**Lecture 149**

**What is Sequelize?**

Now before we start using it, what is sequelize actually? Sequelize is a third party package, to be precise it's an object relational mapping library and this is a pretty long name which simply means it does all the heavy lifting, all the SQL code behind the scenes for us and maps it into javascript objects with convenience methods which we can call to execute that behind the scenes SQL code so that we never have to write SQL code on our own. It works like this, we got our object let's say a user with a name, age, email and password but of course this can be anything, could be a product, whatever you need and this is mapped to a database table by sequelize, so it automatically creates that table for us even, it automatically sets up relations and tables even for us, it does all that and when we create a new user for example, we simply call a method on that user javascript object and sequelize executes the SQL query or the SQL command that is required. So instead of writing this on our own, we simply create a javascript object and work with that and here would be one example using sequelize to create a new user which would behind the scenes execute the SQL code we don't have to write. Sequelize offers us the models to work with our database as I showed you on the last slide and it allows us to define such models, so basically define which data makes up a model and therefore which data will be saved in the database. We can then instantiate these models, so these classes so to say, we can execute the constructor functions or use utility methods to create let's say a new user object based on that model so we have a connection here and we can then run queries on that. That could be that we save a new user but it could also be that we find all users as an example and here again, this always relates back to our model which we define with sequelize. And we can also associate our models, for example we could associate our user model to a product model. So this is what sequelize does but of course we dont just want to learn that in theory, we want to see that in practice. So let's add sequelize to our project and let's slowly integrate it to manage our products in our cart and so on through sequelize.

**Lecture 150**

**Connecting to the database**

So back in the project, we first of all have to install sequelize and just as we installed all the other packages, we'll do this by running npm install --save because this also is a production dependency, it's a core dependency of our project and then the name is sequelize, like this. Now important, sequelize needs that MySQL 2 package which we already installed, so this MySQL 2 package we installed in the last module needs to be installed. If you skipped the last module or anything like that, make sure to also install MySQL 2. Now with that, we got sequelize installed and now we can do a couple of cool things with it. The first step always is that we create a model with sequelize, well and also that we connect to the database of course. Now therefore the first step is that when I connect to MySQL database with the workbench which we also used in the last module already and in there, I will go into my node complete database and delete the products table by right clicking on it, drop table and then simply click drop now, I do this because I now want to use sequelize to manage my tables. With that let's go back to our project and now let's go into the database.js file in the util folder, there I now want to write some code to connect sequelize to the database. Now sequelize uses MySQL 2 behind the scenes, therefore sequelize behind the scenes will do something like this but we won't write this. Instead we will import sequelize and I'll store it in a sequelize constant, you can name this constant however you want but I'll name it with a capital S cecause I actually import a constructor function or a class here so I'll import sequelize like this and then I'll create a new sequelize instance by calling new sequelize, like this. Now sequelize, here the constructor function needs some options. You can see we have to configure it with the database name with a username to connect to it, a password. So here I will connect it to my database, so to my schema name which is node-complete, node-complete with my root username which is root and with my password which is nodecomplete in my case, use the root password you assigned. Now we can also pass a fourth argument, an options object and in there you can see, I saw this menu or get this menu with control space, you can see we can set up a bunch of stuff, for example the dialect, we can set this to MySQL to make it clear that we connect to a MySQL database because different SQL engines or databases use slightly different SQL syntax and you can dive into the official sequelize docs to learn all about it, a link can be found in the last lecture of this module. The one thing I want to set for now is the host, by default it would use localhost so we don't need to set it but I will explicitly set this to localhost. With that, we're creating a new sequelize object and it will automatically connect to the database then or to be precise, it will set up a connection pool just as we did it manually in the last course module. So I can now export my sequelize object here which is essentially that database connection pool however managed by sequelize giving us a lot of useful features. With that, we got the connection set up, let's now focus on working on the models.

**Lecture 151**

**Defining a Model**

With the connection set up, let's go into our models folder and there to the product.js file and there we can actually delete everything. We don't need all of that right now, we'll write it from scratch. Here I first of all need to import two things. The first thing is sequelize itself, so again I will require, whoops, require sequelize here and that will give me back a class or constructor function hence I name this here with a capital S. The next thing I'll import is my database connection pool managed by sequelize, I'll also name this sequelize but with lowercase s, so I'll basically import what I export in my database.js file. Here I will import this by requiring and then go up one level into util and then there to the database.js file. Now with these two things imported, we can now define a model that will be managed by sequelize. So here in this file below my imports, I'll create a new constant and I'll name it product because in the product.js file I want to define my product model. Now this model is not defined as a class as we did it before but instead I use sequelize with the lowercase s, so my database connection pool which is more than a connection pool, it's actually a fully configured sequelize environment which does also have the connection pool but also all the features of the sequelize package and there we can define a new model by calling define. The first name that is the model name and by the way, here in the suggestion you also see an example and so on as you do in the official docs of course. Now the model name typically is a lowercase name and I'll name it product like this, you are free to name this however you want but to follow along smoothly, I recommend taking this name. The second argument defines the structure of our model and therefore also of the automatically created database table. This will be a javascript object and in there, we simply define the attributes or fields our product should have, for example I want to have an ID. Now an ID is then in turn defined with an object where I configured this attribute, there for example we set the type and the type is now one of the types defined by the sequelize package. So now I use sequelize with a capital S, so from this import here and there you find a bunch of types like number or string, so not var, char and so on but here you have the more javascript-ish names I could, would say and I want to use an integer here because my ID will be a number starting at 1 and then incrementing. To make it increment, I also configured this attribute to be auto-incrementing by setting auto-increment to true and in case you're wondering how I do know all of that, well again you find that in the official docs. On docs.sequelizejs.com, you can dive into sequelize, learn more about how it works and how you can configure it and under model definition, you learn all about how to define a model, how you can define your different tags for the model and you also see a list of all the supported data types, like the integer we just used. So this is where you can learn more about this and I can tell you that I want this to be auto-incrementing so I will set auto-increment to null, to true. I also don't want to allow this value to be empty, so I will allow null to false because I don't want to allow null values in there. Last but not least, I'll set primary key to true to basically define this as the primary key of the table which is an important concept in SQL databases for retrieving the data and then also for later defining relations. With that defined, a product of course also has other fields, for example a title. Now we can again define a javascript object to configure it in detail or if you just want to set the type, you can also use sequelize and then the type, so this is a shortcut in case you only want to set the type, though you could argue that we also at least want to set that this should never be null but here as a demo, I will just assign the type. For the price I'll go back to the javascript object and set the type to sequelize and then double and I will again set allow null to false to not allow null values here. Then I will set my image url here again to a javascript object where I set the type to sequelize string and I will set allow null to false to not allow null values here. And last but not least, I also want to have a description which also will have a type that is of type sequelize string and I will set allow null to false. Now this is my product model, last but not least we need to export it. So below my model definition here,I'll simply use module exports and export my product here, so this constant in which I define, store the defined model, this gets exported here. And with that, we made a huge step forward and we can now start using this product. Now let's see how we use it over the next lectures.

**Lecture 152**

**Syncing JS Definitions to the Database**

We define the product model and I mentioned that we can now work with this model to create new products and so on but for this, we of course also need a product table in our database and there right now we got no tables because I deleted the products table earlier in this module. I also mentioned that sequelize can create these tables for you and it indeed can. Now to create tables for you, you just have to tell sequelize to do that and I'll do that in the app.js file. In there I want to ensure that all my models are basically transferred into tables or get a table that belongs to them whenever we start our application and if the table already exists, it will of course not override it by default though we can tell it to do so. Now in my app.js file which is the file execute when I do start my program, I want to import from the database file as I did it before, I'll just rename it to sequelize because I do import my sequelize object with a lower case s because I'm importing from my own utility database set up file and then towards the end of this file, let's say here, I want to call sequelize and then there's a special method, the sync method. The sync method has a look at all the models you defined and keep in mind you defined your models in your model files by calling sequelize defined on that same sequelize object, so it is aware of all your models and it then basically creates tables for them. That is what sync does, it syncs your models to the database by creating the appropriate tables and if you have them, relations. So here I will call sync and then I can listen to the result of this, let's see what we get back as a response here or what we get back as a value here and we can also of course catch potential errors that occurred and if an error occurred here, well then we can essentially also log that and I only want to let's say start my server if we somehow made it into then, let's see what this gives us. If I now run npm start here, it starts up and we see there is some log output. If you scroll up quite a bit because we got back a complex object, you see that this is a default log thrown by sequelize, it executed this SQL query for us without us writing this query. It created a table if it did not exist yet which it named product, products and that is that automatically inferred name because we named our model product, it automatically pluralizes that and then it assigned a couple of fields there which it configured according to our model definition. And then this is the return value we get back, basically our sequelize object you can tell and if I now quit this server with control c, clear the console and rerun npm start, you'll see it runs this again but it does not overwrite the existing table because we have that if not exists check in there automatically. So we can run this again without issues and our server starts up even if this table already exists, we still make it into then. Now I will comment out that result log because I don't want to have this long object every time, in the console every time we start this. So now I just get MySQL query here and if we now have a look at the workbench and we right click on our database and click on refresh all, we see that under tables, we get a products table and if we inspect that with this icon, we see all the fields we defined and that is added by sequelize to new fields, created at and updated at. So it automatically manages some timestamps for us, we could disable this but I actually like this feature, so we get these automatically managed fields too. This is how we sync our tables to the database and what sequelize does for us, and with that, we're now ready to use that.

**Lecture 153**

**Inserting Data & Creating a Product**

So time to use sequelize outside of the table creation. For this, let's go into our controller files and there let's start with creating a new product because that will be helpful for later also retrieving them. In admin.js, we right now import our product model from the models file and that is fine because we do still export it there at the bottom, so we should still import it in admin.js but here in post add product, we will now work with that product a bit differently. What it will do here is I will get rid of that old code we used for creating a product and instead, I will now create a new product here by calling one of the methods provided by sequelize and by just typing the dot, my IDE suggests me a lot of methods I can call and these methods are all coming from sequelize which shows its great power and there we got create for example. Create creates a new element based on that model and immediately saves it to the database. There also is build which also creates a new object based on the model but only in javascript and then we need to save it manually, so create basically does it in one go, with build we get the object in javascript first before we then have to save it manually. I would go for create to immediately store it and now create here simply takes some arguments that we need to pass per our model definition. So I can pass in a javascript object and now here, I even get the auto-completion by my IDE to assign field values. Now I don't need to assign an ID, that will be managed automatically but for example I need to assign a title by storing the title here, so on the right side of the colon, I'm referring to my constant, left side refers to one of the attributes I defined in the model. I also have a price which I assign to price, I also have an image url which I assign to image url, whoops, image url and I also get a description which I assign the description value I fetched and as I said, this will immediately save it to database. Now just as MySQL did in the last course module, sequelize works with promises. So here we can chain then and we can chain catch, we can catch any error we might get to see what's going wrong and we can now also have a look at the result we get in case it does not go wrong. Now with that if we save this, let's go to our application and it won't work by default because fetch all is not working because we have no function place to fetch products. So for now, we have to immediately go to /admin/add product to load this page, the other pages won't work and here I'll store my first book. I fetch an image url, make sure it's not a base64 url but a real url pointing at some jpg or png file. I'll assign a price here and I will store a description, this is the first book I add through sequelize. If I click add product here and we go back to nodejs, this seems to have succeeded here because we get no errors, here we see the SQL statement it executed, insert into. I got an error before that though, that was when I tried to reach the well the page listing all the products where we failed to call fetch all because that is not provided by sequelize but thereafter once I was in the right page, inserting this was executed. You see it also executed the syntax where it escapes the values automatically as we did in the last module and in the return value, this is looking good, it looks like it did create this book and we can prove that by going to our products table and refresh this and we should see our book being added there, so this indeed succeeded. So this is how we can add a value through sequelize and this is a huge step forward of course and I'll comment out my result log here and simply log created product to not pollute my console as much. So now we created a product, let's in the next step have a look at how we can use sequelize to retrieve data.

**Lecture 155**

**Retrievning Data & Finding Products**

With the product created, time to also retrieve a product when we for example visit our index route. So in shop controller here in get index, it would be nice if we could get our products. This current approach with fetch all will not work because product is now a sequelize model as we're importing it from our model file and the sequelize model simply have no fetch all method but sequelize models have plenty of methods for getting data and instead of fetch all, they for example have find all to get all the records for this model. Now find all as you can imagine also gives us back a promise where we can use the result. Now find all by the way also can be configured with some options. We can pass our options here and we could define a where condition to also restrict the kind of data we retrieve and you can read more about the possible options you have there and how to write this in the official docs but we'll also see a way of limiting the data we retrieve when we later fetch a single product. For now let's get all without any restrictions and then here in the then block, we should have our products. So here, let's assume we get a products list array to our function that get executed here, that gets executed here. Here I will log any potential errors we might have and now in the then block, I essentially want to render my page of course once we got the products and simply pass the products into the prods key of my render function here. Let's remove the fetch all call down there and with that, time to save this, let's go back to just localhost and indeed this is looking good. You see it retrieves the data, the data still has the same field names as before and therefore rendering this automatically works. Now we need the same logic on the products page and therefore I will just copy that and go to get products added here and of course replace the render function here, make sure to pass products to prods though. And of course as I mentioned earlier in this course, you could refactor that for the index and the products page to reuse this code instead of copying it, I just like the more explicit approach here which makes it really clear what's happening. So now we got get products working too hopefully, yeah this is looking good, both work and we see even a difference because here I have a blank between the dollar sign and the text and here I don't have it. So this is working and this is a huge step forward, now as a next step let me show you how to retrieve a single product if we click on details here.

**Lecture 156**

**Getting a Single Product with the ‘where’ condition**

Time to retrieve a single book when we click on details, so essentially I'm talking about the get product action here in my controller. We get the product ID here and indeed this is the old find by ID method I defined but if we were to delete it, we see that sequelize also has a find by ID method. So actually if I leave this code exactly as it is right here at least, it will work, one difference is that with sequelize, we don't get an array of products here instead I get a single product so I can just use it like this. So with these tiny changes, turning this into a single product and using it as such and leaving this as it is actually, I should already be able to view the details and I am, just the images may be a bit oversized here, that is something we can quickly fix in css. I quickly added the image class to the div wrapping my product in the product detail view here and now in the main.css file where I also have my centered class, I will add a new image class and restrict the max height to let's say 20rem and then also say that any image text in the, nested in an element with the image class should respect that height and that should also be just height not max height. And with that if I now reload this page here, this is looking better and we should have no problems with the image anymore. Just a tiny change, the main takeaway of course is we can retrieve the product like this. However I also want to show you an alternative way, so using this syntax here is perfectly fine but let me also show you how we can use the normal find method we also got, find all to be precise. We only have one product with that ID of course but I want to show you that where syntax, so any object we can pass to find all, I'll set the where key and there you got a rich query language or a rich amount of options you can use to configure this. More can of course be found in the official tutorial, if you go to querying there, there you'll find all the details about how to configure your queries, for example you can also control which attributes are retrieved, you don't have to get all attributes all the time, if you only need the title, you can define this too and you'll also see how to use where. How to use it in a basic form, where you want to simply check that one attribute has one exact value but also how you can use operators to have alternative conditions or to check if something is greater than or greater than equal or lower than a value and so on. So you got a lot of options there and definitely check out these docs, here we could say I'm looking for all products where the ID is equal to prod ID and then I will use them or catch any errors. Now the one important thing here of course is by default this gives us an array because even though we know that only one product will have this ID, find all per definition always gives you multiple items even if it's an array with only one element. So if we use this syntax and we render this, we'll have products and we're interested in the first product, in this case also the only product as we know. So now I can comment out the other approach and if we save that and we go back to our application and we reload that detail page, it looks like it works but actually this keeps on loading so something went wrong and indeed we got an error here, product is not defined. Yeah here for the title I should also use products zero of course. So now with that if we save that and let it restart therefore, now we can reload this page and it works as before but now is using find where, find all with that where query and I simply wanted to show you this alternative approach. Of course it's perfectly fine and even preferrable in this case to use find by ID, so I will actually switch back to that other approach. It's good to know how you can query though.

**Lecture 157**

**Fetching Admin Products**

Now let's work on the admin products page because there, I also want to display my products so that we can interact with them. So in admin.js again, get product still uses fetch all, now it's of course a good practice for you to make this work with sequelize. So definitely pause the video at this point and try to change this code here such that it does retrieve the products and render them on the admin page. Were you successful? Well the code to you is of course the same code we used in the shop.js, instead of fetch all, we'll use find all and we don't use the callback function approach here because we're not using a callback function with sequelize, instead we have then and catch since we're using promises. So let's console log our error in this case or pass our function we used as a callback before to then and now here, we have our render function where we render the products. Therefore if I now reload this admin page, I see my product here and I can now add it or delete it. These however are two things that don't work, added doesn't work because I don't load the product successfully and delete wouldn't work either. So in the next lectures, we'll work on these two things.

**Lecture 158**

**Updating Products**

Let's make edit work and for this, we first of all need to load the product that gets edited so we're looking at the get edit product function in the admin.js controller. We still want to retrieve that edit mode thing and the product ID as we did it before. We can also still find a product by ID, however as before, we'll not use a callback function here but our promise where we catch any errors we might have and then here we pass that function which gets the product to then. If we get no product, we redirect which is fine and otherwise we'll render the view with our loaded product. With this tiny change already if we save this and we reload the edit product page, this is looking good, the fields get populated with our values. Now we just need to make sure that if we do change something and we save it, this does get saved to the database correctly. For that, we of course have to have a look at post edit product which gets called once we submit this page. There, we retrieve all the data we need which is good but we won't save our product like this instead here, we'll use the product model and first of all here, I want to find the element in the database which we do want to change, so I'll use find by ID here and I'll pass my prod ID to find it. Then you know the game, we got then and catch, so let's just handle any error by logging them for now and in then, let's work with the product we retrieved, that product now needs to be updated. So here we can simply do that by now saying product title equals updated title, so we can simply work with all the attributes our product has per our model definition and change them, please note this will not directly change the data in the database though, it will only do it locally in our app, in our javascript app here for the moment. So I can also change product title to updated price, to updated price, I can change the description to updated desc and I can also change product image url to updated image url. Now as I said this will not directly edit it in the database, to do that we simply have to call product save. This is another method provided by sequelize and this now takes the product as we edit it and saves it back to the database. If the product does not exist yet, it will create a new one but if it does as this one, then it will overwrite or update the old one with our new values. Now here we can again chain then and catch but to not start nesting our promises which would yield the same ugly picture as nesting callbacks, we can now return this here, so we return the promise which is returned by save and we can simply add a then block here and this catch block, whoops, this catch block would catch errors both for this first promise here and for the second promise. This then block will now handle any success responses from this save promise here, so here we get back result and I will simply log updated product. Time to save that and to go back and edit this by adding a couple of exclamation marks. Let's click update product now, we get redirected to the products page, here we don't see our change though but if we reload, we do and in the database if we refreshed that here, we are of course also can see that change here. Now do you know why we didn't immediately see that change on our admin products page though? Well the reason is we redirect here and as you learned, javascript and nodejs simply executes your code from top to bottom but async operations like this simply get registered and started and then here for promises, we registered that this function should be executed once the promise is resolved. So it will not wait for this to finish but instead move onto the next line and the next line is this one here, so it will redirect before our promise is done. So we should simply move redirect into the then block here. By the way this also means that for now if we have an error, we never load a new page which is not the best user experience but we'll learn more about error handling in a separate module of the course. So for now, this is the setup I want to use and with that if I now edit this and I remove the exclamation marks and I do let's say also change the price here, if I now save that, now we immediately see the new values because now we only redirect after the update was successful.

**Lecture 159**

**Deleting Products**

Updating was successful, now let's also make sure we can delete products. So in admin.js, I'm talking about the post delete product method here now. We get the product ID which is good, now delete by ID does not exist in a sequelize world, instead on the product we can call destroy and destroy allows us to destroy any product we find through our options here and these options allow us to for example add a where condition to narrow down which product to delete. We can also use a different approach, instead of calling destroy like this and adding a condition which product to find which is of course fine, we can also use find by ID again to find a product by that ID and then again have our well known then and error, catch methods and in then, we know we got a product and on that product, we can now also call destroy and now this will destroy the found product. We can return this because this will also yield a promise and therefore add another then block which will execute once the destruction succeeded and there we can if we want console log destroyed product and we can now also redirect thereafter. We should redirect here to make sure we only redirect once the deletion succeeded. If we now go back to our application and I add a new dummy product without any picture, just like this and we go to admin products, I got stuck on loading this automatically, I'll look into this a second but if we go to admin products manually, we see it here if I click delete, this works. Now one thing I noticed is that if I added a product here, we get stuck and we don't redirect, so for adding a product host add product here, we don't do anything in then block, there we should of course also redirect to /admin/product, so basically to the same we redirect when deleting. So now if we go back and we add a new product again, this is working and now we can also delete the products here. So this is now all working and now we can manage the products, we can view them here, now one thing we haven't touched at all are relations though because we don't just have products, we all have a cart and eventually, we'll also have users.

**Lecture 160**

**Creating a User Model**

So it's now time to introduce more models and I will introduce a brand new model before I start working on the cart again, that new model will be my user model but for now we have no real authentication process, so we'll only work with one dummy user who doesn't really have to log in, authentication will follow later in the course but I still want to show you how you could have a user who did create a product and who therefore is connected to that product and at the same time, a user should own a cart and that cart will hold multiple products and this is how we can then overall connect everything. So I'll add a user.js file in the models folder and in there first of all, let's define a user model. And this is also something you can try on your own, try to define a sequelize model with a user that has an ID, the ID should have the same function as it has for the product model and let's say for now a user also has a name and an email. Definitely try this on your own, pause the video at this point, we'll thereafter define the user model together. Were you successful? Let's define the user model by first of all requiring the sequelize constructor or class and then also with lowercase s, let's import our own sequelize object which holds the connections on from the util folder and there, the database file. We can then define a user and store the user in a user constant by calling sequelize define, I'll name the model user like this and then as an object you define the structure of the user. I want to have an ID for my user and the type here will be sequelize integer, it should auto-incrementing, I don't allow null values and this will be my primary key. Besides that ID, I'll also have a name and here I'll use a very short definition where I say that the name will be a string and the email here will also be a string. So this is my user, now we need to export this model with module exports user like that and now with that being exported, we can start using it. And one thing I want to start using it for is that I want to create an association. Now let's have a look at how we do that and what an association actually is in the next lecture.

**Lecture 161**

**Adding a One To Many Relationship**

What do I mean by association? You could also say relation. let's say in our project and that is basically our project, we have products, users, carts and at some point also orders. Now if we want to connect all these things on and define how they work together, then a product would probably belong to many carts because our users will have carts therefore we have multiple users, multiple carts and therefore a product can belong to many carts because of course different users can add the same product to their carts. Each user only has one cart though, so this is how we could relate that, a product also can be part of multiple orders and a user can have multiple orders because you typically order more than one thing. A user can also own multiple products in a sense of this user created this product, so not own it in the sense of I bought it but in the sense of hey I offer this product, I created it in the shop. This is a rough outline of how we can communicate or relate different models and this is what we can also reflect in sequelize. There I'll go to my app.js file here, let's close the views, I'll go to my app.js file and before I sync all my data to the database, I want to define my models. So for this I will add more imports and I will import my product model by requiring this from the models file and there the product file, add a models folder in the product file and I'll import my user model from the user file in the models folder. With the two models imported, we can relate them and I will relate them here in the same place where I sync sequelize but before I sync it. Here I can basically say that a product belongs to a user. Now you can learn way more about these relations in the sequelize documentation. There, there is a whole article about associations, which kind of associations exist and how you define them in sequelize and which effect this has. I will show you some important implications in this model of course. So back to our code, we are now setting up that for sequelize, a product belongs to a user and this is now talking about a user created this product, we're not talking about purchases at this point. We can also configure this by passing a second argument which is optional, here we can define how this relationship be managed and very important, we can define so-called constraints, set them to true and for example say that on delete, so if a user is deleted, what should happen to any connected products? And here we can say cascade which simply means well the deletion would then also be executed for the product, so if we delete a user, any price related to the user would also be gone. This is totally optional and you definitely need to learn a bit more about SQL to fully understand your options here, this is beyond the scope of this course but this can all be done with sequelize and now we got this relation set up. You can also define the inverse and say that one user has many products because one user can of course add more than one product to the shop. Now this is optional, you don't need that, you can basically replace belongs to with a has many call but here I also like to define both directions to really make it clear how this relation works. Now with this being set up, sequelize sync will not just create tables for our models but also define the relations in our database as we define them here. The one problem we have right now is that we already created the products table and therefore will not override it with the new information and we can ensure that it will by setting force to true. Of course a setting you wouldn't really use in production because you don't always want to overwrite your tables all the time but here during development, I want to reflect my new changes so I'll set this to true and therefore after restarting, we indeed see a couple of statements were executed. First of all it dropped any existing tables and then it created a new table, users with all the set up and then it also creates a new products table and besides adding all the fields there, it also defined that there is a new field, the user id field which is an integer and which is a foreign key that references the ID field in the users table and that on delete, it should cascade and on update cascade is the default. So this is some meta setup in the database which sequelize now also added to connect our tables there too. And if we go to workbench and we right click on our database and set call refresh all, we see there are two tables now and indeed if we inspect products, we see that our product is gone because it recreated the table but now besides created at and updated at that were added by sequelize, there is a user ID field which was also added by sequelize and this will automatically be populated by sequelize too once we create products that are related to a user. So let's make sure that we have a user because right now that table is empty and that we then can connect users and products in our app.

**Lecture 162**

**Creating and Managing a dummy user**

So we need a user and as I mentioned, for now we have no authentication process, so for the moment I will create a user manually. First of all, I will remove this force through call because I don't always want to overwrite my tables otherwise all our products and so on will always be gone, so I'll go back to sync now that the relations are set up and once all the tables were created, so in this then call here, I actually now also want to create my user and therefore here, I will use my user model and first of all check if I find a user with the ID one and this is of course just some dummy code to see if I do have one user because I only need one for now as we have no authentication and if I do have it, I'll not create a new one, if I don't have it I will. So here I have user find by id one and I will return this and therefore also take app listen out of this then block and then add a new then block where I will get my retrieved user. Here I'll then check if I don't have a user, so if this is null because if I don't have a user, I want to create a new one by calling user create and there I'll pass in a javascript object where I set the name to Max and the email to some dummy e-mail, you can enter any values you want here, just make sure you populate all the fields you defined in your user model. This also returns a promise so I'll return this too but also we might never execute this because we already got a user. So for this case I'll return user but now we're inconsistent because now this anonymous function either returns a promise or just an object, we should always return the same so that we can chain then here successfully and therefore I will actually call promise resolve here which is essentially a promise that will immediately resolve to user. Technically you can omit this though because if you return a value in a then block, it is automatically wrapped into a new promise, just wanted to highlight that you should make sure that the values are equal but here it's managed for us and therefore here I now definitely know that I got a user, we can console log the user here and I also start listening to my server here. So here's my user creation result that I get back and if I go back to my workbench database and I refresh the users table, we see the user here too. And if I do restart my server with npm start, that still works and it doesn't create a new user here because we already have one. So this code is working as it should, I'll comment out this and with this change made, we now always have a user available. As a next step, I'll will register a new middleware because I want to store that user in my request so that I can use it from anywhere in my app conveniently. So maybe here I'll add a new middleware, I'll simply add a function with request response in next as you learned it before in this course and in here, I want to reach out to my database and retrieve my user with user find by id one. Now you might be wondering if this can ever return a user and if we only create it down there. Now keep in mind, app use here only registers a middleware so for an incoming request, we will then execute this function. Npm start runs this code for the first time and npm start is what runs sequelize here not incoming requests, incoming requests are only funneled through our middleware. So npm start runs this, this code which sets up our database but never this anonymous function, it just registers it as middleware for incoming requests. So this code will only run for incoming requests which on the other hand can only reach this if we did successfully start our server here with app listen and that in turn is only true if we are done with our initialization code here, so we are guaranteed to find a user here. So we can find a user by id here and what do we want to do with my user in the then block then? Well I want to store it in a request. So here I will set request user equal to user and we can do that, we can simply add a new field to our request object, we should just make sure we don't overwrite an existing one, like body. But user is undefined by default, now I'm storing the user I retrieved from the database in there. Also keep in mind the user we're retrieving from the database here is not just a javascript object with the values stored in a database, it's a sequelize object with the value stored in the database and with all these utility methods sequelize added, like destroy. So we're storing this sequelize object here in the request and not just a javascript object with the field values, it is that we got the extended version here and therefore whenever we call request user in the future in our app, we can also execute methods like destroy. So with that, all I need to do is I need to call next here of course so that we can continue with the next step if we get our user and stored it. Now with that, we've got the user set up and retrieved, let's now make sure that we can also use it to create new products that are associated to that user.

**Lecture 163**

**Using Magic Association Methods**

From now on all new products that are created should be associated to the currently logged in user and for now, this will only be this one dummy user. That means that if I'm in the admin.js controller, here when we create a new product in post add product, we'll not create the product like this anymore, we need to pass in extra information regarding our user that is associated. One way of doing this is that we set this new user ID field we got, keep in mind user ID was added as a database field because we now have a relation set up and we set this to request user ID, keep in mind that request user is the sequelize user object which holds both the database data there for that user as well as the helper methods. So this should create new products with that user being associated to it, let's test this. Let's go back to add product and simply enter some dummy values here for now, seems to work, we get no error here and if we look into our products table by clicking on this icon here, we indeed have the user ID stored here. Now we also have one tiny problem or thing we can improve at least. We manually fetch the user id and this is not a lot of work here but there is a more elegant way of setting this so that we don't manually have to set the user ID. We can use another cool feature of sequelize, we can use our user object as it's stored in the request and always keep in mind, this is a sequelize object with all the magic features and there we'll actually have a create product method. Now where is that coming from? Well it is something you can get from the official docs if you read through associations. There you'll learn that if you set up associations, sequelize add special methods depending on the association you added and for a belongs to has many association as we did, sequelize adds methods that allow us for example to create a new associated object. So since a user has many products or a product belongs to a user as we learned or as we set it up in app.js, since we have that relation defined, sequelize automatically adds a create product method to the user. Create product because our model is named product and create is then automatically added at the beginning of the method name, that is some magic done by sequelize. So create product is available and there we simply pass in the object with the product data that can't be inferred by sequelize so basically anything but the user ID and the timestamps and then we can chain our then and catch block here, just as we did it before. The rest doesn't change but this now automatically creates a connected model. So if I now save this and I add a new product, this still works and if we have a look at our database and we refresh the products table here, we see we also get the user ID here even though we did not set it explicitly. Now this is done by sequelize with this magic way of connecting it and that is a really cool way of using associations in sequelize and making sure that our models know about each other.

**Lecture 164**

**Fetching related products**

We learn about associations and that we get these cool magic methods here about which you can learn more in the official docs of course, now which implications does this have for our other admin.js actions? Well for get added product, there's no implication, we fetch a single product here and we can do this as before, we don't really care about the user here, you could argue that you only want to find products for the currently logged in user though. Then what you actually want to do is you want to use request user get products and there define where ID equals prods ID to also have that filter and then you can chain your normal then call and so on. And with that if you click on edit, we see an empty form because it generally did work and we see the SQL statement here where it simply looks for a product with the user id one and that is not the condition we wrote, we're responsible for this part where it then also narrows down the product ID but user ID one was added by sequelize because we use get products on the user but keep in mind here we get back an array even if it only holds one element, so we got products and therefore we know that one product, the one we are interested in will always be the first element, so we have to store that separately in a new constant and now if we reload this, this works. So this is a more elegant way, though we could also still only use the ID if we want to use the old approach. If we move on for post editing a product, there I'm fine with finding the product like this because if we are at this point, I assume we already have a product for this user only, so if I update it like this, it's fine. Now get product should change, instead of finding all products, I want to find products for this user, so I'll call get products like that which will give me all products for this user and then I can render them here. So if I now go to admin products, I see the products for this user here because this select statement gets executed where we narrow down the user to the user with the ID one and post delete product, here again we could only find a product for this user with this ID, I'm fine with this setup though. So some tiny changes showing you more of that power sequelize gives you for associated models. Let's next have a look at how we can reintroduce the cart to our application.

**Lecture 165**

**One To Many and Many to Many Relations**

We don't just have products and users, we also have the cart and we haven't worked on the cart for quite some time. Now from a relation or association perspective, a cart should belong to a user and a cart that in turn simply holds products, many products with a quantity associated to them. So actually we need more than one model, I can already tell you that but let's work on the cart model first. I delete everything here and start writing it from scratch. I'll first of all import sequelize by requiring the sequelize package and of course then, I will also import sequelize from my database file, so from util database like this and then it's time to create the cart model with sequelize define as we did it before, name it cart and a cart for me here simply has an ID and that id will be of type sequelize string, auto-increment should be true, allow null should be false and the primary key will be true and then I will export this. Now you might be wondering where are the products? Well we have to keep in mind that a cart should belong to a single user but may hold multiple products. The carts table however should hold the different carts for the different users, so we'll not just need the carts table and model, we'll also need a new cart item.js file and in there, I will also again, I'll copy the code from carts, define a new model, I'll call it cart item though and export cart item here and I'll name it cart item here when I define the model name and this will also have an ID but then also a quantity and that quantity will be an integer because each cart item is essentially a combination of a product and the ID of the cart in which this product lies and the quantity. Now the ID of the cart to which this is related doesn't have to be added by us here because we will again create an association and let sequelize manage this, so it's time for some associations then. Let's go to the app.js file here and in there, time to add more associations besides product and user. For example a user has one cart and for this to work, we need to import the cart, so let's scroll up to the imports and let's also import cart here from the cart file. We also need the cart item so I'll import cart item from the cart item file in the models folder, so these two new imports have to be added. And with them added if we go down again, let's go back defining the relations, a user has one cart and a cart on the other hand belongs to a user, so this is basically the inverse of this relation and it's optional, you don't need to add it, a one direction is enough. This setup here, either of the two approaches will add a key to the cart, a new field to the cart which is the user id to which the cart belongs. Next I will add my cart here and say that this cart belongs to many products and on the other hand, a product belongs to many carts, it's a many-to-many relationship because one cart can hold multiple products and a single product can be part of multiple different carts. This only works with an intermediate table that connects them which basically stores a combination of product IDs and cart IDs and for that, I created my cart item model. I add a second argument to belongs to many and there we add the through keep telling sequelize where these connection should be stored and that is my cart item model, so I'll add that to both belongs to many calls here. So now we got this set up here and therefore now we should have all the setup we need. If we now again temporarily set force to true to ensure that we do recreate all tables as required and we go back to the workbench and refresh all, we actually only see two tables because we have an error here. If we scroll up a bit, we see incorrect column specifier for column ID, so something went wrong here when sequelize tried to create our tables because it looks like we got some error in one of our models. And the thing is the ID here in cart item, that should be an integer not a string, same in cart, should be an integer not a string. Now this is looking much better and now if we go back and refresh all, now we see carts and cart items here and we can see that in carts, we only got Id created and updated and the user id to which the cart belongs. In users we don't care about that, we only have the user information and in cart items, we have a combination of the cart ID to which this cart item belongs and the Product ID. So now we get everything we need to manage our cart items for the carts we need.

**Lecture 166**

**Creating and Fetching a Cart**

To work with our new cart let's go to the controllers folder here and there to the shop.js file which is where we have our cart related actions. Get products get product, get index, that is all working, get cart however will not work. In there, I want to use the cart associated with my existing user to get all the products in it and render them to the screen. For this, let's get rid of that code but I'll comment it out so that I easily can use my render function again and let's use request user, that's still the user stored in our request. By the way one important adjustment, let's disable that force syncing again so that we don't always override any data we stored. So let's go back to app.js, to shop.js excuse me and let's use the user we retrieve for every new request and which we store in the request and there, let's execute cart and let's look into this and see if this gives us anything meaningful. If I now save this and I go to cart, it doesn't load the page and if I go back we see undefined here, so we can't access the cart like this. But what we can do here is we can use our request user and get the cart which is associated to it. Let's add then and catch, here we'll get the cart and let's log the cart here. If we save that and I reload this page, it still will not load but now I get null here and not undefined. Now the reason why we don't find anything here for either of the two approaches is that we got no carts yet, if we look into our database, carts is totally empty. Just as we create a user right at startup, I'll create a cart for that user, so here I create a user. Now in the next step where I get that user, I will also create a cart by adding user create cart, like this and I don't need to pass any data in there because cart in the beginning will not hold any special data, it just needs to be there. I'll then return this and only listen in the next step where I get my cart so to say and with this now set up, you see an insert into carts call is done here. If we now have a look into our carts, we see now we get a cart associated to our user with the ID 1 and now if I try to reload that carts page again, here we now get some output and this output is stemming from our get cart call here, so from this console log where I do log that cart, so now this is working. Now for completeness sake, let's also console log request user cart again, does this also work? If I reload here, this still is the old log but if I scroll above it, we still see undefined. So this does not work, we can't access cart as a property here but we can call get cart to work with the cart. So now we've got the cart available, we loaded the cart from the database, we know more about that cart here. With the cart available, we can use it to fetch the products that are inside of it by returning carts get products. Remember a cart is associated to products in our app.js file through belongs to many and sequelize will do the magic of looking into cart item and so on, so into this in-between table. So we can get products, this was added by sequelize as a magic method and therefore here in this then block, we'll have products available and here I'll again just log any error I get. But in the then block here, we should have the products that are in this cart and that of course means that we can now render these products. So here I will render my products here, store them in products and pass them to the view. Now this will not work as expected though, if I now reload this page, we see no products in cart. Now we got no products in a cart but we'll see that later once we do have them, it'll also not work but I'll come back to that, for now let's just have a look at our query statement. We can see that this is the statement executed by sequelize and if you didn't see it before, here we can definitely see the advantage we have by using a package like this. We don't have to write that SQL statement on our own, we use sequelize and let it do that behind the scenes. So before I come back to what we will have to do regarding getting the cart products, let's make sure we can add products to the cart. So for now I'll get rid of all that logic here and I will focus on adding products to the cart in the next step.

**Lecture 167**

**Adding new products to the cart**

To ensure that we can add products to the cart, we need to work on the post cart method here. The post cart method is responsible for adding new products to the cart. For this, let's get rid of the existing code there because we'll rewrite it from scratch, well almost, I'll keep the code where we get the product ID because I still need to do that. Next I'll first of all get access to the cart in exactly the same way I did it up there in get cart with request user get cart and then my then block, so I'll repeat that and then have then and thereafter also catch here. And catch as always I'll just log any potential error I get and in then here I simply have access to the cart. So now we have the cart available. Now the third step is that I want to find out if the product I'm trying to add is already part of the cart because if it is, then I just need to increase the quantity, if it's not I need to add it with a quantity of one. So I will return cart get products here with a where condition where I restrict the retrieved products to the product with my prod Id, like that. So now I only retrieve that single product and in the next then block, I will get an array of products as you learned but we know that this will only hold one product at most, it might even hold no product if a product with this ID is not part of this cart yet. So here I'll retrieve my single product as the first element of this array but first of all I need to check if products length is greater than zero. I'll right this a bit differently, create a product variable here and assign a value to that variable if we do have products, otherwise it will stay undefined. So now we do have this check in place, then I'll create a new variable, new quantity and this will be one by default and then I'll check if product is anything but undefined, so if we actually do have a valid product. If that is the case, I need to increase the quantity, so if we do have a product here, I now need to basically get my old quantity for this product and then change it. We'll do this later because right now we don't have any products in there so let's now only work on the new product case. So if we got no product here, we know this product is not part of the cart yet, so what I'll do at this point here is I will return a call to product find by ID because I need to find the general product data for this product now, for my prod ID it's not part of the cart but it's of course still in the database in the products table and here, I will have a nested then call because it makes things a bit easier here because I will execute two very different kinds of code for the case that we have an existing product in the cart or that we have to add a new one and therefore find it first of all. So here I know this is my product as it's stored in the products table and this is the product I want to add to my cart and this can now be done by returning and now I need to access my cart again. Now my cart is available here in this anonymous function but not in this one. To make it available down there too, I'll create a new cart to variable or fetch the cart maybe to not always rename, use the same name and then here I'll store the cart in fetched cart, so now it's not only available here but in this overall function here and therefore down there, I now have access to my fetched cart and on that fetched cart, I can call add product. It's another magic method added by sequelize for many to many relationships, I can add a single product here and I will add it to this in-between table with its ID. So here I add the product I retrieved, I just need to also make sure that I set this extra field I added to my cart item, this is the in-between table but for every entry, I also want to have quantity and if I have more than just two matching IDs, I need to tell sequelize that there is an extra field that needs to be set and I do this by passing an object to add product as the second argument and there I'll add through, you might remember through from app.js, there we use that to tell sequelize which model to use as the in between model and therefore as the in-between table, now I'm telling sequelize, well for that in-between table, here's some additional information you need to set the values in there, so that's another object and there I'm basically setting the keys or the fields that should be set in the in-between table, in this case it's the quantity field which should be set to new quantity. So this is the case that I add a new product for the first time, new quantity will be one here and I'm storing the product with that quantity. Let's see if that works, if I save this and I go to products, got no products so let's quickly add one, just with some dummy data to speed this up. If I go to products and I click add to cart, we're not redirecting which is why we're stuck on this page but we got no error here by the looks of it and if we refresh or load cart items, we see a new product was added or a new element was added to the cart with quantity one pointing at that cart with ID 1 and the product with ID 1, so this seems to work. Now let's just make sure we also well do see our carts page, so here I'll add a then block where I simply call res redirect/cart. So let's try this again now and now I get an error on that cart page and that is what I mentioned before, this will now break. So before we handle the case that we add a product to the cart which is already in the cart and therefore increase its quantity, let's make sure we can see the cart items on this page again.

**Lecture 168**

**Adding existing products and retrieving cart items**

So let's ensure we can see items on the cart page and the problem is that right now, this does not work because here on cart.ejs . we're still accessing product data for every product we loaded but the products we have here are now just the products from our database. So we don't need to access any nested product data instead here, we'll just have p title, we'll have a look at the quantity in a second and pId down there, so it directly access the fields of our products on well, the product we're looping through. Now the quantity is not part of that but of the related cart item you could say and conveniently, sequelize also gives us a cart item key for this which stores information about this in-between table and the entry that is related to this product there. So with this if we save this and we now reload this page, it still doesn't work because that should be cart item with a capital I, this is how I named my model. So now you can see this works and now we're displaying our cart item here and that of course is not quantity but quantity, so the key name we defined in our in-between table and now we see the quantity too. So now we can see the cart items again, now we can go back to making sure that we also handle the case that we add an existing item to the cart. So let's make sure we also handle this case, that we add a product to the cart which is already part of the cart and that should of course simply increment the quantity then. Therefore here if we make it into this if block, I essentially want to get my old quantity first of all which I can get from my product by accessing cart item as we just did it in the view in the last lecture, this is this extra field that gets added by sequelize to give us access to this in-between table and there to the quantity and sequelize does not just give us access to the in-between table but to this exact product in the in-between table, so therefore we get the quantity for this product as it's stored in the cart. So now we have our old quantity and new quantity, the variable we initialize with one here is now simply old quantity plus one. With that we just have to add it to the cart, so here I will return fetched cart to get access to the cart and then add product as I do it down there too, add product and then simply my product here and my through call where I set my quantity to, whoops new quantity just as I do it down there. If we now wanted to avoid nested then blocks here, what we could do is we could add a new then block here where we get some data to which I'll come back, where we do handle this fetch cart thing which is the same for both cases essentially, we add a product with new quantity, so we can certainly remove that but now the difference is that data here actually should hold both the product that needs to be added and our quantity right because the quantity is calculated differently, it either stays at one or here we set it to old quantity plus one. To make sure that we correctly get that data, we can of course pull new quantity out of this anonymous function, make it a top level variable in this overall function here and therefore new quantity is available in all then blocks and we either leave it at one here or if we got a product, we also need to return that product here because that will then be our product we receive in the then block, it's automatically wrapped by a promise as I mentioned earlier and now we have a setup that should work again and that ultimately should ensure that we redirect. So now let's see this, let's click add to cart again and the quantity indeed is two now and if I add a second product here, this one with the price 66 and I add this to my cart, this is now add 1. And that is now pretty neat, we can now start adding all these products. Now let's make sure we can also delete them from the cart.

**Lecture 169**

**Deleting related items and deleting cart**

Our program is taking shape, we can't delete products from the cart yet though so we should definitely also do that. To do that, we still need the product ID we want to delete but now again, I will first of all get my cart for the user by accessing request user get cart and then adding then and catch calls as we did it multiple times already and in the then block here, I know that I've got access to my cart and in that cart, I now want to find the products for this user and to be precise, the products, not all products but the product with that product ID. So here I will return cart get products and then simply here where ID is equal to prod ID. In the next then block we add, I therefore have my products and I have to extract that exact product I'm looking for as the first element in the products array and now I want to destroy that product but not in the products table of course but only in that in-between cart item table that connects my cart with that product. To do that, I can simply call product cart item using that magic field sequelize gives me to access the element in the in-between table and then destroy and that will remove it from that in-between table. Now I will return this so that I can add another then block here with the result of this operation if I would care and then there I'll simply redirect back to the cart, where I should then not see that product anymore. Let's give it a try. If we reload the cart and I delete that one with quantity three, it's gone. Got no errors here and if we reload the cart items, we only have one element in there with quantity two and the other one is gone indeed. So this is how simple we can delete connected items in this in-between table. Here are the example of the cart.

**Lecture 170**

**Adding an Order Model**

So our cart is looking pretty fine and actually we have the old functionality again, now without storing anything in files but now working with a database as we should. Now one thing is missing and that are the orders. Now I'll not add a real checkout flow here but I want to have a checkout button in my cart which will basically for now immediately move all the elements in this cart out of this cart, so clear the cart and instead create a new order, an order that is related to a couple of products and a user. Now if you're feeling fancy, you can certainly try this on your own otherwise we'll of course do it together here and we'll get there step by step adding a simple checkout functionality. Let's start with the model and for this, I'll create a new order.js file and copy my cart item.js file, move it in there, import sequelize, rename it here to order, also here and now how should an order look like? Well an order is in the end just an in-between table between a user to which the order belongs and then multiple products that are part of the order and these products again do have a quantity attached to them. So just as we had cart items for the cart, I'll have order items for my order. So I can copy cart items again, move that into order item and rename cart item here to order item, starting with a lowercase o here in the string name definition and then it will have the same structure as an order, as in cart item here in order item and the order itself will not have anything but the ID because the order essentially is like the cart here for me. You could of course say an order has more information like an address and you could add this here, you just have to make sure you add this for every order you create but that is essentially it. Now regarding the relations, there is one important difference to the cart though. If we go to the app.js file where we do set up all the relations, I'll import my order here from the order, whoops, from the order file and I'll import my order item from the order item file but now if we connect the models here at the bottom, we can say that an order belongs to a user and it doesn't belong to many users because a single order is always belonging to one user who placed the order and the user may of course have many orders, like this. So this is not a many-to-many relationship, it's a one to many relationship, one user can have many orders. Now an order however can belong to many products and it does so with an in-between table which we specify with through which is our order item table, like this and we can also define the inverse here if we want and that would be basically that a product belongs to many orders but we can also leave it out here. Now we get this set up. now let me make sure we can sync again so let's turn on forcing this. If we now refresh our database, we should have new orders and order items, in orders we see a connection to a user and in order items, we see a connection to an order and to our product ID then. So this is now the set up we need to use and I'll disable forcing this so that we don't overwrite tables all the time now. So as a next step, let's make sure we have a checkout button and that this button does actually then trigger something to create such a order.

**Lecture 171**

**Storing CartItems as OrderItems**

Time to add the checkout button. For that I'll go into my cart view here and in the cart view, I'll add a button below all my list items, so below the unordered list here but still in the first if block, I'll add it inside a form and that button will get a class button where it will have a caption of order now let's say, it will be of type submit and the form will have an action that leads to let's say create order, the name is up to you, whoops no comma here and a method of post. If I save this, we can go back to the cart page and reload it, now we got no product in there because I recreated all table so let's quickly add a new product and let's add it to the cart. Now it's not centered, let's do that by wrapping our form with a div which gets the centered helper class I defined earlier, so it's already part of your css files, save this and now we have the order now button here, let's also maybe add a horizontal line to have some space. And with that set up here which is good for me now, I want to click that button to move the cart items into order items so to say, so to create an order with all the elements in there. And to see this, I'll add a second product here real quick so that we have one more item, one and more items in the cart. So now order now should create a new order with these two items and clear the cart. Let's go to shop.js and make sure we have an action for creating a new order, so here I will go there and exports post order which has the normal middleware function as we know it, I'll also create the new route for this to handle a post request to this order route, so a post request to /create order. For that let's go to routes shop.js and let's register a new route here, router post create order and there, I'll use my shop controller post order, this new action I just added which doesn't do anything yet. So post order here should now take all the cart items and move them into an order. For this, let's first of all get all the cart items by accessing the request user and then calling get cart. This gives me access to the cart as we did it before, so here I'll console log any errors I might have and then here, I simply have access to the cart. Now with access to the cart, we can get access to all the products in there with cart get products and this will return all products by default, so now here I can call then, products and I can simply console log my products here if I want. If we now click this button, it won't do anything here but in the console, we can see the products that were retrieved and we see that the products that were retrieved are the products here which also have that cart item attribute which in turn gives us information about the cart item in this in-between table. So this is looking good, we got access to the products, now the idea is that we move the products into a newly created order. For this let's import the order model here first of all, so time to import order here from the order file. By the way, we don't need the cart import here because we never directly use the cart model, we always do so through the user model but we'll need the order model here or do we? Well just as a cart is related to user, so is an order, so we don't even need that import, we can clear both cart and order because we'll create a new order as an order that is associated to a user. So in post order here, we can now call request user and just as we create a cart for that user in app.js with create cart, we can now call create order for that user. Now this gives us an order but we don't just need the order, we also need to associate our products to that order, so here I'll return request user create order. And with the order created, and here I'll again do this nested, you can always restructure it to not use a nested promise here though if you want but here I will get my created order and now I want to associate my products to that order and that can be done easily by calling order add products and passing my products here. Now important, we need to specify true and set the quantity here correctly too but now which value would we assign there because we get different quantities for all the products? The approach is a little different, we don't pass it like this, we just pass products to add products but each product needs to have a special key, a special field which is then understood by sequelize. Now to assign that special field I'm talking of, the products we pass in here have to be modified and we can do this with the map method, a default javascript method that runs on an array and returns a new array with slightly modified elements. We add a function here that is executed for every element in the array and takes the element as an input and returns the modified version. I'll return products here in the end but before I do so, I do edit it slightly, a new property which sequelize will look for named order item. Now the name here is important to get this right. If in your order item model, you define this name, that is the name you have to use, if you chose a different name, you have to use the different name. So here I have order item with a lower case o and a capital I and therefore here, I have product order item written in the same way. This now stores a javascript object where I configure the value for this in-between table, so here I simply define quantity which is the value I need to store in between and I set this equal to product cart item, this is the related table I have due to my cart, quantity. So I get the quantity from the cart and store that for the order item, this then gets returned here, so now in the end I have an array of products with all the old product data but also this new information regarding the quantity for my order and add products will pick this up and add the products to the order with that quantity. This is what's happening here, now we can return order add products here and add a new then block here where I get any result and in here, I will then redirect to orders. With that set up, let's go back and reload our cart page, if we click order now, I'm on the orders page where we never display anything but we should be able to see some data if we load the orders table, there is one order and order items also has the respective elements that belong to the order with the right quantities.

**Lecture 172**

**Resetting the Cart and Fetching and Outputting Orders**

Now that we can add orders, one thing is missing and that is that we clear the cart, I want to do this right after we edit the orders. So down there I need to work with the cart and therefore first of all, I'll store it in a new variable, fetch cart, initially it's empty and here once I got the cart, I'll store the cart in the fetch cart variable so that I can also use it down there. Now that cart should essentially drop all its cart items. Now I can use my fetch cart here to call another method, set products for this cart and set the products to null. Now let's return this and then add a new then block with the result of this operation where I simply want to redirect. Let's give it a try, let's save this and reload the cart page here, click order now, we're on the orders page, go back to the cart and we got no products in the cart because if we go into the workbench, we see that the cart items indeed are empty because we dropped all the items in the cart by setting them to null, this is how easy we can clean up the cart here. So with that, we got all that logic done, the last step I want to do here is that I actually show my orders and we can get rid of the checkout page here for now because we have no such step at the moment. So now let's make sure we also retrieve the orders correctly and can display them on our orders page. For this here, I'll first of all get my user and now I'm interested in the orders of this user which I can get with get orders, a magic method added by sequelize, thanks to our associations. Here we again get a promise, let's log any potential errors we get and in the then block, I know that I have my orders. This is where I want to render my orders page therefore and I will pass a new variable into that page, the orders variable which simply stores all the retrieved orders. So with that I got my orders for this user, now let's go to the respective view in the views folder, here's the orders.ejs file which always shows nothing there. Now obviously that's not what I always want to show, instead I only want to show nothing there or no orders placed yet, whatever you like if we got no orders, so I'll first of all add a normal ejs statement where I check if orders length is smaller or equal than zero, then we probably have no orders, so in this case I want to display nothing there. Now else and that is of course an important part, let's also close that statement of course with the closing curly brace, else I want to output my orders and for now I'll do this in a very ugly way with an unordered list of list items which I repeat for every order, so again ejs time to loop through all orders with forEach for example. Now forEach as you know takes an anonymous function which gives us access to every order then we repeat this code here for every order and then here, we close the curly brace and the bracket of our forEach method, we can add a semi-colon if you want and now this is executed for every order we got. Now every order we got has a couple of products that belong to the order, so for every order I have here, I will output a h1 tag with the ID, order.Id, output like this and below that another nested unordered list with more list items where I loop through the products belonging to the order, so here I'll have order and then it's order item, that is our connected model so to say, forEach because that will be a list of all the related items and here I get my item. Let's close this here and let's also close the forEach syntax then down there and now this list item is repeated for every item in this order and there I simply want to output the title, so output item title. Keep in mind this is a product so it will have a title and then maybe in normal parentheses, I'll output item quantity and this will not work, I can already tell you that. If I save that, I get another error actually and that is simply coming from the fact that I deleted one route, the checkout route, we need to delete it here too so I deleted the action, we need to delete the route too but still if I reload the orders page now, this will just crash here because I have an ejs syntax error though, let's quickly fix that. Ok here for item quantity, you need to close that but even after this change, this will not work now because order is not defined because that should be orders forEach but now because we can't loop through order item here, this does not work. And to understand this, let's have a look at our shop.js controller file and let's output orders here and let's see what exactly we get. For this let's now reload the page, it will still break of course but now we can have a look at the output there, we want to scroll up. There is what we log with this line, console log orders and there we can see that we do have an array of orders but an order does not have an order item key, this is not provided by sequelize. If we also want to fetch the related products to an order, we have to pass an object here where we set include to an array with the field products or the element products as a string. Now why products? Because in app.js, we associate an order to many product, products of course and if we have a look at our model, the product model has the name product. Sequelize pluralizes this and then we can use a concept called eager loading where we basically instruct sequelize hey if you are fetching all the orders, please also fetch all related products already and give me back one array of orders that also includes the products per order. Now this only works of course because we do have a relation between orders and products as set up in app.js here and now we can load both together. This will still not make our template work immediately but now we got orders with more data in them. Each order will now have a products array and with that in mind, we can go back to our view here, to the orders view and tweak that. We can loop through the orders and every order will have an ID, that's fine. Now our order will still not have a order item but it will have a products list and now we can loop through all the products and each item which is now simply a product and therefore we can name it as such to make this clear, each product of course has a title which we can output and it does not have a quantity but the product will now have an order item key which then stores the quantity for that product in that order. And now if we reload this page, we do see our orders with the nested products in there, not the most beautiful presentation you have for sure but this is how it actually works. We also see there is one dummy order that was created accidentally in between with no products and we can always verify this by looking into the database, we get four order items related to orders with the ID 9 and 11 in the orders, we got 9, 10 and 11 so indeed there is the order with the ID 10 which has no items. We can simply clear it by right clicking on it here, delete row, apply apply and close and now if we reload the orders page, it's gone and we only see these two orders. So this is now working and now we get the full flow, we got a user who is related to created products, we can edit the products, we can delete the products, we can add products to the cart. We can then also create an order based on the data in our cart, we can view product details and so on. So this is all working fine as it should and with that, we get the set up we need for now. This gave you an introduction to sequelize and definitely dive into the official docs to play around with that and learn more about it and how it works.