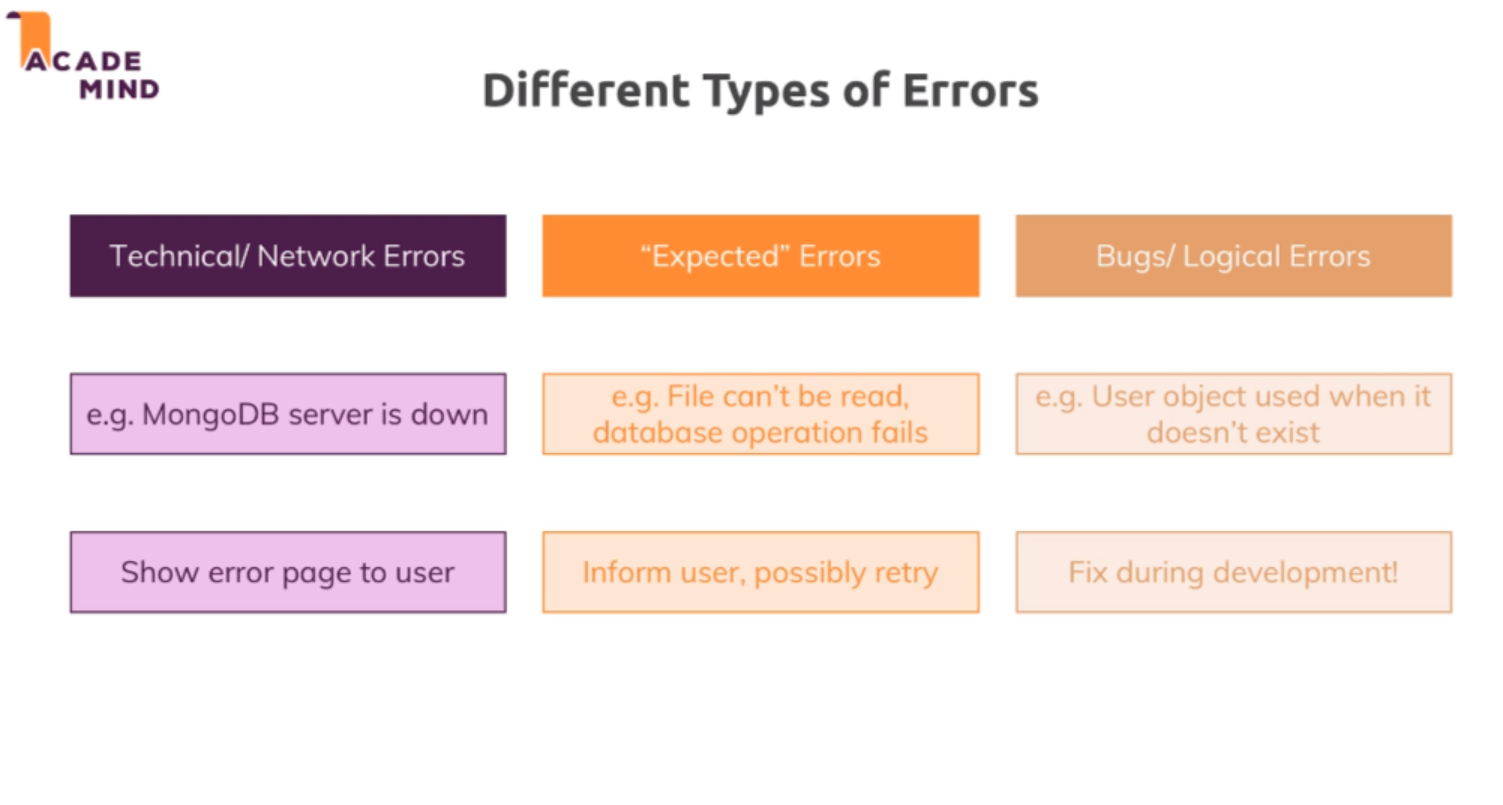
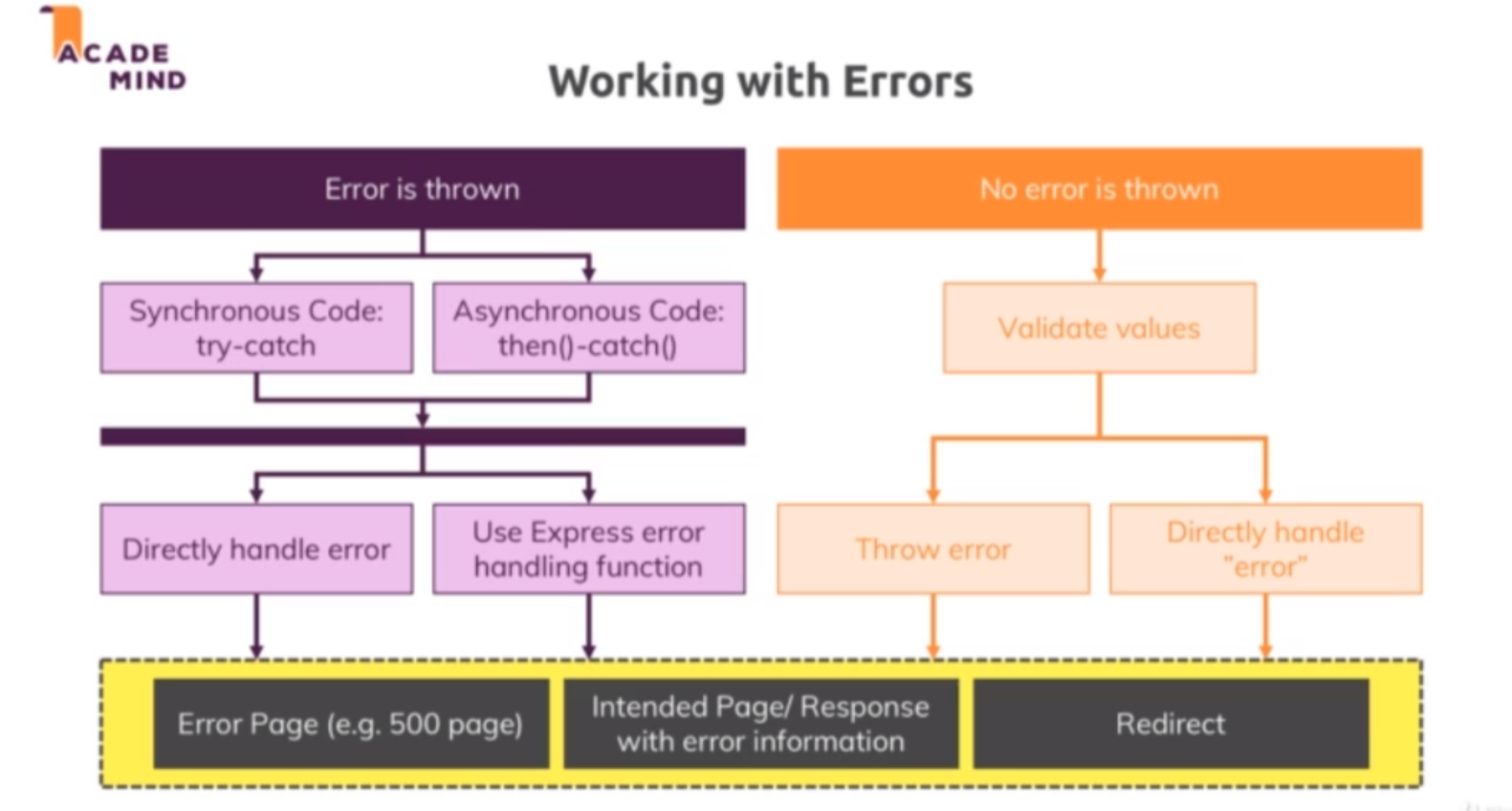
**Lecture 306**

**Types of Errors and Error Handling**





* 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

**Lecture 308**

**Errors – Some Theory**

* Refer code 01-errors-some-theory
* 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

**Lecture 309**

**Throwing errors in Code**

* Refer code 01-throwing-errors-in-code
* App.js 🡪 throwing error in middlware 🡪 line 63
* 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**

* Refer code 02-returning-error-pages
* Controlers/admin.js 🡪 postAddProduct method 🡪 if try to add the \_id of an existing product while creating product model ;while trying to save the product we get database exception🡪 duplicate key problem. But it is not properly handled in our code
* One way to handle this is show an error message in the existing page.
* Other way is to show another page , a 500 error page

Views/500.ejs

Controlers/admin.js 🡪 postAddProduct

Controllers/error.js 🡪 adding get500 method

App.js 🡪 adding route for /500

**Lecture 311,312**

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

* Refer code 03-using-the-express-error-handling-middleware
* Controllers/admin.js 🡪 postAddProduct method 🡪 throwing error in catch block and calling next
* App.js 🡪 registering error middlware
* Added the same code in every catch block
* Handling the error in the previous way causes a lot of code duplication, because we have to write the redirecting code in every catch block
* Instead we throw an error in the catch block , and call the *next()* method with that error passed to it
* 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.
* If you got more than one error-handling middleware , they’ll execue from top to bottom. Just like the normal middeware behaviour

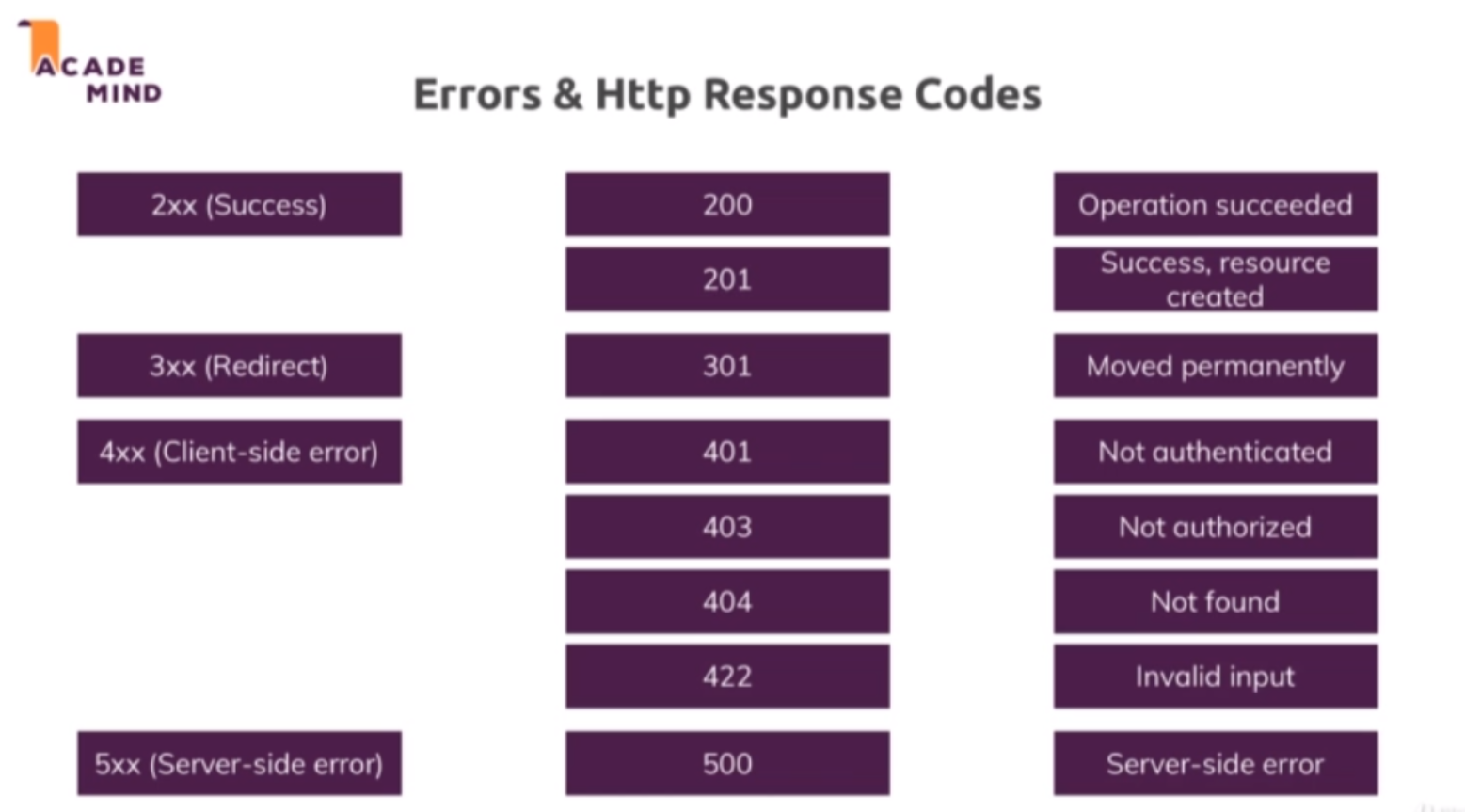
**Lecture 313**

**Using the error handling middleware correctly**

* Using throw new Error () in synchronous code causes the error middlware to be executed correctly
* But in async code , e.g inside then or catch blocks , we have to call next(error) to move the control to error middleware. If we simply throw error using throw new Error() in async code , the error middlware will not be executed
* Also don’t use res.redirect in the error middleware to goto a new page. In the code attached, we are manually throwing an error using throw new Error(‘Sync Dummy’) in the middlware which sets the user object inside the req. It causes the error middlware to be executed. Currently we have res.redirect(‘/500) in the error middleware . Executing it generates a new request. Since it is a new request , all the middlewares will be executed in order and when it comes in the middlware where we manually throw error , all these process happens again in an infinite loop.
* To avoid such situations , we should not use res.redirect inside error middlware. Instead use res.render() to goto a new page

**Lecture 314**

**Status Codes**



* Using res.redirect sets the status code as 300 by default
* 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.
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>