**Lecture 117**

**Module Introduction**

It's now finally time that we start storing our data in a more correct or realistic way because thus far, we either used the memory when we just stored the data in a variable in our node program and it was then even shared across requests or we stored it in a file and that's also not optimal because accessing files is pretty slow, especially as we start storing more and more data in them and it's therefore not really something we would do in a real application. Instead there, you typically would use a database which is a specific program that allows you or that is built for storing data and for efficiently retrieving data too. Therefore in this module, we'll have a look at the different kinds of databases or two important different kinds of databases, SQL and NoSQL databases and I will compare them, highlight the differences and we will actually see examples for both in the course and we'll start with SQL also in this module, so NoSQL will also follow in a separate module but let me first of all show you how you would use a SQL based database together with your nodejs application. This is what we'll cover and that will actually be a lot, so let's dive right into it and let's learn what the different kinds of databases are and whether a SQL or NoSQL could be the right choice for your application.

**Lecture 135**

**Choosing a Database**

So SQL or NoSQL, that is the question and to answer that question, we first of all have to understand the differences or what SQL and NoSQL databases are, how they differ, how they differ regarding how we store the data and so on. Our goal always is to store data and make it easily available or accessible so that we have an easy way of accessing our data and not just easy from a code perspective but of course efficient, it should be fast. That is why we use a database, it's simply quicker than accessing a file especially as the data in there grows, it also helps us with things like we don't have to read the entire file to just find one piece of information. Now as I mentioned, we can opt for a SQL-based database, MySQL would be an example database engine that you can use or a NoSQL database and there mongodb is one of the most prominent and well known alternatives. So what is SQL, how does it work then? SQL database thinks in so-called tables, so we might have a users, a product and let's say an orders table and in each table, you have so-called fields or columns, for example a user could be defined by having an ID, an email, a name and a product could have an ID, title, price and a description. Now we fill in data for these fields, so-called records, so basically the rows in our tables. For example here we got a couple of users with their data and we get a couple of products too. SQL based databases also have one important thing, they allow you to relate different tables, for example an order could simply be described as a connection of a user and a product right because a user might order a couple of different products and a product might be ordered by a couple of different users. So basically we have such relations in SQL based databases, here we can see one example relation. This is one of the core things about SQL and in general, the core SQL database characteristics are that we have a strong data schema so that for each table, we clearly define how the data in there should look like, so which fields do we have, which type of data does each field store, is it a number, is it string, is it a text, is it a boolean? So that we have this strongly or strictly defined schema and all data in the table has to fit the schema for this table, this is really important, so this schema, this definition of how the data has to look like is one core thing in a SQL database. We also have relations between our data, that is another core characteristic of SQL based database, we relate our different tables with basically three important kinds of relations, one to one, one to many or many to many, this simply means that we can have two tables where each record fits one other record, a record might fit multiple other records or multiple records in table A can fit multiple records in table B and you'll see this in practice and in the code in this module. So tables are connected, that's another important thing. Now SQL simply stands for structured query language, so queries are a crucial things, queries are simply commands we use to interact with the database. And in SQL, a query look something like this, of course there are different commands, this would be a command that selects all users, so all entries, all records in the users table where the age is greater than 28. So this is a so-called query, we've got a couple of keywords there which are making up that SQL language, so the structured query language simply has these keywords and then we insert some parameters or some data we connect with these keywords, this is how SQL works. Now let's have a look at NoSQL in the next lecture then.

**Lecture 136**

**NoSQL Introduction**

We had a look at SQL based databases, now it's time to have a look at NoSQL-based databases. Now the name NoSQL simply means that it doesn't follow the approach SQL follows, it also uses a different query language but instead of having schemas and relations, NoSQL has other focuses or other strengths. So in NoSQL we can still have a database and we can give this database a name, shop let's say. Now that's the same for SQL by the way, there we also have databases. Now in SQL, we then had tables, users and orders and maybe also products, these are just examples here. Now in NoSQL, tables are called collections but you can think of them as tables, so as the table equivalent but we call them collections in the NoSQL world. Now in a collection, we don't find records but so-called documents which look like this and since we're working with javascript in this course here, we of course can kind of see that this looks a bit like a javascript object, so documents are very close to how we describe data in Javascript you could say. Now that are the documents in our collections and what you can already see here in the users collections example is that NoSQL doesn't have a strict schema. Here we got two documents in the same collection but the second document, Manuel here does not have an age and that is perfectly fine in NoSQL, you can store multiple documents with different structures in the same collection. Now often you of course still try to have kind of a similar structure but it's also not uncommon for some applications that you don't always have exactly the same fields available for the data you are storing in the database and that is ok in NoSQL, you can definitely store documents which are generally equal but where some fields might differ. One other thing is that in the NoSQL world, we got no real relations, instead we go for duplicate data. Now that simply means that if we have an orders collection here, we have a nested document, the user which also is stored as a separate document with more details maybe in the users collection and we don't connect that through some ID or behind the scenes setup relation, instead we simply duplicate data, to be precise the data we need in the orders collection here. That of course means that if that data changes, we have to update it in multiple places, if all these places need the latest update or the latest data change but that can be OK because this on the other hand gives us the huge advantage that if we ever retrieve data, we don't have to join multiple tables together which can lead to very long and difficult code and which can also impact performance, instead we can simply read the data from the orders collection and we probably got all the data we need to display on the orders page without having to reach out to other collections and therefore this can be done in a super fast way and that is one of the huge advantages of NoSQL, it can be very fast and efficient. So NoSQL characteristics in general are that we have no strong data schema, we can have mixed data in the same collection, no structure is required and that we have generally no data relations. Now we can relate documents in some way and this is possible and we will see how to work with connected data in the NoSQL module of this course but generally we have no or only a few connections and instead try to copy data and have a collection with documents that work on their own. This is the difference, we also got a difference between SQL and NoSQL regarding our scalability. So as our application grows and we need to store more add more data and access that data or work with it more frequently, we might need to scale our database servers and we can differentiate between horizontal and vertical scaling. We'll have a look at this and in general how the two worlds, SQL and NoSQL compare in the next lecture.

**Lecture 137**

**Comparing SQL and NoSQL**

So we learn about SQL and NoSQL, let's now compare them and let's start with horizontal and vertical scaling because as I mentioned at the end of the last lecture, we often need to scale our database to keep up with our growing application with more and more users. Horizontal and vertical scaling are the two approaches we can use to scale our database. Now what do they mean? Well in horizontal scaling, we simply add more servers. and the advantage here of course is that we can do this infinitely. We can always buy new servers, be that on a cloud provider or in our own data center and connect them to our database and split our data across all these servers, of course this means that we also need some process that runs queries on all of them and merges them together intelligently, so this is generally something which is not that easy to do but this is of course a good way of scaling. Vertical scaling simply means that we make our existing server stronger by adding more CPU or memory or with something like that, especially with cloud providers, this is typically very easy, you simply choose another option from the dropdown, you pay more and you're done, the problem here is that you have some limit, you can't fit infinitely much CPU power into a single machine. So these are the two ways we can scale, how let's compare a SQL and NoSQL regarding that and in general. Now in general in SQL we use schemas, we also have relations, these are two core characteristics and data is typically distributed across many many tables which are then connected through relations. Now regarding the scaling, it's important that horizontal scaling often is very difficult or even impossible due to the way SQL works, you can of course add more servers but running them all on one shared data cloud so to say, one shared database is pretty difficult. Vertical scaling is easily possible, you can simply make your server stronger but adding more servers can be very hard or even impossible, so definitely not trivial. So this is a problem possibly if we have multiple or thousands of read and write queries per second, then maybe our SQL database especially if we do very complex joins between related tables can reach limits or can not be the best choice. NoSQL is schemaless and has only a few relations if at all, the data is typically not distributed across multiple collections but instead we work with merged or nested documents in an existing document, though we of course also have a couple of collections for the different features of our application typically. With NoSQL, horizontal scaling is easier, still something where you have to know what you do but there are cloud providers which do that for us so we don't have to know the ins and outs of that and in general, due to the way it works with less connections and so on, this is possible. And therefore we also get great performance for mass read and write requests and NoSQL can be very performant in an application with high throughput. Now this makes SQL look very bad but the full truth is that it always depends on the kind of data you are storing, if you are storing where the relations are really important and where you want to have a split up across tables and where you want to have strong schemas, SQL can be perfect, also not every part of your data is accessed multiple times per second. You can have parts of your application where you manage general data, let's say user data which does not change that often and therefore, SQL might be very good there. Other parts of the application, let's say orders or shopping carts that do change frequently could be stored with NoSQL and there, the relations might also not be that important because you can always put all the information that belongs to a shopping cart or to an order in one single document and even if you do for example store some user data there, you might not need to touch that document just because the user change his photo because you probably didn't store that along with the order anyways. Now in this course, we will build both, we will use both because I want to teach you both and it's not so much about this course application but you should know how to use SQL with nodejs because maybe you need to add in your application or you're working on a project where you don't decide which database to use but you simply have to use it. So we will use SQL first but then I will also show you how to use NoSQL of mongodb, so we'll basically implement the course project or the current state of the course project with both databases and you will see how to interact with both of them. And let's start with SQL in this module, let's install a MySQL database and learn how to interact with it from inside our nodejs code.

**Lecture 138**

**Setting up MySQL**

Now to work with SQL, we need a SQL based database and I'll go for MySQL because you can install and use that for free, at least in the basic version which does suffice for this course. You can visit MySQL.com to download and install it and there simply click on downloads and scroll down, we don't need the enterprise edition here instead we'll go for the community edition which is the free one, click on downloads there too and now what we will need is the MySQL community server and we'll also need the workbench. Now you can also download a combined installer, especially here for Windows you should choose the MySQL on Windows Installer in tools which allows you to set up everything in one go but you can also click on MySQL community server here simply and this should give you a download which allows you to install all these things in one go anyways. So let's choose the MySQL community server here, always dive into the installation instructions in case you're facing any problems on your system and then if you scroll down, select your operating system here, the one you plan on using, so for Windows you can then also download this general installer which allows you to pick the programs you need, for MacOS we simply can go for this first download, the DMG archive and then once you click there you don't have to log in, you can simply choose no thanks just start my download to well start the download. Now this can take a while because it's quite a big file, on Windows you have the choice between downloading a web-based installer which is smaller but downloads what you need to during the installation or a complete installer which downloads everything in advance. Once your download is done, simply execute the downloaded file so the windows installer or this Mac DMG archive. Here I'm asked to well continue through the installation wizard and I will do that, agree to the license, leave the default installation path, click on customize here though to choose what to install and make sure that you do install the MySQL server here. If you're running this installer on Windows, you should have a different installer where you can choose to install the MySQL server but then also in one go, install the MySQL workbench, so make sure you select both for installation and click install here and now on both operating systems, this should install it. Now once installation is done, a configuration process should start automatically. There you should make sure that during this configuration process, you do choose the legacy password encryption which sounds insecure but which is perfectly fine and the newer version simply is not supported by the node SQL package we're using yet. So choose the legacy password encryption and choose next, you also have to choose a password for your root user and you can also leave start MySQL server once installation is complete checked or check it if it is unchecked and click finish. On Windows, simply leave all the other settings as they are and click through them and by the end, it should be done and it should then also start a MySQL server for you with the setup you chose here in the configuration steps. Now once installation is complete, on Windows if you also installed the MySQL workbench, you're done, on Mac or Linux, you're not done yet, instead you should go back to your previous page where you chose MySQL community server and there also install the MySQL workbench. The workbench basically is a client, a visual client we can use to connect to our database to inspect it and play around with it outside of our node application which simply makes debugging and developing a bit easier. So choose the MySQL workbench here and there you also can choose your operating system, now again on Windows if you used that general installer, you already have it. Download that file here, again you don't need to log in, just start to download and wait for the download to finish so that you can again execute the, well executable once it is done. Finish the installation process as instructed by the installer and once you're done with that, you should be all good. You can now test your setup by starting the MySQL workbench you just installed and then there, you should already see your MySQL instance running. If that is not the case, have a look at the attached document where I describe some common issues or in general give you some links on how to make this work and how to bring this up, you just need to make sure that your MySQL server is running and during installation, you had a choice to check that it should always start with your system, on Mac you can also open your system preferences and there you should have the MySQL option where you can also stop and start the server. To connect to the database, simply click on that instance and now enter that root password you assigned during the installation. This should now allow you to connect to your SQL server instance like this, then once you enter the password you should be connected to your database system. And right now this is what you have, basically an empty window and we won't work too much in that, we'll of course work with our database from inside our node application but this will allow us to conveniently look into our database from time to time and see what is stored and one thing we can do already is we can go down to schemas here which can be translated with database here you could say and there, we can define a new schema, so a new database with which we'll work and I'll name it node complete, you can name this whatever you want and you can also leave the other settings here and then on the bottom right and I know it's small here, unfortunately you't zoom in, you can click apply. Now this will create a new database you can say with which we can interact later, you can leave all the default settings and simply click apply here and now you should see that node complete thing which has a couple of tables or none right now but where you can then later connect to and store your data in the tables that will be created here. So with that, we can continue and we can now move on with our code and start interacting with MySQL from inside our node application.

**Lecture 139**

**Connecting our App to the SQL Database**

To use SQL or to interact with SQL from inside node, we need to install a new package and we do that with npm install of course. Npm install and let me simply bring that up to make it easier to read and npm install --save because it will be a production dependency, one which is a crucial part of our application and the name is MySQL 2, this is simply a later version of well MySQL one as you might guess and it allows us to write SQL code and execute SQL code in node and interact with a database. Now with that installed, we made one important step forward towards using MySQL. The next step is that we need to connect to our database from inside our application and for that, I will close my views here and go to the util folder we created in the past, there we have that path.js file in there which we don't really use anymore but we can also create a new file in here, the database.js file and the name is totally up to you by the way. In there, I want to set up the code that will allow us to connect to the SQL database and then also give us back a connection object so to say which allows us to run queries. For this I'll first of all import that MySQL package and store it in a MySQL constant, so require MySQL 2 is the command I need here and now there are two ways of connecting with a SQL database. One is that we set up one connection which we can then use to run queries and we should always close the connection once we're done with a query and the downside is that we need to re-execute the code to create the connection for every new query and there will be a lot of queries because we fetch data, we write data, we delete data, creating new connections all the time quickly becomes very inefficient both in our code and also regarding the connection to the database which is established and the performance this may cost. So a better way is to create a so-called connection pool and by the way, you can learn way more about this package, its options regarding how to set up connections and so on in the official docs for this tool, for this package and you find a link to those docs in the last lecture of this module. So to set up such a pool, I'll create a new constant pool, the name is up to you and I'll use that MySQL object and there I will call create pool and there you also see that create connection we could use. Now I don't want a single connection but a pool of connections which will allow us to always reach out to it whenever we have a query to run and then we get a new connection from that pool which manages multiple connections so that we can run multiple queries simultaneously because each query needs its own connection and once the query is done, the connection will be handed back into the pool and it's available again for a new query and the pool can then be finished when our application shuts down. So I will call create pool here and I need to pass in a javascript object with some information about our database engine, our database host we're connecting to. For that I'll first of all define the host, so the server IP address or name and that is localhost because it's running on our local machine here. Then I need to define the username and that by default is root that was given to us during the configuration process, I also need to define the exact database name because this gives us access to a database server but that server typically has multiple databases and here our databases are our schemas, so we'll take the node complete database here so that the value here simply is node complete. Now with that, we simply have to enter one more piece of information and that is of course our password. So here enter a password and then I used this password, you should of course use the password you assigned during installation. This will create a pool and now I can export this pool and I will export it in a special way actually, I will call promise here because this will allow us to use promises when working with these connections which of course handle asynchronous tasks, asynchronous data instead of callbacks because promises allow us to write code in a bit more structured way, we don't have many nested callbacks, instead we can use promise chains and you will see what I mean in this module of course. So now we can always import from the database.js file to get access to that pool and to the connections in there and I can show this as an example by simply going to app.js which will of course run when our application starts up and there let's simply import database, maybe store it in a constant named db by requiring util database, like this, so reaching out to that new database file we just created. This will then be the pool basically or well the pool which allows us to use a connection in it and if I now use this let's say here, I can run db and now we got a couple of methods, one of them is execute which allows us to execute queries, we also got query but execute is a bit safer so we'll use that. We can also end it and we want to end it whenever our application is to shut down but we don't shut our application down in this case here so no need to call end here, instead we just want to connect or execute a command. And we can execute a command here by writing SQL syntax as a string. Now of course that means you need to know SQL and I will teach you a very basic SQL here in this course. If you plan on using MySQL together with your node app, you definitely have to take a SQL course which dive into all the depths of the SQL language because that is far beyond the scope of this course which is of course a node course and not a SQL course but what we could do here is we could select everything from products and of course we have no products table yet, so let's quickly go back to our workbench and here on tables, right click and click create table and this gives you now the table editor. The name should now be products and in there, you can add new fields, remeber that schema thing, you add fields. So let's work on this and finish this and execute our first little code in the next lecture.

**Lecture 140**

**Basic SQL and Creating a Table**

So let's finish that setup of the products table which I need for my first little demo code. Here in the table editor so to say, in the workbench, we can define how a product's entry should look like. For that we first of all define the name of a field, ID and the data type and for the ID, an integer is fine. We can also check that it should be the primary key by which records in this table will be identified, that it must not be null, that it should be unique, that should definitely be the case, if it should hold binary data which is not the case, if it's unsigned so if it holds no negative values which should also be the case because that should be an integer starting at 1 and then incrementing, here if it is zero fill and for us important, if it's auto-incrementing and that should be the case because every new record should receive that automatically and it should be a higher number than in the last record. Now a product also typically has a title and there I'll use a var char which is basically a string, I'll just define that it may be up to 255 characters long and longer titles will simply be cut off, so that's something to keep in mind. It must also not be null, so we have to have a value in there but I don't need any other setting here. For a product, I also want to have a price and here I want to have a double so that we can enter decimal places, this must also not be null. I also want to have a description which now will not be a var char but will be text and if you're wondering which data types are available, that is exactly what I meant, you should definitely consult a full SQL course to learn more about the available data types and how to work with them. So here I got my text which is simply a longer text than the var char which has a limitation and I will have an image url which I'll also set to var char 255 which means longer urls also won't work. With that I defined how my product should look like, you can leave everything else as it is and then on the bottom right, I then click on apply here. It shows you the SQL statement it will execute and you could execute this on your own, for example in node of course to always create this new table, here we'll do it in the workbench, so click apply, close and now on tables, you see the new products table and if you click this icon here on the very right, you can see the entries in there. By the way, you also see the SQL query that was executed to look into that and that's pretty similar to the query we're executing here. So now that the table is set up, we just need to enter one dummy data so that we have something to fetch and I will simply add a book here, whoops, with a price of let's say 12.99, description this is an awesome book and also an image url. Now if you've got problems copying a value in here, you simply have to type it manually, copying also didn't work for me, of course you can also enter some dummy value and just live with no image being displayed. Now let's also enter an ID here though that should be auto-generated if you don't do it, click on apply on the bottom right, apply here, close and now if you again click on this icon here next to products and the left column, you'll see that now this one element was added here. Now that we get a book in here, let's go back to our node code and there, we can now chain then and now this is something provided by the fact that we're using promise here when exporting the pool. We now get back promises when executing queries like this with execute and promises have two functions, then and catch. Let's explore them in the next lecture.

**Lecture 141**

**Retrieving Data**

So we're executing a query here with execute on our pool, on the products table we just created and I added then and catch. These are functions we can chain onto the result of the execute call, so they will execute on whatever this gives us back and this whatever is some so-called promise. Now a promise is a basic javascript object not specific to node, it's also available in javascript in the browser which allows us to work with asynchronous code. Instead of using callbacks which we could also use with the MySQL package, promises allow us to write more structured code because instead of having a nested anonymous function here as a second argument, we simply have a then block which will then get the anonymous function to execute. Now the advantage is we can also write it like this and now we have very readable code instead of having a nested code here and that nested code especially becomes a problem if we have more and more asynchronous tasks depending on each other. Now we also don't just have then, we also have catch and this also has a function which executes in case of an error, so for example if the database connection fails or something like this. We get the error object here and this is just a modern javascript syntax where we get one argument and then we handle it here and I can simply log it with the console so that we at least see what the error was. Now we hopefully end up here and we probably get an argument here too, let's name it result for now and let's also log result here then and now let's see what we get. If I run npm start to bring up that server again, this immediately executes because it's part of the app.js file and if we have a look at this, we see this is the object we got back and in this object, we essentially also see the data that was retrieved here. The data we do get back has a format of an array with a nested array where the first nested array seems to depict our data, the rows it fetched and the second array seems to hold some metadata about the table we fetched it from. So result basically is an array with two nested elements, so we can also logout results 0 and result 1 here and if we now save this and therefore our server restarts, we have almost the same output but now it's not a nested array but here we have the first object we retrieved, the row we got and then this ends here, here is the closing square bracket and then here we get the second log, so this is the result one with the metadata. This is what we get back and this should allow us to work with our data. Now obviously we don't just want to work with dummy data but let's now see how we can adjust our models to interact with our database instead of local files.

**Lecture 142**

**Fetching Products**

Now that we successfully connected our code to the SQL database, let's start working on the project and for this, I'll remove this code in the app.js file because this was just some testing code, not the real code we'll use, instead I of course want to work on my models here, let's say on the product model. There we right now already have an exported class which we can instantiate and where we then for example have a save method to create a new product. Now even though we're still fetching product from our file for now, we also have our static methods for fetching data and there we fetch data from our files. Now we can do that but of course this is not really the set up I want to use. I will still work with static methods for fetching data but I want to fetch data from the database and not from a file. To do that in my product.js file in the models folder, I don't need fs and path because I'll not work with files and paths anymore, I don't need to construct that path here at the top. We can leave the cart for now but that functionality will be broken for the moment, I don't need my helper function here for getting products from a file because we'll not work with files anymore, I will still create a product like this so I will leave that code as it is, however I will delete my code for saving here, we'll override this eventually, I'll delete my code for deleting, for fetching and for finding by ID and I want to start with fetch all now. Here I also don't want to work with callbacks anymore but with promises, so I don't need that argument, the callback argument, I shouldn't need that in any of my functions here. So fetch all should now reach out to the database and what do we need to do to do that? Well we need to get access to the database. So let's import our pool object from the database.js file, so I'll create a new constant db and require database, whoops, not database but we'll go up one level into util and there, database.js, without js. So now we get access to the pool and now in fetch all, we can execute a query and now which query do we need to execute here? Definitely feel free to pause the video and write it on your own if you already know it. Well here we want to fetch all products so it's the exact same query we ran before, select everything, the star stands for everything, we could also select just the ID and title with this syntax but I want to select everything, all fields from products. Now as a side note, you could write select and from in lowercase too but I like to keep these keywords uppercase to indicate what is core SQL syntax and what are our dynamic values. So we select everything from products here and now as I said, this returns a promise. Now we could add then and catch here but actually I'm interested in the returned value in the place where I'm calling fetch all, so I will simply return the entire promise that execute returns so that we can use it somewhere else. So now we can go to the place where we do call fetch all and that is in the shop.js file in the controllers folder. There for example where we get the index page, we do call fetch all but right now we still pass in a function that previously was the callback. Now we got no callback anymore, so let's take out that render code we'll need that later and remove that anonymous function. Instead fetch all will now return a promise, so we can add then and catch, you don't have to add both but you typically also want to have some error handling mechanism, though we'll learn about a better one in the future, so later in this course. So here I will again simply log my error and not do anything else with it but in then, you remember we got this nested array, now we can use some next gen syntax with a feature called destructuring where I can already pull out information of the value I'm receiving as an argument here in my argument list. So here is my anonymous function which will be executed once we get data and instead of using result or anything like that which is a nested array, I can use the syntax here where I pull out my rows and my field data, you can name this however you want and this will simply be the first element of the nested array which would be our argument data and that will be the second element. And now we can use these two variables which simply hold these two nested arrays and therefore I can console log them but I don't need to log them, I instead want to render my page inside of this anonymous function, so once we got that data and rows should be my products because my rows here are the entries in the products table and therefore these should be my products. If we now save that and we go back to our running application in the server, on localhost 3000 we shouldn't see the book here and also have no errors on our console. Now we see that book because our data is retrieved from the database and therefore if we were to go to the database and we for example add an exclamation mark here in the title and then click apply, you always need to do that, apply, close, if I reload my page here, it has no exclamation mark right now but now it has, so this is really coming from the database. Now here's your little mini task, also make sure you're fetching data from the database when loading the products page which right now is broken or not working because there we still try to reach out to a file which will not work. Try to fix this on your own before we also start inserting documents or elements into the database.

**Lecture 144**

**Inserting Data into the database**

Adding products is also very simple. We got our admin.js controller where we do interact with the product in the way of creating it, here post add product, we create a new product and call save and I essentially want to leave that code as it is with one tiny change. So first of all, we have to go to the product.js file in the models folder and there, the save method is not doing anything at the moment. Now what it should do is it should reach out to the database and save the data there, so again I'll use my db constant, the one up here which gives me access to my database pool, to my connection pool and I'll call execute to execute a query. Now with SQL, we saw select for getting data, for inserting data there is the insert into command and there, we then define the table where we want to insert something and I'll use the products table here followed by brackets where we list the different fields we want to insert value into. So we have the title, the price, the image url and the description and important, you need to make sure that the fields you define here match the field names you defined in your table, in the database, you don't need to specify the ID because that should be generated automatically by the database engine. Now we're not done, this only defines where do we want to insert something, the what is missing, we now need the values keyword followed by brackets with the values. Now to safely insert values and not face the issue of SQL injection which is an attack pattern where users can insert special data into your input fields in your webpage that runs as SQL queries, we should use an approach where we just use question marks, one for each of the fields we insert data into separated with commas and then there is a second argument we pass to execute with the values that will be injected instead of these question marks, so the order of the elements we add here to this array is the order of arguments here. And we don't do this on our own because this MySQL package we're using here will then safely escape our input values to basically parse it for a hidden SQL commands and remove them, so now this is an extra security layer. And here I want to insert this title because the first question mark will be inserted as a value for title, so the first element here should be the title, second one will be the price, so here we should have this price followed by this image url and this description. This allows me to insert elements there and again I will simply return the promise that execute yields. That allows us to go back to the admin.js file, to the controller and on save, I can then add then and catch again, in catch I'll just log the error but in then, I don't care so much about the result, I just want to make sure that I only redirect inside this function, so only redirect once the insert completed. Now with that, we can save all our files and if we now try to insert a second product with some dummy url which of course will not work, 9.99 and some description and we click add product here, we are redirected and this is looking pretty good and if we have a look at our database and click that refresh button here, we see our entry, so our new entry with an auto-generated ID. So this is working, we now are able to also insert data into our database. Now as a next step, let's make sure we can click the details icon here and therefore retrieve data for a single document or a single entry in our database.

**Lecture 145**

**Fetching a single product with where condition**

We're able to to save a new product and fetch all products, let's now find a single product by ID. This also is not that difficult, we can again use our db pool, execute a query and the query we want to execute here is select everything from products and everything here means not all rows but simply all fields but now we can restrict the number of rows with a where condition and where is another SQL keyword. So there we can execute where products ID equals one equals sign only, not multiple ones as in javascript, where the ID is equal to question mark simply to let my MySql inject the value again, the ID we're getting as an argument here. Now let's return this promise here and this is our statement for fetching a single product with all the columns though, so with all the data. Now we can go back to shop.js and there where we do fetch a single product in get product, well there we still use find by ID for a given product ID but of course no nested function here, that didn't change so let's strip that out of there, simply call find by id like this and instead add then and catch and in catch, we got our error which we can output and in then, we got our product in the end or to be precise we of course got that nested array where we know that the first element will be all the rows we got and that will just be our product or it should just be the product. Make sure to wrap that special syntax with the square brackets in the parentheses. So now we just need the render function here and place it in this function we passed to then and we're extracting a product here, we're passing it to our view here, let's see if that works. If I now save this and I click on details, this failed, now let's simply console log our products here to see why it failed. If I reload this page, here's what gets logged and the reason why it failed is that product still is an array, an array with one element only but still an array but the view simply expects one single object, not an array with one object. The solution is to simply pass the first element in that array and we know that there will only be one element in that array. So now if I reload this page, we see the second product, I just didn't add an image for that otherwise we would see that too. So this is working, we're able to fetch a single product too.