**Lecture 42**

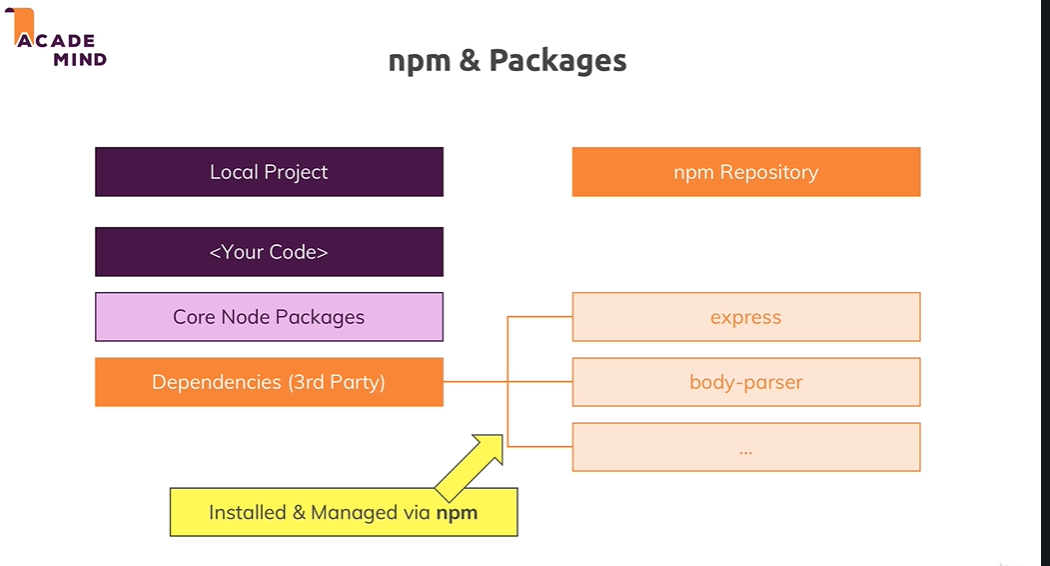
**Understanding NPM Scripts**

For this module I'm back in the application or the project we worked on in the last core section and you can find my state attached to this video so that you can start right at the same point as I do. So it is just the code we wrote over the last lectures and one thing we constantly had to do in the last section or throughout this course was that we always had to run node app.js to start our well app or our node application with that file. Now this is certainly ok and not too much work but it actually is possible to define some scripts in a nodejs project that can help us with tasks like this, we can also use them for other tasks but especially for this, they can be useful right now and for that we have to use a feature we didn't use thus far. We have to use npm, npm stands for node package manager and it is installed together with nodejs, so you don't need to do anything, you got this installed already. Now we will also use npm to install some additional third party packages to our project soon, so packages that are not included in nodes core, so not part of nodes core modules but we can also use npm to initialize a so-called node project or to add some extra features to it to be precise because we obviously already got a node project here but now in this project, in a terminal navigated into this project, you can run npm init. Now this will not delete your code or anything like that, no worries, simply hit enter and you'll be prompted with a couple of questions here. It'll ask you first of all for a name of your package and for now you can simply translate this with project name. Now you can pick any name you want, the part in the parentheses here always is the suggestion, the default it will pick if you don't choose your own name, so if I just hit enter here, it will take that default. So you can basically do that for all these questions, for the description, you can leave that empty then it will have no default text either but you can also enter some text like complete nodejs guide, this is what this course is and then define an entry point with fs for example. You can leave the task command empty, leave that empty, keywords you can choose some if you want, you can put your name into the offer field though this is also not required and you can always choose a license but if you don't plan on sharing this project publicly, this also doesn't really matter. So with this what you get is this package.json file and there you also see all these settings or configurations you just set up and you can of course edit them there too, so if you had a typo in your description, you can just edit it here. Now this is using the json format which is a special kind of data format which basically looks a lot like javascript objects and it pretty much is based on that, there is one important thing to understand though, the keys are always put between double quotation marks and so are the values, except for numbers or arrays or true or false which are not put between these but that's too much for now, we can ignore that for now just so that you understand what you got there, it's basically a configuration file for your project. Now what does this configuration file give you? Well let's clear the console here, with this configuration file, you'll see that we got a scripts section there which has one default script that won't do anything for now, you can add your own scripts here and I will tell you how to execute them too of course. For that, let's add a comma after this test script and add a new script name which has to be put between double quotation marks and there, let's name it start. Now start is actually a special script name as you will see in a second, so make sure to type this correctly and then between the quotation marks, you type a command that should be executed. So this is a command which you could also type down there and there we always have to type node app.js, so let's now put that between these double quotation marks here node app.js, like this. With that save that file and then you can run npm start. Start is a reserved name and this will always look for such a start script here. And if you do that, it will as you can see here just execute node app.js. So it does the same you had before but now you can always just well run this command instead of running node app.js. Not that much of a saved characters but a few at least and it's also a good practice because if you ever share this project, it's pretty common that people just have to run npm start and that they don't have to guess which of your javascript files is the entry file. So you can quit this with control c of course as always and I mentioned that this would be a special script name. You can add more scripts, also without using a special name, you can indeed choose any name you want, just make sure to always wrap the name in double quotation marks and that it does not contain any blanks or whitespaces. So for example we could have start-server. Now this can also run node app.js, so it will do the exact same as this script and therefore it's of course redundant but I want to show you something and now if you try running npm start-server, you'll get an error. You basically get an error that is not a known command and indeed it isn't because just typing the script name here will not work, start just was the special case as I mentioned. Indeed for normal scripts with their own custom names, you have to run npm run and then your script name, so npm run start-server will now also start the server. Now as I mentioned, this is of course redundant, npm start is way shorter but I want you to understand how you can add your own scripts. Now if you worked with something like angular or react or vue or any modern frontend development workflow, you will have seen that you use such scripts a lot to trigger build workflows for your projects for example and indeed you can use that for all kinds of tasks to want to execute but for now we'll not dive deeper into that and if you haven't worked with angular or react, it's also no problem, you will see what I mean later in this course when I explore node's functionality as a build tool a bit more. For now let's just use that npm start script to start our application conveniently.

**Lecture 43**

**Installing 3rd party packages**

So in the last lecture, we added a script to start our application and this is nice to know that we have this scripting functionality. Now with such a package.json file available, so therefore with this being a managed node project you could say, we can also do one other very important thing, we can install third party packages because a typical node project looks like that. You have your local project with your code obviously and you use a lot of core node packages like the fs package or the http package we're already using but often this does not suffice and in the next core section, we will install one big third package actually because you typically have such dependencies third party packages. So you want to use some functionalities some code which you didn't write on your own but which is also not included into nodejs. Packages could help you with parsing incoming requests, validating user input, anything of that kind. Now we will use express in the next lectures or in the next module to be precise, body parser is another package we'll use throughout this course and there are thousands of packages available that offer all kinds of utilities you can add to your projects so that you don't have to reinvent the wheel. These packages are available through the npm repository, that is a cloud package repository where all these packages live and you can conveniently install and manage them via npm, remember that tool that shipped with node. And this is exactly what we will do now to install a first little utility package that will speed up our development workflow. Because right now what we have to do is whenever we change our code, we have to quit the development server with Control-C and restart it, right, so if we have it running with npm start now, remember this is our new command to start the server and I would change something here, like for example here I fixed that /head right, whenever I do this I have to save this and for my change to have an effect, I have to quit the server with Control-C and restart it. Now this is a bit cumbersome because we just want to be able to type and then hit save and it should automatically restart and right, that would be a great workflow during development and to achieve this, we have to install a third party package that gives us just this functionality. Now how do we add such a third party package? We do that with the help of npm and there, we get the install command, so just as we had run to run one of our scripts, install installs a third party package. Now how do we install it? First of all we have to know the package name and if you're wondering well how do I know the package name, well that comes with courses like this, experience or by simply googling for certain problems which you want to have solved and all of a sudden you find a thread where some package is mentioned. Now I can tell you for this auto-restart mechanism, there is a package called nodemon, written like this. By the way you can always search for npm and then the package name if you know that name and you will find an entry on npmjs.com. Now this is the package page basically where you find a description, installation instructions, usage instructions, how to configure it because most packages give you an easy way of using it and then always give you configuration possibilities. You'll see how popular the package is, what's version it's using, if there is an exernal little home page, where the source code can be found if it's open source. So you find a bunch of stuff here, pretty useful, you also see how many versions are existing and by default, you will always install the latest version by the way but let's go back to installing it before we dive deeper into this whole npm thing. So we want to install it and this command would install it but don't hit enter yet, you can define how this should be installed because packages which you install can be divided into development packages, so packages which mostly help you during development and production dependencies, so packages that helps you for the app as it's running on a server, for example nodemon would be a development dependency because we only use it during the development process, once we install our app on a real server we don't need it there. The real server which is running somewhere in the Internet of course shouldn't restart and it also doesn't have to because we'll not change its code dynamically. And you can basically tell npm which kind of dependency this is, this does not make a huge difference and you can omit the setting but it helps you understand which package is used for what. Now you do add this by adding --save-dev, if you had just save like this, this would install it as a production dependency, so a package which we really use and use in our code and work with and with this we're indicating that this only adds something we used during development. There also is a third option by the way, -g, we'll not install it in this project but globally on your machine so that you can use it anywhere. Now let's first of all install it with save dev, like this --save-dev. Now what this will do is it will download it from the npm repository and install it into this project, so not globally on your machine but into this project. So now you see you get a report here that it finished successfully, what it did and it gives you a couple of new things in your project. It gives you that node modules folder, the package log json file and it updated the package.json file. There we see that the new dev dependency section was added and that stands for development dependencies, as I said you can differentiate between different dependencies, we'll see production dependencies later too and there you see that nodemon was installed and which version was picked. Now regarding that character here, well this basically defines how this package will be updated if you rerun just npm install, without defining an extra package name because this command standalone will simply go through all your packages mentioned in package.json and install them and it would automatically pick a later version if available but more on npm and packages can also be found in a separate module later in the course. So this is basically how we now install this and the question is where is it installed? Well that is the node modules folder and actually that is a huge folder as you can tell. The reason for this is that for one we got nodemon in there, if we look for n, we see it here. Now this is basically the source code of the package or the build version of the package we installed and this package simply happens to have a couple of peer dependencies, you can see them here and here. So we got a bunch of dependencies in there and these and their dependencies are also installed, that is why you could end up with quite a big node modules folder but you can always delete that node modules folder if you need to free up space. Now you can't use that package but you can then rerun npm install if you start working on that project again and it will re-install this package and all its pure dependencies and therefore recreate the node modules folder, this is how packages work in node projects. So you need that node modules folder while still using the packages but if you're not working on the project, you can delete it if you want, if you need the free space and then just remember to rerun npm install once you are working on the project again. The package log json file by the way just stores the exact versions I installed today so that if you share your project with others, they can actually get these exact versions too instead of the latest versions but again, more on npm in a separate module.



Global Features vs Core Modules vs Third-Party Modules

The last lectures contained important concepts about available Node.js features and how to unlock them.

You can basically differentiate between:

* **Global features**: Keywords like const or function but also some global objects like process
* **Core Node.js Modules**: Examples would be the file-system module ("fs"), the path module ("path") or the Http module ("http")
* **Third-party Modules**: Installed via npm install - you can add any kind of feature to your app via this way

**Global features** are **always available**, you don't need to import them into the files where you want to use them.

**Core Node.js Modules** don't need to be installed (**NO npm install** is required) but you **need to import them** when you want to use features exposed by them.

Example:

const fs = require('fs');

You can now use the fs object exported by the "fs" module.

**Third-party Modules** **need to be installed**(via npm install in the project folder) **AND imported**.

Example (which you don't need to understand yet - we'll cover this later in the course):

1. // In terminal/ command prompt
2. npm install --save express-session
3. // In code file (e.g. app.js)
4. const sessions = require('express-session');

**Lecture 45**

**Using nodemon for autorestarts**

So for now we got that nodemon package installed and it will just work because it automatically installed all the dependencies this package needs in turn. So how can we now use it? Nodemon is a utility tool and it allows us to run our application, our node application through this package here which will run node apps.js in the end but which will also watch our files for changes and restart the process for us if we do change something. So we can simply change start here, so the node app.js command to nodemon app.js and this will look for a nodemon tool which it will find in this project because we installed it here. As a side note if you were to run nodemon app.js down there, you would get an error that this command is not found because it's only installed in this project and not globally on your machine but the terminal will try to find this globally. Here it will work because this will look globally. So if you now run npm start, this will simply start the node server and output some extra information and if you now go to your routes.js file and edit something, let's just add an extra line and you save that file, you see it's restarting. And this of course is very convenient because now, we dont have to spend time on manually exiting and restarting, now is this done for us.

Global & Local npm Packages

In the last lecture, we added nodemon as a local dependency to our project.

The good thing about local dependencies is that you can share projects **without the node\_modules** folder (where they are stored) and you can run npm install in a project to then re-create that node\_modules folder. This allows you to share only your source code, hence reducing the size of the shared project vastly.

The attached course code snippets also are shared in that way, hence you need to run npm install in the extracted packages to be able to run my code!

I showed that nodemon app.js would **not work** in the terminal or command line because we don't use local dependencies there but global packages.

You could install nodemon globally if you wanted (this is NOT required though - because we can just run it locally): npm install -g nodemon would do the trick. Specifically the -g flag ensures that the package gets added as a global package which you now can use anywhere on your machine, directly from inside the terminal or command prompt.

**Lecture 47**

**Understanding different error types**

In the last lectures we worked a bit on the project setup and how we could use external packages which is super important and which we'll do a lot throughout this course. But now I want to have a look at something else which helps us during development and that is of course fixing errors, we need to understand how we can find and fix errors in our code because there always will be errors, you will never write a program that just runs perfectly. For this, we first of all have to understand that there are different types of errors so that you can understand how each type can be tackled. The first kind of error you often get or you sometimes getting your programs are a syntax error, this is if you have a typo in your code or you forget like a closing curly brace or anything like that, so you have an error which should automatically be thrown when you try to run your project. Most of the time, these are pretty easy to fix but we'll have a look at this and how we find and fix them in a second. The second category of errors are runtime errors, at least this is how I like to call them. Now these are errors which are not typos but where you try to execute some code which will just break when it runs and I'll show an example for this in a second too. The third kind of errors are logical errors. Now these are the most difficult ones because there, you will never see an error message, your app just doesn't work the way it should and it can be hard to find these but I will also show you some tool which can help you with that in this module. So let's now start with syntax errors and let's see how we can find these and fix these.

**Lecture 48**

**Finding and fixing syntax errors**

Syntax errors can be very annoying but most of the time, they're not that hard to fix. Let's add a simple syntax error, here in app.js let's simply remove the t in const, now all of a sudden, this is an invalid keyword, it doesn't exist, it's not a keyword and we already get some help in the IDE which is showing us something is wrong here. Now if we hover over this, we see that a semi-colon is expected and actually this is not that helpful and this is simply because visual studio code thinks so that this is a variable and since we have a variable in the same line, we should close this with a semi-colon and now it actually doesn't show an error. But let's save this and let's try running our app with npm start without that semi-colon we added. And you will see, it right away crashes and in there we see unexpected identifier const server. So in the end, whilst it doesn't clearly tell us that we forgot the t here because it's not smart enough to understand that this is the error here, we still see that the error seems to be stemming from this line and in such cases, as dumb as it sounds, you should simply take a closer look at this line and see what could be wrong there and you should quickly be able to see I forgot a t here. So these are syntax errors, other errors would be that you maybe go to the routes.js file and let's say you forget a closing curly brace here. Now again visual studio code does complain and does show an error over there, if we jump there we see at the end that the closing curly brace is expected. Now it's not expected at this point where it's showing this message but it's expected somewhere in the file and whilst it can be cumbersome, you should then check your block statements like this if statement and see if there are all closed correctly. In visual studio code, you also get some help because if you click next to such a curly brace, you'll see that this line gets highlighted and it actually shows you where it thinks that this is getting closed and this is far too much at the bottom here for example. Again if you run this, you will see that it crashes and there, you also see unexpected token and it points us at routes.js . and then you see the line number too. So here it points us at line 52 and it basically shows us the same place as visual studio code which is the wrong place but in such a case here, you should really take the IDE help, see that error message and then go through your file and see hmm where could I be missing a character or where do I have an extra closing curly brace or something like that and eventually you should be able to find this error and therefore of course fix it. So syntax errors should always result in an error message and then that can sometimes be hard to find them, often it's like a typo, a mistyped variable name of a variable that therefore doesn't exist or missing or extra characters but you can find them and it just takes some time to go through your file.

**Lecture 49**

**Dealing with runtime errors**

The next category are runtime errors and a great example for this can be shown with the res write method here. Now as I mentioned, you have to return here to prevent the execution of the code after this statement otherwise and this is something node specific of course, you would end a request here but the code execution would continue and eventually we would reach this line and this is no syntax error, this is correct but if we run our code here, seems to work right, there is no syntax error after all. But if we now visit our page, eventually it breaks here and this is the point where it should go back to your code and should find an error message there too. And this error message is something you shouldn't just take and paste into the Q & A section but actually read it. A lot of the error messages are indeed helpful, you just need to know how to read them, at the bottom you always find uninteresting stuff I'd say but you have to scroll to the top of the error message and all of a sudden, it should get more meaningful. For example here, you see the error code which already indicates that it's something with headers being sent and then here, you actually find a detailed error message, cannot set headers after they are set to the client and then you see that it was caused by a call to set header and unfortunately, the line numbers don't help you, here at least but then it does help you here, you see it's in the request handler and there it points at the routes.js file line 32. And now this is the point where you can dive in and see, ok I'm calling set header here, now since it's complaining that I can't set them after they are sent to the client, I should look well in the code before this statement because it looks like I'm finishing my response there for this example. And indeed at some point you should find this statement and see I do actually not finish my code execution after this statement which is not per se a problem but it becomes a problem if in the following code, I then work on my response again as I'm doing it here. So now this is the point where we can fix this by stopping the code execution or by wrapping this in an extra if statement which is guaranteed to not run if we make it into this statement. And now with that, we can of course restart our server or actually we don't need to, we got nodemon now so you don't need to do that and now if you reload your page, it will work again.

**Lecture 50**

**Logical Errors**

Now the third kind of error I want to discuss is the logical error and that is often the most difficult one to fix because it will not cause an error message, your app will just not behave the way you expect it to. Let's see here in the routes.js file where we parse our body, we're retrieving the message here and that is of course nice but let's say I actually do have the first element being stored in message here and keep in mind the second element which we previously had is the actual message the user entered because we're splitting that key value pair we're getting automatically and now we're actually storing the wrong element. So now if I save this and my server restarts due to nodemon, I can enter some test here and if I do this and I open my message text file, I actually see message there. Now this is a logical error because we got no error message but the app is not behaving the way we want it to. Now of course we know that we used the wrong index here because I just changed that but let's say you worked on that code for the first time and you just use zero here because you forgot that this should be one, so you didn't change that back on purpose but you really made this mistake. Now it's not super obvious to see and how can you now work on such errors? Well with the help of the nodejs debugger which actually has a great integration into visual studio code which I strongly recommend using. Let me show you how that debugger works then. For this, let's quit that process and let's select the app.js file now, that's important and now go to debug here in visual studio code in the menu and choose start debugging or use the keyboard shortcut you see there. Now with that, you have to choose an environment and there you should use nodejs and you will now see that extra bar here at the top which allows you to control the debugger, you see the red bar at the bottom that indicates that you're in debugging mode and you actually also have a debug console here where you now see that the debugger is attached and listening. Now what does this mean? This means that now you actually can look into your code as it's running but to really do that, you need to set so-called breakpoints. So let's go to the route.js file and let's say we assume it has something to do our error here with the wrong text being stored with this code snippet because ultimately, that is where we write it to the file. So what you can do is you can set a breakpoint in visual studio code by going to the left of the line numbers until you see that red dot and clicking there. Now we got a breakpoint there and what will now happen is that with that debugging process running, don't quit this, you can quit it by clicking the red square but let's leave it running, if you now submit some text here, it should automatically jump back and mark this line like this, highlighted yellow and kind of put this yellow arrow around our breakpoint. This means that the code execution now stopped here, it did not continue, it stopped and it stopped so that you can look inside of it and actually you can now analyze your code in the moment it's running. For example you can hover over your variables to see what's stored inside of them, so for example you see that the parsed body is this string which we're splitting in the yellow highlighted line. You see what's inside of the body you passed to concat, that it's an array with a buffer of length 12 for example. You can even expand this to look into it a bit more and find more details about the buffer, in case you're interested here. You also can go to view and then debug here, this gives you a special debug mode where you see the key variables you have available in your code right now, message is undefined because we stopped in that line where we would set it but it stops before it executes the line, so therefore message is still undefined at this point but parsed body for example does hold the value, well it has which you can also see if you hover over it. You don't just see local variables which are available in this function but block variables too, this is the variable which is always available outside of this function, it's this variable and you can also look into that therefore and you can generally click around to see some global values and the values they currently have stored and a couple of these things. You also can define watchers here, for example you can click a plus here and then watch message and hit enter and now you will always see that here, you also see it up here but if you ever close that, you can define some variables you're particularly interested in and watch them whilst you go through your code. And you also see your breakpoints down there, you see all the breakpoints you set, you can uncheck them here to not stop execution the next time they are reached and so on. You also see the call stack which looks very cryptic but what in the end just shows how the process went through your code and you can click on these different parts to see where actually this code which belongs to the code that was executed can be found and not all of that is code you wrote, a lot of that is core nodejs code. Ok so that's the debugger in a nutshell. Now to work it and to continue with your code execution, you can use that panel here at the top. You can resume code execution with the play button but we don't want to do that instead we want to step through our code step by step so that we can see when it fails or where it goes wrong. And you can do that with this button which steps to the next line basically or with this button here which doesn't just step into the next line but actually even goes into functions like this one. So if you click here and again, now all of a sudden you're in the write file function defined by nodejs and you can step out of there with this key. Now you see if you continue here that this wasn't too helpful but that we can now navigate like this in our code. So now let's make this more helpful and for that, let's try this again and use that debugger properly now. We'll do this in the next lecture.

**Lecture 51**

**Using the Debugger**

So I still have my debugger up and running and now that I gave you a brief overlook of how it works, let's actually send another message here. Hit send, it breaks again because I didn't remove the breakpoint and let's now fix that error we have. I showed you how we can navigate, so let's click this line here or this button to move to the next line. Now it registers this code and here you also see that it doesn't immediately execute this function because if I click that button again, we jump to the end of this function here and that makes a lot of sense because as I mentioned, this function is just registered by nodejs to be executed in the future once the file is read. So for now, code execution will continue with the other lines. Now the problem we want to solve is not inside of here though but if you want to get notified once a part in here is reached, you can simply add a breakpoint there, resume execution and it will break once it reaches that, so once that callback is triggered because the right file operation is done. But our problem of course is the message, right, and the message is not available here anymore, so this analysis is actually not useful to us. So let's remove that breakpoint and resume and let's again run this one more time and now let's not resume execution but look into message and we see message is message and that looks wrong, we see that parsed body is message equals and then our value. Now since this is what we stored in the message constant, we now can tell pretty clearly that our error has to be stemming from that part here because we clearly have our value in here but then we seem to be extracting the wrong piece. Now split also does its job and you can even use the console here to for example run some commands, you can run parsed body split equal sign here and see what this returns, so you see you get an array with message and tist because I mistyped test. Now this clearly tells you ok so split is working, I'm in theory getting all the values I need, so the only thing that can be wrong now is the value I'm extracting here and indeed we see that message, holds message which is incorrect, message the first element in the array so I seem to be extracting the wrong element from the array and indeed we are extracting the first one when we should be extracting the second one here. So let's now change it to a one and now we get this error fixed. Obviously this is a bit of a constructed example but this is just to show you how you can use the debugger to go through your code step by step, that you have to keep in mind that node doesn't execute line after line but as I explained multiple times now that it registers callbacks which are executed sometime in the future and which should therefore also have to control with breakpoints if you want to look into them. And with that, you should have some powerful tools for finding and fixing errors in your code.

**Lecture 52**

**Restarting the Debugger Automatically after editing code**

Now we worked with the debugger a bit and after this lecture, you'll also find a lecture with a link to a more detailed explanation of the debugging capabilities Visual Studio Code gives you. Don't learn them by heart but feel free to explore them to get the most out of the debugger for you. Now there are two things I want to show you or I want to draw your attention to, one is something you already saw. Let's add a breakpoint again and again submit this, I already showed you that you can execute code here in that debug console at the bottom, so you don't see the console logs here, you can also execute code here. For example you can type variable name here, so something which is available in your code at this point of time, in general something which you can find in local or in block here or in global, so the things you find here like message, you can also type that down there and hit enter to print its value. And this is of course not really useful for printing because you can see the value on the left too but as I showed you for parsed body, you cannot just print the value, you can also run operations on it that will not affect your code as it's running but that will allow you to look into it or try out some transformation before you add it to your real code. So whatever you run down there does not affect the code you run up here but it is nice for you to understand your code. The other thing I want to show you is that for now at the moment, if we change something in our code here, let's say we add a blank line, our debugger doesn't restart but with nodemon we actually have a package that does allow us to restart. So it would be nice if the debugger would also restart if we change our code otherwise it just behaves differently than the rest of our app and we have to restart it manually. Now to restart it, let's go back to the explorer view for a second, we have to go to debug and then add a configuration for nodejs. This adds the .vs code folder with the launch json file and this allows you to configure debugging for this project and how it behaves. You can click on add configuration to see some demo settings you can add but you can also just type in there and one setting you can add is restart and you can set it to true, you just have to make sure that you also add some other fields. For example you have to make sure that nodemon is used and for that, you set the runtime executable not to node which would be the default but to nodemon here. So now it'll use nodemon and it will restart the debugger when a change is detected, so that not just the server is restarted but also the debugging process. Here by the way in this configuration, you can also define that it should always start with the app.js file so that you can even have the routes.js file selected while starting debugging and it will still go for the app.js file which is more convenient than always selecting the file you want to debug first before starting the debugger and you always have to pick the app.js file because that starts your server. So you can't say I'm going to look into the routes.js file so I'll start with that, it always has to be the app.js file because you always have to start the entire server, so here you can define that it will always pick that no matter which file you have currently open. You can also change the console where things are logged to to the integrated terminal which is the normal terminal. With this if you save all of that and you now start debugging, it fails though and the reason for this is that it will not use the local nodemon but it looks for it globally. Now to add it globally, you have to run an install nodemon -g and now important, on Mac and Linux you might need to add sudo in front of this, on Windows you don't, to get the right permissions to install this. With that you might be prompted for your password and now it will install nodemon globally which makes sure that if you ever run nodemon in the terminal now, it will not fail with a not found error but instead it will find it and therefore now you can also run this debugger here which will use the global version and you'll see it now opens terminal, starts nodemon and if you now add a breakpoint somewhere and submit this again it still works as before, just important, you now see the terminal here logs all the things. You can still use the debug console to output messages though, so you can still work with this as I showed it to you, so this is not stopped from working, this still is something you can use but in the terminal, you get the normal output and you have to use the terminal because if you now change something, it restarts the debugger and node and these are two separate processes and if you stop the debugger, nodemon has to quit separately or has to exit separately and you do this by hitting control c here and this couldn't be done in the debug console which is why you have to funnel this to the terminal. So that's just something to keep in mind, you have to stop that process separately which you can do from the terminal which is why if you are using that nodemon process, you should use the integrated terminal and you can read more about that in that detailed visual studio code debugging article but now you have a setup where you have a debugger attached to your program that will restart if you edit something, where the server will restart if you edit something and where you have now all the tools you should need to debug your node app and hopefully solve any errors you get. Now one important note, for the rest of the course I'll mostly use npm start and not have the debugger attached all the time, I will only use that if there is some error I want to particularly debug with the debugger and want to find with the debugger. So you can use it as a default, always on option, I prefer to not to use it to not have that extra panel and so on but to really just use the server and work on that.

<https://code.visualstudio.com/docs/nodejs/nodejs-debugging>

**Lecture 54**

**Changing variables in the debug console**

There's one more feature about the nodejs debugger which I also haven't shown you yet but which I of course don't want to forget. If you start debugging here and here I just use the shortcut and you place a breakpoint anywhere in your app and then do something to trigger that breakpoint, I showed you that you can look into all these variables and that you can look into them in the debug console without changing them. Well sometimes you might want to change them and that can be done by going to the debug view here and then by simply typing in there. Now for example, the parsed body holds this string and you can't just change it and now this will affect the runtime as you can see if you hover over this. So now if I just resume execution with my code which doesn't have a bug right now and I go back to my explorer view here, then you will see that I stored testing here which is the value I edited in the debugger. So you can actually manipulate the values there of the variables and so on if you need to, you can do that because this of course allows you to make sure that you can quickly test something. This is just another useful feature which I wanted to share with you since it can help you with debugging your code, being able to manipulate some values there.

**Lecture 55**

**Wrapup**

Useful resources:

* More on debugging Node.js: <https://nodejs.org/en/docs/guides/debugging-getting-started/>
* Debugging Node in Visual Studio Code: <https://code.visualstudio.com/docs/nodejs/nodejs-debugging>