriately. Now for the errors thrown part, we have certain tools we can use to test code and catch potential errors so that we can handle them gracefully and for synchronous code, that would be try catch blocks. For asynchronous code with promises, we have then and catch which you already saw quite a lot throughout this course. In the end in both scenarios, we then have the choice if we want to directly handle the error or if we use a mechanism built into express, a special error handling middleware which we haven't used thus far which you can use to catch errors and then return a response to the user and I'll show how this works in this module of course. For this scenario that no error is thrown, well we just have to check values with if checks for example and then we can decide whether we want to throw an error, to kind of enter the left world here and then kick off that error handling process or if we want to directly handle the error which is not a technical error but where we simply add some code that can continue with the missing input data for example. In all cases, we've got different ways of communicating with our users. We can return an error page, so a dedicated page that informs the user hey we have a problem and that of course should be kind of the last resort because there the user loses all his input, can't continue. We also can return the page which user was on and just give some error information, that is what we did for validating for example, there we returned the page the user was on, kept the input values and just added an error message. Or we could redirect, for example if we try to access a page which we are not authenticated to visit, then we can redirect the user. So these are the different tools we have, the different ways of working with errors we have and now let's dive into our code and see what we're already using and what we can use.

**Lecture 308**

**Errors – Some Theory**

So let's dive into throwing and handling errors and here I do have an error in my custom validator. Now this will be handled behind the scenes by the express validator package, so let's add some code that is not handled behind the scenes. And for that, I'll first of all take a step backward, I even quit my development server and I'll temporarily add a new file here, errorplayground.js . I will remove that, it's not related to the app but there we can execute javascript code and we can execute it with the node runtime of course and in here, let's create a new constant, sum where I take two values, A and B, whoops, so sum will be a constant that holds an anonymous arrow function, A and B and I return A plus B and then now there I call sum with 1 and 2 and I console log the result let's say. Now if I do this, we can execute this file with node error playground.js like this and we get three in the console down there. Now that's of course nice but now let's add an error. Let's say I only pass one argument instead of two, now if I execute this again, I get not a number but not an error, not a technical error object. Now let's say here I check if A and B, so both has to be true-ish, then I will return A plus B otherwise I'll throw, that is a built-in keyword, a new error and that's a built-in object node ships with where I say invalid arguments, something like this. Now let's execute this file again and now we see such an error message which we saw before in the course as well. Here we have our own error message and then we got a callstack which allows us to find out at which function and which line number this error was thrown and what was called before that error. We saw that before because that is an unhandled error. We throw an error and that is a built-in functionality, throwing such errors and node and a lot of the packages we use also throw errors behind the scenes, for example mongodb will throw an error if it can't connect or if an operation fails. So such errors can be thrown and if we don't handle them, then our application just crashes and that's what we saw earlier in the course too. You might remember cases where I did something and we got stuck and that refresh icon in the browser kept on spinning and nothing happened, that was because our server crashed because we had an error which we did not handle. Now how can we handle errors? Well one solution for synchronous code, so code that executes line by line immediately and does not wait for anything, so for example where we don't interact with files or where we don't send requests, well such code can be wrapped with try catch, another built-in language feature. We try a certain code and then we have to add catch where we catch a potential error that might have been thrown and in catch we can now handle it, for example here I could output error occured and then I console log my error like this. Now if I re-execute myerrorplayground.js file, I still get this error here but I get this additional error occured message and if I not log my error object here and I execute this, then I actually get just this, so then it does not crash and log it automatically but we could do anything we want. We could continue with other code after this, so here I could have this works. I could execute this line and we see this works being output here. Just to demonstrate this, if I comment out try catch and I just try to console log someone, so I call this with an error being thrown, then we don't see this works anywhere because it crashes with our error that is being thrown and it does not continue with other code. So this is why handling code like this is a good thing to do because this ensures that we can continue with code, that we can handle this gracefully, in our node express application we could send an error response here which renders a valid page without crashing everything but which informs the user that something bad happened and in the end, this is what the express validator package does for us with our thrown error. In auth.js where I throw this error, in the end express validator catches this and then just adds it to its own error array and allows us to read that list of errors it caught, so that is what happens behind the scenes you could say. Now here we have a look at an error and synchronous code throwing an error which we can handle with try catch. Now we also have async operations that can fail of course and such operations when using promises are handled with then and catch and that is what we can see a lot in our code. Here where I do something, where I find a user, I have my then block where I handle the case that the database operation succeeded, then I here still have my custom case to see if we did get a user because the database operation can succeed even without retrieving a user but I then also have a catch block here where I catch any errors that happened. So here for example that would be the catch block related to my find one method. So if the database operation fails because we don't have read access because the database server is down temporarily, anything like that, then we make it into this catch block. So this is try catch, just with async code you could say, then is your success case and catch allows you to execute code if that fails. Catch by the way collects all errors that are thrown by any prior then blocks, so if we had more than then block in our chain here, catch would fire on any error thrown in any then block or any operation executed in a then block, that's just a side note. So this is how we can work with errors, how we can handle them. I'll now get rid of the playground file, you find it attached to the video if you want to explore it a bit more, let's now see in our application how we can improve error handling.

**Lecture 309**

**Throwing errors in Code**

Let's start in app.js and there, I have some code where I do get my user out of the session and then I find the user by ID in a database and then I store the found user in my request object so that for the entire request, I can access that user object, it's a mongoose user object and I also catch any potential errors that might be happening. Again because it's really important, this catch block will not fire if I don't find the user with this ID, it will only fire if there are any technical issues you could say, if the database is down or if the user of this app does not have sufficient permissions to execute this action. Now we kind of have our own error handling in place and again we're not working with technical error objects here but here I am checking whether we do have a session user and I do have a solution for the case that this is not the case because if I would not add this check, then I could try to find a user without the session object existing and that would then crash our app. So it makes sense to write clever code that avoids such scenarios and tries to only execute code that will succeed. It might of course still fail and for some reason, we might still not find that user even if we have it stored in a session, maybe because the user was deleted in a database in-between. So it would make sense to also check for the existence of user or for the opposite, so if user does not exist and if it does not exist, here we could also return next without storing the request user. So just that we are super safe that we don't store some undefined object in the user object but that we continue without the user instead if we can't find the user. Here in the catch block, logging it is not really useful though. It will make more sense to throw a new error here where we simply wrap the error object we get here. Throwing this error has a significant advantage which I will show you in a second but with that, this looks good. We added an extra check and if we do have some technical issue, we throw a real error and as it turns out, expressjs gives us a way of taking care of such errors, that is why I'm doing it like this. Alternatively, we could of course also simply call next here to continue without request user being set or anything like that but I want to throw an error because we had a technical issue connecting to our database and that is something that might be a bigger problem than just a non-existing user.

**Lecture 310**

**Returning Error Pages**

I added this throw new error in app.js and to show you what it does, let me go to another place, to admin.js, to the controller and change some code there. In there we of course have plenty of code to for example add a product and we do validate the user input which is also a form of error prevention or error handling, we do validate that input in admin.js in the routes folder, there we added all our validation logic. Now in admin.js, I'm extracting my logic. Now let me force an error by quickly looking into my database with compass, the graphical user interface we used before for interacting with the database and in there, in the shop database, let's go to products and let me pick up product ID here. Now when creating a new product, I will also add \_id and I will temporarily import something from the mongoose package because I want to create a new object id here and I can use mongoose types object id here, pass my object id and add the new keyword and now I will basically create a new product with an ID that already exists and therefore this should fail. Now let me start my application and let's quickly log in with a user here and add a product and here you can enter anything. Let's click add product here and now it will fail, you've got this infinitely spinning refresh icon and if you have a look at your code or at the terminal where you ran npm start, you will see that our server crashed because mongodb threw an error with a duplicate key problem which makes sense because we added a product with an existing ID and therefore everything crashed. Now of course this is a constructed scenario but just because I want to ensure that something goes wrong at this point because now I want to show you how you could handle such a problem. Now we are logging that error and I can confirm that this catch block is called by adding console log an error occured. So now if I save this, obviously our node server restarts, let's add a product again or let's try this again, it will fail again of course but if I click add product now and I go back to my code, we see an error occurred here, so this proves that this catch block gets fired but that's great because this means that we now have a chance of handling that error. Now what we could do here of course is we could return some 500 error page, we could also redirect the user to any other page or we could render the add product page again, we could even keep the old input and that might make sense if we expect this to be a temporary issue. So of course we could take our code from validating, copy all that and output it here or use it here in the catch block, set a status code of 500 which is a code indicating that a server side issue occurred and then we want to return added product with add product page title, add product in the path, by the way it should be fixed up there too, minor thing but make sure that the navigation item is selected. Editing set to false, has error can be set to true, the error message we can put our own error message here, something like database operation failed, please try again. Validation errors can be an empty array, we don't want to put a red border around anything and we want to keep the user input. If we save this and everything restarts, let's try this another time. So let's add a new product here like this and if I click add product, indeed now we have a better error handling because now I am returning that same page, I give the user some error message and the user can try again but of course this here will keep on failing because we have a fundamental problem in the code. But if that was a temporary network issue, this might be a fine way of handling this. Sometimes you got bigger problems though and you don't want to use this solution, you don't want to return the same page again, instead you really want to show an error page to show the user something bigger is wrong, we're working on it but for now you'll probably not be able to continue. And for such scenarios, I'll add a new view next to my 404.ejs file, I'll add a 500.ejs file. We actually already have a view for error handling, the 400 page, 404 page, now I'll add another page that will render error messages for bigger technical issues, not for not found routes. I'll copy the code from 404, include it in 500 and there I will say some error occurred and there may be some paragraph, we're working on fixing this, sorry for the inconvenience, something like that. You could add anything you want here. Now we have that 500.ejs route and that might be something you also want to do, in cases where you are not sure if you will be able to handle this otherwise. Now to return this 500 page, I'll go to my error.js controller and I'll duplicate my get 400 route and add the get 500 action here, with status code 500 we render the 500.ejs file with a page title of error or whatever you want, path can be /500 and isAuthenticated, yeah we use our session information for that. Now we have get 500, we just need to add a route for that now. So let's go to the app.js because in there, I use get 404 for every page, for every middleware that does not get handled ahead of time. Now there is one extra route I will add with app get/500, here I want to use error controller get 500. So now I define a route like this, /500, I now just have to make sure that we do redirect to that route when a 500 error occurs and we want to show that page and therefore in the admin.js in the controller, in the catch block, I will indeed redirect to /500. And now if we save that and I try submitting this again, I end up on my 500 error page and this can be a decent way of handling this for bigger problems.

**Lecture 311**

**Using the Express.js Error Handling Middlware**

So we are redirecting to the 500 error page in this catch block and we can do this in any catch block where we want to ensure that we do handle errors. Now the problem is we'll quickly duplicate our code a lot because we've got a lot of catch blocks where we interact with the database and in every catch block here, we should redirect to the 500 page, at least in a lot of them, we would want to do that probably because for all these database failures or permission problems, it seems to be a bigger problem so returning this page might be a good solution. But replicating that code all over the place is rarely what we want to do, what we can do instead is instead of redirecting here, we can throw a new error, this built-in error object with the built-in throw function or keyword. We do that inside of the catch block, in this case here, of the post add product action. So here I throw a new error and I can throw a new error with a bit more details by first of all creating an error object with new error, I can pass my error message like creating a product failed or something like that or simply keep that error object which also has a message which you get returned by the catch block here. So I'm creating my own object which is built up on that now and then we can also add a new field here, http status code and set this to 500 for example and now here's something cool, we can return next and pass the error as an argument to next. We didn't do that before right, before when we called next like in app.js here, we simply called it without arguments to let the next middleware take over. Well when we call next with an error passed as an argument, then we actually let express know that an error occurred and it will skip all other middlewares and move right away to an error handling middleware and that's the middleware I'll define now. So next error is the trick here with an error object being passed instead of throwing it and now we can go to app.js and there at the bottom, let's add a new middleware with app use and normally this could never be reached because we have our catch all middleware down there but there's a special type of middleware which we haven't seen before. All middlewares we added thus far, for example this get 404 action in the error controller, all these middlewares use three arguments, request response and next. Express also knows a middleware with four arguments, a so-called error handling middleware and there, the first argument will be the error and then followed by the other three arguments. Now express is clever enough to detect that this is a special kind of middleware and it will move right away to these error handling middlewares when you call next with an error passed to it, so it will then skip all the other middlewares and move to that and therefore here I could now render my 500 page or simply redirect to /500, I could do that. And now if I save that and I go back to my application and I try adding that product which will still fail because I didn't fix that issue, I load my error handling page but now with this central error handling middleware. Now that we added this error handling code, we can of course also repeat this in other places and you could refactor it into one function of course which you then just call in all these places, for example to also send an error here where we fail to edit or load the data for editing a product. We can force this to fail by manually throwing a new dummy error in here, again this is of course now kind of a set up scenario, you would never write this but this allows you to force this to fail, it would be kind of hard to force the database to fail right now because most of the time thankfully it works. But now I have this dummy error and hence if I now edit a product, I also see my 500 page because I throw this dummy error which I'll of course remove now because it does not make any sense to add this but then we made it into catch and there we created this error object and we called next with it and that again will then trigger this special error handling middleware. Now in case you're wondering what the status code is doing here, well in this scenario, I'm always redirecting to the 500 route here but of course we could have a different scenario where we don't redirect here but we want to render a page instead or we want return some json data, something we'll do later in the course and then I would want to set my error http status code code here as a response code. So this is not the solution we're using here but this is what you could do and I just want to show that you can pass extra information with the error object so that you can use it in this central error handler here. Now please note, the this error handler will not execute for 404 errors. There we still handle this manually because technically, the 404 error is simply just a valid url which we catch with our catch all handler there where we then just happen to render the 404 page, it's not a technical error object that gets created at any point here.

**Lecture 313**

**Using the error handling middleware correctly**

Now that we learned about this express error handling middleware which we can use, which we added, what exactly does throw new error in this catch block, in this middleware in the app.js file do for us? I mentioned that this will become important. Now like this, it'll unfortunately not do anything. Let me start up our server real quick again and let me ensure that we do get an error by simply throwing a new error, a dummy error here, again to simply simulate that something goes wrong. I throw it here, if I throw it in a then block, the next catch block will catch it and handle it, now let's see what this does. Now to see the effect, I have to log in to ensure that we do get a user session and we can already see this crashes here, it loads infinitely because I have an error here, my dummy error. So this is now not doing anything, the app is still crashing. So one important takeaway is throwing an error here does not lead to our general error handling middleware being called and that is important. This is true because we're inside some async code, we're inside a promise here, we're inside a then or a catch block. If you throw errors there, you will not reach that express error handling middleware. The interesting thing is if you would throw an error outside of async code, so in a place where the code executes synchronously, so basically outside of a promise, then catch block or outside of a callback, so here if I throw my sync dummy like this, if we do that, so this is now throwing in a normal function, not nested inside a promise or a callback or anything like that. If I now reload this, you'll see that it tried to load the 500 page but it still failed. The reason for this is really simple, we have our middleware in place here where I retrieve my user and there I throw the error but this executes for every incoming request. Now when we redirect here, we do send a new request so we kind of enter an infinite loop here, we execute this again, it throws an error, we go to the error handling middleware, we trigger a new request. A solution here can be to simply not redirect to 500 but immediately execute our rendering code, so this code in our get 500 controller action and we could absolutely do that. We could render our error here instead of redirecting, we could render our page here and if we do that, you will see that now if I reload, I just get a problem regarding my csrf token because that can't be generated based on the incoming request because the request basically has issues here in our middleware before we set up the csrf token. So the solution for that would be to switch the order to make sure we set this token before we actually do something with the user which we tried to fetch, so now this works and now we get an error handling in place. The interesting thing is not just that we had to switch the order though and that we should avoid infinite loops but the interesting thing here also is that here I'm just throwing an error and we still reach this global error handling middleware. The reason for that is that in synchronous places, so outside of callbacks and promises, you throw an error and express will detect this and execute your next error handling middleware. Inside of async code, so inside of then, catch or callbacks, this does not work however. Inside of that, you have to use next with an error included. So this is then detected by express again and this is what we used in the other files and inside of async code snippets, you need to use next wrapping that error, outside you can just throw that error. And so if I now uncomment this, you will see that if I try to reload my page, I get this error no matter what I do because I still have this dummy error being thrown in my promise, hence the catch block executes, hence this line executes. So now I will get rid of that but I will leave this catch block as it is and you have to understand that for one, you should avoid infinite loops triggered through your error handling middleware as we had it initially here and second, that you can throw the error in synchronous code places like this one but inside of promise, then or catch blocks or inside of callbacks, you have to use next around the error.

**Lecture 314**

**Status Codes**

Now there's one thing which I also want to cover in this module and that are the http status codes, which codes do we have and why do we use them? Let me first of all start with what the codes are therefore. The codes are simply extra information we pass to the browser which helps the browser understand if an operation succeeded or not. If you're writing an application with a lot of client side javascript or a mobile app and you will fetch only data instead of complete html pages, something we'll do in the rest module later, status codes also allow you to understand if an error happened, which kind of error because you typically map certain kinds of errors to certain kinds of status codes. For example you have 200 status codes, most importantly 200 and 201, these are always success status codes, they indicate that the operation simply succeeded. You have 300 status codes which simply indicates that a redirection happened. You have 400 status codes which show you that something happened because an error was done by the client, for example incorrect data was entered into a form, well then we return this 422 error code if you remember correctly and we have 500 status codes which indicate that a server side error occurred. Now here are some examples, you can find a full, a link with a full list in the lecture after this one. For success cases we have 200 and 201, the difference is that we use 201 typically when we created a resource on the server, so in the database, it's not a must but it's a pattern you can use and you see often. 300 like 301 simply is a code that is used in combination with redirection to inform us whether for example this resource moved permanently or just temporary thing, whatever. The 400 codes, there we have 401 if we are not authenticated, 403 which you could translate with not authorized so you might be authenticated but you were still not allowed to do that specific operation, 404 for a page that's not found, 422 which we often use for invalid input and a couple of other error codes and for 500, the most common one is 500 which indicates hey there was a server side error but you got also other codes for timeout and so on, again in the next lecture you find a full list. Now let's quickly scan through our code. If we have a look at our code, let's start with the admin controller, we rarely set a status code. The default then always is 200 which is fine for this case but for creating a new product, it would be better if we not just render 422 when validation fails which is correct but that when we succeeded, we also return 201. The thing just as if we succeeded, we redirect which will use a 300 code but later when we dive into creating restful APIs which is a different kind of node app where we don't return html but just data, there you will see the 201 code coming back, here it doesn't really make sense. Now forget added product again, we're fine with 200 and we're fine with 200 here. If we did fetch data and we do load the page and if we try to save or update a product, then we get our validation error with 422 and if we succeed or in all cases here, we redirect which again will use 300 automatically. So all these codes here are pretty fine, pretty decent, one of the most important things, we're in good practice at least is that we do use 422 for validating. Now we also have isAuth.js where I do redirect when we are trying to do something where I'm not logged in. Now again since I'm redirecting, I'm sending a 300 status code but of course we could add status 401 here to kind of also make clear what the problem is but it will be overwritten with a 300 code due to redirect, so it does not make a lot of sense and that is fine here. Later again when we have a restful API where we don't redirect because we don't route around on the server, then we'll use that 400 code. And for that reason that we now most of the time redirect or directly render a page, we don't set that many status codes here, we'll see them later again as I mentioned and we see some codes at least, also in error.js where I set 404 and 500. And these status codes not mean that our app crashes, that's important to understand, instead if I do enter some invalid route here, I do get page not found and if we open the chrome developer tools with the network tab and I reload, we see that here, this 404 code can be seen, it's also marked as read because chrome is intelligent and detects that anything which is not a 200 or 300 code is an error but error does not mean that it crashes. This still renders a valid page in the end, we just pass that extra information of hey here's the page but you see that page because something went wrong and chrome knows this too, for example tell us in the developer tools and later when we create that restful API I referring to, we'll also benefit from that because there we do have a more direct interaction with our requests because we don't render new pages all the time and then we can get useful information out of these status codes. So that is http status codes and how they are related with errors, your key takeaway is a status code does not mean that the request failed or that the app crashed. We had some problem and now we're returning information with the problem to the client and that is also a way of gracefully handling errors.