**Lecture 209**

**What is Mongoose?**

So what is mongoose? Mongoose is an ODM, an object document mapping library and that's really similar to sequelize which was an ORM, an object relational mapping library and the difference of course just is that mongodb is not a relational database, it's a document database, it thinks in documents and hence we have an ODM here. So the idea stays the same though, we have some data, some entity in our application, let's say a user and we want to save that to a collection, we want to map our javascript object to a document in a collection that could look something like this and of course we can write the query for that on our own, that is exactly what we did in the last module but it would be a bit easier if we could just focus on our objects, on our data and see how it should look like and then work with it and this is not even the final syntax you see here, we can use mongoose a bit differently than you see it here but even that would be a bit more concise. So just as sequelize, mongoose tries to allow us to define models with which we then work and where all the queries are done behind the scenes which of course does not mean that we can't influence and that we can't change some things. The core concepts are that we work with schemas and models where we define how our data should look like and then we have so-called instances where we instantiate our models, so where we create real javascript objects we can work with that are based on our blueprints and once we get that setup, we can run queries and there we again use our objects, we use our models and we can then query the database but through mongoose with various helpers we get, some syntactical sugar and so on. So that's the idea behind mongoose, really similar to what sequelize did for SQL and therefore with that, let's dive in, let's add it to our project and let's see how we can use it there.

**Lecture 210**

**Connecting to the MongoDB Server with Mongoose**

Now first of all let me show you the official docs, you can always visit mongoosejs.com and there you can dive into the full docs to learn everything about mongoose, all its details, all its advanced features which we'll not cover in this course because this is not a mongoose course, I just want to give you an introduction to mongoose here, so everything can be seen in great detail here but that being said, I will walk you through all the core fundamentals in this course and in this module of course and we'll also keep on using mongoose for the rest of this course so we'll gradually use more features as we add more features to our application like authentication and so on. So this is how you can learn more about mongoose, now to add it to our project, I quit my npm start process and then I can simply run npm install --save mongoose, like that, that is the name of the package. Now this will download and add it to our project and then we are already ready to start using it and the first thing we want to do when we start using it is that we want to connect to our database and for that, we could use our database utility file here but actually mongoose does all of that utility management and the management of that connection behind the scenes for us. What we can do is we can get rid of the database.js file and we can go to the app.js file and in there, simply import mongoose. So here let's import mongoose by requiring mongoose, whoops, mongoose like this and with mongoose imported, we can set up a connection. So down there where I used my own mongo connect, I can use mongoose like this and then there, there is a connect method we can use. Now that connect method takes the same url we used for connecting before, so make sure you grab that url from your backend again and enter that here, make sure you're using the right user which you set up for that and of course enter your password and this will now connect to mongoose. Then you get a promise here or you can call then at least and there you'll get the result of this connection but most importantly, you know that at this point of time you are connected, so here I then also want to bring up my node server or to be precise, start listening for incoming requests. I can also add a catch block here and log any potential error I might be getting when trying to get connected. So with this, we already have everything in place we need to connect and mongoose as I mentioned will manage that one connection behind the scenes for us so that in other places where we start using mongoose from the mongoose package, we use that same connection we set up here, really convenient of course. And that already leads us to the next thing we have to do. We have to move our code over to mongoose now and not to use the mongodb driver directly as we are currently doing it. So we'll again have to rewrite all our models but I hope you still see the advantage of this, you first of all learn how to use the mongodb driver and you could of course stick to that and continue using that if you wanted to but now I will show you how you can move that over to mongoose to focus more on how your data should look like and less on all the logic for interacting with the database. Now to move over, again I do something I did before, I will comment out my different routes here in the shop.js and in the admin.js file so that we can gradually comment them back in as we make them work again. Now with all routes commented out if I run npm start, I get an error of course that util database was not found, in app.js I certainly have to get rid of that import here where I do import something from that database and the same is of course true for my models where I do also import something from the database. Now the problem is if I comment that out, nothing will work here anymore and therefore I will simply comment out everything in my models for the moment. Now with that, it looks like we are connected because we don't get any error here, this warning can be ignored and therefore we now are connected to our same mongodb server but now by using the mongoose package. And with that set up, let's start working on all our models and all our code to make it work with mongoose in the next lecture.

**Lecture 211**

**Creating the Product Schema**

Ok so time to fix our code and make it work again. And for that first of all, I also connected to my mongodb server with compass again and there I want to clear everything so that we can start from scratch essentially and therefore I will go to my shop database and simply delete that entire database, it will again be created on the fly or when required when we start inserting data. Now I've got a problem, I connected with the wrong user where I'm not allowed to delete a database because I connected with a user who has only read or write access, so I'll just delete the collections here of course, the alternative would be to simply connect with a user where I am allowed to manage the overall database but this will do for now. With that I also implicitly got rid of the shop database and now we can start working from scratch again and let's start working with it by making sure that we can create products and for now, products that will simply use very simple data without taking the user into account for now and where we can simply add them through the admin controller. So I'll go to my product model, to the model file and there I will eventually remove all the commented out code, I'll leave it here for reference for now and first of all in here, we need to import mongoose. So I will require mongoose here, whoops, mongoose like this. Now with this imported, I'll create a new constant and I'll name it Schema with a capital S, the name is up to you but I will use something from the mongoose object I imported, I will use the schema constructor here. This constructor allows me to create new schemas and here I will create a new schema, a new product schema, you can also name this however you want, I create a new schema by instantiating a schema object by calling new schema using that constructor here. Now this is just how mongoose works and to this constructor here, I pass a javascript object and in that object you now define how your product should look like, so you now define the data schema of a product in our case here, so this is how you set up such a blueprint. In that object, you define a schema with simple key value pairs, so for example our product will have a title. Now you don't just define which keys you have but also which type these keys will have and that's important and the type I define here would be string, string is a default javascript object hence we can use that here. And this would say ok so here I create a schema for an object which I will eventually be able to work with which must have or which will have a title that is of type string. Now important in the mongodb module, I mention that mongodb is schemaless so why do we now start to create schemas? Well the idea simply is that whilst we have the flexibility of not being restricted to a specific schema, we often will have a certain structure in the data we work with and therefore mongoose wants to give you the advantage of focusing on just your data but for that, it needs to know how your data looks like and therefore we define such a schema for the structure our data will have. But important, we can still deviate from this by assigning a title like this, we could even work with a product and create a new one and save it to the database without setting a title because we still have the flexibility of not enforcing this, though what we can do is we can pass an object instead of just the type here as a value and then set a type property which could be set to string and then set required to true, this is basically a more complex way of configuring the value for this key and here we would say well the type of this is a string as before but it's also required and now we indeed give up some of the flexibility we had before and we force all objects to have a title but in the end in our application, every product needs to have a title indeed because we will run into other errors otherwise, for example when outputting products in our views. So having some kind of schema makes sense even though we have the flexibility to deviate from that and it really just depends on your application, whether you need all that flexibility or whether you want to have some structure and you want to have some tool, mongoose that takes some structure and then helps you work with that data. So that was a lot of talking about schemas but it is important that you understand why we now all of a sudden start working with them, it's just a deliberate decision to give up some flexibility but gain other advantages. Now of course a product doesn't just have a title, we also need to have a price and there we can set the type to number for example and then this would also be required and we also want to have a description and image url. So we'll add a description here, this will be of type string and let's say this is also required. Now of course you can always deviate from my setup here but I want to require that all and we'll have an image url and in that image url, this will be of type string because it's just a url and this will also be required. So that would be my product schema, this is basically a description of how a product should look like in my application. Please note that I don't add \_id here because this will still be added automatically as an object ID so we don't need to define that here, the user ID is something we'll add later. So now we got that product schema defined, in the next step we will create a model based on the schema and then create an object based on the model and work with that.

**Lecture 212**

**Saving Data Through Mongoose**

So we got a data definition defined, we got a blueprint, we got a schema for our product. Mongoose now also works with so-called models and the model is also what we'll export here, so I can already say model exports and now what I want to export is mongoose model. Now model is a function I call and a model basically is important for mongoose behind the scenes to connect a schema, a blueprint with a name basically, so here you give that model a name and that name here would be product. Typically you name it here like this with a capital starting character and then simply just well the name of the entity this reflects in your project or in your application. The second argument then is the schema so in my case that product schema we defined and this model is what I export because this model is what we'll work with in our code. So with that model defined here, we can now move over to the admin controller where we have post add product where we do save a new product and there indeed I want to create a new product and then I want to be able to save that. Now for that, we can basically keep the code we have here. We still import product from our models folder from the product file because I do export a model here and we can basically use that just in the way I used it here. One adjustment is required though, to the product constructor we don't pass multiple arguments like this, instead we pass one argument only and that one argument is a javascript object where we map the different values we defined in our schema. So here I would map the title to the title and you basically now have to go through all the fields you defined in the schema, the order does not matter though since it's in a javascript object, so you then map let's say price to price, you map description to description and you map the image url to image url and just in case you're wondering, the part on the right side of the colon, so title, price, description, image url refers to the data you receive in your controller action and the part on the left side of the colon refers to the keys you defined in your schema, so to these things. And now with this, this creates a new product based on our model and therefore a product managed by mongoose you could say and indeed such a product happens to have or such a model happens to have a save method now provided by mongoose, that's really important, this is not defined by us. In the product model, we define no save method, we did before but this is commented out. So we're not defining a save method, this save method here is coming from mongoose and then we can indeed call then on that, technically we don't get a promise but mongoose still gives us a then method, it also still gives us a catch method we can call and therefore this code should actually continue to work. Let's see if that is the case and let's head over to our application and click on add product. Now I do get an error from app.js because there I of course still use my user model, let me comment that out and let me comment it out here, let me comment out that entire middleware, for now we have no user and otherwise all requests will break for now so let's remove all that user related stuff in the app.js file, this middleware and the user model import and with that removed, let's try reloading that add product page and obviously we need to import the route again, so we should do that. Not only the get route but also the post route, so make sure you re-add both admin add product routes, this was the wrong one, this one is required. So make sure you have the get and the post and the product route added again because we manipulated these two admin controller actions and now with that, finally if I click on add product, this works and now let's test this, let's test a test product with some price, some image, does this work? Click add product, we get a page not found because we can't load any other pages, that is ok. In the code, I can't see an error but I got created product which looks good. And in compass let's refresh by clicking that icon on the top left and I connected to the wrong url, I'll fix that, therefore I'm connected to the test database instead of the shop database but theoretically it worked, theoretically we get a products collection with the product added. Now where is the products collection coming from you might be wondering by the way because we never defined that name? Well mongoose takes your model name, so product, turns it to all lowercase and takes the plural form of that and that will then be used as a collection name, so this is why our product here has a great impact because it was used for naming this collection and here we see the data we entered. So this is working, that's all great, I will still drop this collection here and quickly fix my connection settings in app.js at the bottom of the file. Here I do connect to the test database, I don't want to do that, I'll connect to the shop database instead just as we did it before. And with this tiny change, if I now go back to my application and I try adding this again, a book, now let's use that book url I used before, must read and click add product, now this still works in the same way it did before but now if I refresh here, my data is saved in the shop database and there in the products collection. So this is looking really good, we're now able to save data through mongoose.

**Lecture 213**

**Fetching All Products**

Now that we're able to save products, obviously we again want to be able to fetch them. So let's head over to the shop.js file in the controllers folder and let's start working on get products and get index actions there so that we can indeed fetch all products. Now we got no fetch all method in mongoose, that was a method we defined on our own but we can still use something on the product model which we're still importing from the product models folder and which is the mongoose model in the end and there we got a couple of static methods, you can find all in the official docs of course and one of them is the find method which you already know from the mongodb driver. Now find works a bit differently when used with mongoose, it does not give us a cursor instead it does give us the products, we could add .cursor and call this to get access to the cursor and then use each async which would allow us to loop through them or next to get the next element but I will just use find and this will essentially give me all my products automatically. So then in products, I should get my products and I should be able to output them, so if I now console log my products here so that we can also see them, we have to go to the shop route as well and there we want to include the get index route and the get products route and since I also did include the get index route here, I of course also need to work on get index in my shop.js file in the controllers folder, there we also want to use find instead of fetch all but that should be it. If we now save everything and we go to the products page, indeed I do see my product here so this is looking good and if I go back, well then I see here is the output of the data that was fetched and you see I get an array here because find when used with mongoose automatically gives me that array here. Now again if you know you will query large amounts of data, you should turn this into a cursor or of course manipulate find to limit the set of data that is retrieved, something we will see in the pagination module later in the course. So here I now got my working get products and get index route, let's next work on the detail route for a single product.

**Lecture 214**

**Fetching a Single Product**

Now that we're able to get all products, let's also make sure we can get a single product and for that, I'll work on my get product action in the shop controller and there we extract the product ID which is great, I don't need this code here anymore by the way and the good thing is product is a mongoose model and mongoose indeed has a find by ID method, so little convenience method that defines for us. So again find by ID here is not our own method, it's defined by mongoose. And best of all, we can even pass a string to find by id and mongoose will automatically convert this to an object ID, so it will handle that for us as well, so again getting a lot of that convenience by just using mongoose and then we should be able to get back a product and use that. So actually this should be everything we need to do, nothing basically I guess. One thing I have to do of course is I have to comment that route back, so this route for loading a single product. With that if you save everything and you click on details here, we indeed see the detail page for this product. So this is now also working and that of course is really great because well this allows us to easily adjust our code to use mongoose.

**Lecture 215**

**Updating Products**

Ok getting a single product was real easy, now let's complete the crud functionalities back on the admin side by making sure we can also edit and delete products and for that, let's start with editing. First of all get edit product should load a product and because as I explained in the last lecture, because we would have find by ID by default, I can leave that code as it is, I can leave that code as it is and I will get my product and I shouldn't be loading that into my edit product page. So for that, I'll just need to go to the admin routes file and there I should comment my get products routes back in, I'll work on this in a second too and also my edit product, get edit product route, I should comment that back in as well. Now let me quickly work on that route where I load all products, that is also something we need of course, we can find that here, get products. I mentioned fetch all does not exist but find does and find also already gives us the documents and not just a cursor so we can leave that code as it is and with that if we save that, we won't be able to submit a change but we should be able to see all products and to then click edit here and it is populated with our data, so that is looking great. Now the next step is that we are able to click that update product button and that this works of course. Now for that, let's work on the post edit product route in the admin controller, we extract all the data we require which is of course great and then here we need to change that code a little bit. Instead of creating a new product and calling save, I will fetch a product and I'll fetch a product by ID with the prod ID, add a then block and in that then block, I know I have access to my product right, to the product which was fetched from the database. The cool thing is I can now move product save into my function here and call save on the product that was fetched from the database because thanks to mongoose, this will now not be a javascript object with the data but we will have a full mongoose object here with all the mongoose methods like save and if we call save on an existing object, it will not be saved as a new one but the changes will be saved, so it will automatically do an update behind the scenes. The only thing I need to do is I need to set my fields, so I need to set product title to the updated title, I need to set product price to the updated price, I need to set product description, whoops, description to the updated description and I need to set product image url to the updated image url and thereafter, I can call product save and I can then return this and chain then here. So now I have a setup where I first of all find the product and I get back a full mongoose object hence I can manipulate it and call save again, I return the result of that and then call then on that to redirect once the saving was done. Let's see if that works, if I save my code and I adjust the price and let's say I add an exclamation mark after the title, if I now would click update product, I actually would fail because I need to comment in my route again, I just remembered, here I need to comment in that post edit product route, so make sure you comment that in and save your server side code. You don't need to reload that page, you can now just click update product and this is looking good. I can already see my changes here so this seems to have worked and I can also reload compass and of course I see my changes there too. So this is how we can update with mongoose, again very similar to how we did it with sequelize all through logic and through manipulating our data.

**Lecture 216**

**Deleting Products**

Last but not least let's make sure we can also delete data so that you see that last part of the crud operations with mongoose as well. For that, we got our post delete product action and I'm calling delete by ID here on my product model. Now if I have a look at delete, you'll see this is not something offered by mongoose, however we do have find by id and remove and that is exactly what we want here right, so if we call that then we've got everything we need, this is a built in method provided by mongoose that should remove a document And now we can go to the admin routes again, comment in that last admin route again, save that and with that being saved, if I now click delete, no products are found and we can of course also confirm this in compass if I refresh, there are no documents in the products collection. So this is how we can work with mongoose and do basic crud operations with it. Now let's again add a user and see how we can relate different entities with mongoose, so how we can manage relations and let's then add that cart and orders thing again.

**Lecture 217**

**Adding and using a User Model**

We worked on the admin side and on the basic product, let's now add a user again and let's see how we can add a user model, how we can connect it to a product and so on. For that I'll go to my user.js file in the models folder and first of all, I want to import mongoose here by requiring mongoose, like that. So let's now work on the user schema. Well first of all, we need define the schema, so the user schema. The user schema is defined by using something from the mongoose package which you don't have to but I will store in a separate constant, so mongoose schema. With that we can call new schema down there and I pass a javascript object to that constructor and in that object, we describe how a user should look like. Now here I will say that I want to have a name where the type of data will be a string and this will be required. As a next field, I want to have an email as well, this will be of type string and this will also be required like that. Now we also used a cart before and I still want to embed the cart into my user document, that hasn't changed just because we use mongoose. So we'll add my cart here and now in that cart, that will be an embedded document and we can define it just like that, I will add items here and items will be an array and this is how you define that you want to store an array in here, you simply create an array and then you add, what is inside of this array? You could say an array of strings if you have an array of strings or numbers or booleans or an array of well documents, that of course is also possible. Now I want to have an array of documents where I have a product ID which I'll configure with this document here, I'll come back to that and I'll have a quantity and the quantity I'll configure that right away will have a type of number and is required. Now what's the product ID here though? The product ID will have a type of and now I need to get something from the schema, there we have a types field and there we got all these special types like object ID, so I'm telling mongoose that this will actually store an objectid because it will store a reference to a product and that this also is required. Now that is a bit of a more complex schema therefore because we have an embedded document and we have an array of then even embedded documents but this is how a user should look like. Now just as before, I'll export this by calling mongoose model, give this a name and the name will be user hence this will be stored in a users collection because mongoose will automatically take the plural lowercase version of that as a collection name and I pass my user schema here. Ok so now we get everything we need to inform mongoose how a user should look like, now of course I want to work with a user. And for that I'll go back to the app.js file and I will actually create a user here before we start listening. For that let me import the user model again in app.js and with that import added, I can create a new user by simply calling new user and just as before, I pass a javascript object where I configure it, where I assign a name, where I assign an email and where I also need to set my cart and that cart will have an array of, well empty items. So this is my user, I can then call user save and this will be done when I start my server. So here after it restarted we should have a user already, if we refreshed the overall database server, we indeed see our users and there we see one user with that nested data. So now with that user created, it's that ID I'm interested in, so let me copy that ID and go back to the app.js file and comment this middleware back in. Here I again will find a user by ID, I just need to paste in that ID of that user we just created. Find by ID is a method provided by mongoose so this will work, we get back the user here then and then I can simply store that user in my request and keep in mind, this is a full mongoose model so we can call all these mongoose model functions or methods on that user object and therefore also on the user object which I do store here. So this should work, if I now save this for every incoming request, it should actually give us that user. Now one issue is that with every restart of the server, we also create a new user as you can see here, so let me delete that new user because I have my user creation code down there and I don't check whether I want to create one or not. So what I'll quickly do is I will first of all see if we do have a user with user find one and find one with if I give it no arguments will always give me back the first user it finds and then here in the then block, I will have my user object and only if this is undefined so if it is not set, only then I will create a new user. So therefore now if I refresh this we shouldn't have a new user, if I however delete that first user here and I therefore temporarily get rid of that let's say and I save and it restarts, now if refresh my users collection I do have a new user and now I just need to grab that ID and paste it back in here and now we get a set up where I don't constantly create new users. Ok so this is how we add a user model and how we basically use it but now of course we want to use it in conjunction with the products model and with the products. So let's do that in the next lecture.

**Lecture 218**

**Using relations in mongoose**

With the user model set up, let's make sure we can use it together with the product model. Now obviously every product should be assigned to a user, so first of all we need to change our product schema a little bit. A product should also have a user ID field let's say, just as we had it before in the last module. Now a user ID field is of type and now which type is this? Well it will be a reference to a user, so this will actually be of type schema types objectID and now we can set something special here, we can add a special ref configuration and ref takes a string where we tell mongoose hey which other mongoose model is actually related to the data in that field. We know that we will store a user ID here but just because the type is objectid, this is not obvious, this could be any object ID of any object. So I will add user here and you use the name of your model to which you want to relate this, so since our model here is named user, I will name it user here, so I refer to my user model here and with that I got a relation set up. This also means that in my user model where I do store the product ID, I can also add a reference here and refer to product because I know that for every user in the cart items, I will store products where I refer to some ID and that ID happens to refer to a product stored or defined through the product model. So now we got relation set up with ref, of course you only need this when using references, when using embedded documents as we do with the cart, you don't need to do anything because well you use an embedded document, this already has kind of an implicit relation that is managed inside of one document. So now with the references set up, let me also add a required to true here and now I adjusted my schemas. Now with the schemas adjusted, we of course need to adjust our code like in the admin controller when we create a new product. When creating a new product, we also want to make sure we store the user id and that is super simple. We just add user ID here to the fields we pass in the object to the product constructor and now remember that we did save the user in our request, so I can just do request user\_id like this and this should give me access to the user id and assign it here. Now conveniently in mongoose, you can even just store the entire user object, so this is really the entire user object not just the ID and mongoose will just pick the ID from that object, so that's another convenience you got there. Let me save my code, let's go back and let's add a new product, a nice book. Let's use that image url we used before, let's give it some price, you should not miss that and click add product. Now you see that looks good, it was added and now let's have a look at compass and see if the user ID was added too. So if I refresh there, indeed we see the user id here and just the ID, not the entire user, just the ID as we want it. So this is pretty awesome, this is how we can manage relations through mongoose. Now let me show you something cool when it comes to fetching these relations.

**Lecture 219**

**One important thing about fetching relations**

With the user model set up, let's make sure we can use it together with the product model. Now obviously every product should be assigned to a user, so first of all we need to change our product schema a little bit. A product should also have a user ID field let's say, just as we had it before in the last module. Now a user ID field is of type and now which type is this? Well it will be a reference to a user, so this will actually be of type schema types objectID and now we can set something special here, we can add a special ref configuration and ref takes a string where we tell mongoose hey which other mongoose model is actually related to the data in that field. We know that we will store a user ID here but just because the type is objectid, this is not obvious, this could be any object ID of any object. So I will add user here and you use the name of your model to which you want to relate this, so since our model here is named user, I will name it user here, so I refer to my user model here and with that I got a relation set up. This also means that in my user model where I do store the product ID, I can also add a reference here and refer to product because I know that for every user in the cart items, I will store products where I refer to some ID and that ID happens to refer to a product stored or defined through the product model. So now we got relation set up with ref, of course you only need this when using references, when using embedded documents as we do with the cart, you don't need to do anything because well you use an embedded document, this already has kind of an implicit relation that is managed inside of one document. So now with the references set up, let me also add a required to true here and now I adjusted my schemas. Now with the schemas adjusted, we of course need to adjust our code like in the admin controller when we create a new product. When creating a new product, we also want to make sure we store the user id and that is super simple. We just add user ID here to the fields we pass in the object to the product constructor and now remember that we did save the user in our request, so I can just do request user\_id like this and this should give me access to the user id and assign it here. Now conveniently in mongoose, you can even just store the entire user object, so this is really the entire user object not just the ID and mongoose will just pick the ID from that object, so that's another convenience you got there. Let me save my code, let's go back and let's add a new product, a nice book. Let's use that image url we used before, let's give it some price, you should not miss that and click add product. Now you see that looks good, it was added and now let's have a look at compass and see if the user ID was added too. So if I refresh there, indeed we see the user id here and just the ID, not the entire user, just the ID as we want it. So this is pretty awesome, this is how we can manage relations through mongoose. Now let me show you something cool when it comes to fetching these relations.

**Lecture 220**

**Working on the Shopping cart**

Ok so now that we're able to relate data and store products that are related to a user, let's work on that shopping cart and the orders again so that we can do more to just look at the products. Now previously, in our user model in the part which is now commented out, I had utility methods like add to cart to add products to the cart of that user and actually it was really useful to have these because that allowed us to move logic from our controller into the model which is typically where your data related logic should live. So therefore I will re-add it and mongoose makes this really simple, you can work on your schema so on my user schema here and then there you'll have a methods key. The methods key is an object which allows you to add your own methods by simply well adding them, Add to Cart and this should now be a function and the important part is it has to be a function written like this so that the this keyword in there still refers to the schema and not to something else and now in this function you can add your own logic and that is exactly what I want to do. In this function, I want to add the logic I had in Add to Cart before and tweak it a little bit. So let's grab all that code from Add to Cart, let's add it here and comment it back in. Now the function here should also receive the product which I want to add, that is something we required in the past as well, so we still will get a product to that function, this is something we just need to assume because we are the one writing the code in the end. And now in there, we should still be able to use these cart items because our schema has a cart and then an items array, so this should still work and keep in mind this will be called on a real instance based on that schema, so really on an object which will have a populated cart with either an empty array of items or an array of items with items in there. So this code should still work for getting the product index. I also still want to control my quantity as before and update my cart as before by first of all copying it, then I will keep the logic of checking whether we already do have the product in the cart in which case I'll calculate the new quantity based on the old quantity and then I'll update my array or if I will update my array by pushing a new object onto it. Now here the thing just is this will not work like that but I can just store the product ID like this and mongoose should automatically wrap it in an object ID, the quantity can be stored like this however and this should fit my schema of an item with product in the quantity, so make sure that the names you used up there in your schema are the names you use down there for creating new data. So with that, I got my updated cart, now I don't need to get access to the database like this instead here, I will indeed return something but I will not manually update this instead I will just call this save which should work, before I do that I just need to set this cart equal to the updated cart. Now this should be a utility method that saves itself so where the object saves itself by using the built-in save method where we update the cart, now let's see if that works as it should. Let's go to the shop.js file and there we have all our post cart method. Now I find my product by ID which still should work because mongoose gives us this function and then I do indeed call request user add to cart which should now also work because we just added this method to the user object in our user model, so this should work without any changes hopefully, let's go to the routes therefore and comment that post cart route back in. If I now save everything, let's go to products and let's try add to cart, I do get some output here and let's now have a look at compass and looking into the users, there I have my cart, I got my items and I got an ID in there. Now that was added by mongoose automatically, it adds IDs for sub-documents as well but most importantly I got my product ID and that product ID which ends with 3C6 should be the ID of that product and it is. And now let's confirm if that really works by going back again and adding that same product to the cart again, it still doesn't find the page it wants to load but that is alright. Let's update our users collection now, look into the cart, we got one object with quantity 2. So this is working and this shows us the power of mongoose, we get a bunch of nice built-in methods which we also leverage here in our own method which we can add and which we then can call on our objects and are based on our schema.

**Lecture 221**

**Loading the cart**

So in the last lecture, you saw how you can extend mongoose with your own functionality and that is really powerful. Now obviously, also check the official docs, there under schema you can learn way more about that, this is the technique we just used, we added an instance method, you can add more about that and all the nice features you can add to your own schemas and models, so definitely check that out for all the details that could be interesting to you. Now here I want to work on the route that allows us to load the cart and this is a nice practice for you again, feel free to pause the video at this point and go ahead and try to implement this on your own before we then do it together. Were you successful? Well let's go to the shop.js file in the controllers folder and we're looking for the get cart method here. Now there we have request user get cart and get cart is a method we defined in the past which does not exist by default, mongoose does not give us this method. So we again have to revisit our user model and in there, let's have a look at get cart, how we defined it in the past. In the past we simply reached out to our cart items, we got all the product IDs and then we had a look at the products collection to find all the products in there, convert this to array and return the products, so essentially get cart gave us an array of all the products in the cart. Now with mongoose, this is a bit easier, we already have our nested cart items array and if we have a look at compass, we see the items are objects where we have the product ID and now we simply need to populate the product ID with all the data we are interested in. So in shop.js in get cart, I will not call get cart on my user instead we can call populate on the user to fetch data even though we already fetch the user, we can also call populate on that and tell mongoose to fetch data for, now which path is it? Well for cart.items.productid, let me console log my products here to see if that works. Now we also need to add that route again, so in the shop.js file in the routes folder, let's add that get cart route again and with that changed, let me load the cart page and I get an error and the reason for that is populate does exist but actually it does not return a promise, so calling then on it would not work, we have to chain exec populate after that and then we'll get a promise, so then we should be able to call then. So now with that, let's log products, save that and reload that page, now I get no products in cart but if I go back, well we see actually what we have here is the full user object which makes sense because we're not just fetching products, we still work with the full user, so that is a tiny change so let's actually use or log the user cart items here to see if that was still populated with all the item data and I'll temporarily create an empty products array so the rest of the code does not fail and now let's save that and reload the cart page one more time. If I now go back, you see indeed here what I log, user cart items now is an array of items where the product ID is populated with the product data. So now it works a bit different than before but it still gives us the data we need, so here products now actually is user cart items and that is what I pass to my view but I will need to look into my view now because the structure changed a bit compared to before. The cart.ejs file is what we need to edit and in there, we loop through all products which is fine but remember that our product data will then be nested in a product ID field and you could also rename this to just product in your schema therefore which makes a bit more sense I guess but I still have product ID here, so the title is not available on the top level object which would be this object but on the nested product ID object. So here we have to say p.productid.title. The quantity is on the top level object so this is fine, the product ID again can be found on the product ID nested or embedded document though. So with that, let's save that and let's now reload the cart page and indeed we see our cart item here now. So this is looking better, now we're able to populate our cart with data.

**Lecture 222**

**Deleting Cart Items**

Now that we're able to load the cart data, let's of course make sure we can also delete cart items and for that, we have the post cart delete product action in our shop controller. Now in there, I use the delete item from cart method on the user object which we wrote on our own in the last module and which of course is also not a default mongoose method. So it's time to go back into the user model and have a look at the delete item from cart method and see how we can recreate that functionality. There what I do is essentially I create an updated cart items array by filtering out the items that should survive, so all the items except for the one I want to remove and then I update my model. Well we can of course do that by copying, well basically just that code here and adding another method just as we did it with Add to Cart, so I will use my user schema and there, the methods key again and then I'll add a remove from cart method which will be a function and in that function, I will add that code which I copied from the bottom. This code relies on being aware of the product ID of the product we want to delete, so Product ID is an argument I expect in this function here, in this method and now we get the updated cart items and now all we need to do is we need to set this cart items equal to the updated cart items and then we return this save and now we got a method that we should be able to call to remove an item from the cart, just like that. So let's move over to the shop.js file in the controller's folder and there in the post cart delete product route, well I just renamed it a bit, I named it remove from cart, you could have kept the original name of course. Now I have remove from cart, I pass the product ID and I redirect here to the cart page once we're done. That should work but it will only work if we enable the fitting route again, so let's go to the routes folder, the shop.js file and let's comment in the cart delete item route again. And with that all changed, if we now go back and we click delete here, that looks good, we loaded the cart page and all the data is gone. Let's also verify this in compass by reloading our users collection, there in the cart items is an empty array. So we're now able to add items to the cart and remove items from the cart, let's next work on the orders.

**Lecture 223**

**Creating and Getting Orders**

Time to work on the last two pieces, the orders and again feel free to pause the video and try implementing this on your own, it's always a great practice even if you can't get the full result or full equivalence to my solution or if you get stuck at one point, you can still then go back and watch the solution together with me but you will still have a lot of practice by trying parts of that on your own. So here's your chance to pause the video and we'll do it together thereafter. So let's dive into that together and let's work on our orders now, creating orders and getting the orders, that is what we want to do and let's first of all look into our controller for that, the shop controller. There we have post order, now what are we doing there? Well here first of all we got an unused variable which we can get rid of, there we call add order on the user and that's essentially it, so let's look into the user model and let's see what add order did. Add order simply took our cart, so the products in our carts to be precise and created an order object which contains the product data but also some data about the user and then we inserted this into an order collection. Now since we worked with a collection there, we'll need a new model that's for sure and the then is a result we just cleared the cart. So we get multiple steps we need to do here, first I'll add an order.js file in the models folder because I'll need an order model to store data in an orders collection. First of all let's import mongoose here by requiring mongoose like that, let's create the schema constant where we access mongoose schema like this and let's then define our order schema and how that should look like and of course here you can really also go with the schema that works for you or that you want to use. Now I will try to recreate the schema we used before where an order had the items and the user data and that was it and the items were our products which in the past was all the product data and the quantity. So here in my schema, I will have my products let's say, I named it items in the past, here I'll name it products. Products will be an array of documents and every document will let's say have the product data which will be of type object because this will be a full other document you could say that is required and besides the product data, I'll have to quantity for that product, could also name this just product here by the way if you wanted to. So here the quantity and that will be a type of number and this should also be required. Ok so this is how my products will look like, now I also will have a user and there I want to have a name let's say where the type is a string and this is required and what else did we store about the user in the past, well just the ID so also have my user ID here which is of type schema types objectid and which is also required and this will also refer to the user model. Now type object by the way is a bit of a shortcut, of course we could define the full nested product with all the properties there, feel free to do that, I'll just say well this is any object. So this is my order schema, now I'll export a model based on that schema with mongoose model, name that order and hence the collection will be named orders and then I got my order schema there. Ok so now I got my order model and that is new, we didn't use that before in the last module, we could have but now we well we implemented it differently there. Now since we use a model based package again for interacting with the data, we have to be more strict about that and that is a good thing generally and therefore we have an order model again. Now with the order model added the question is how do we interact between the order model and the user now? We had add order in the past and add order essentially just grabbed user data, products, created an order and stored it, now I still want to do that but I'll do it in a controller now. In post order, I'll work a bit differently, I will first of all import my order model at the top from models order like this, so this import was added and then down there in post order, I will create a new order object by using that order model constructor. Now that order object needs to be initialized and it needs to be initialized in the way we defined it here, so it needs to get products and it needs to get some user data. Now the user data is a bit simpler, we got our request user, so here we have the name which is request user name because remember, request user is a full user object fetched from the database so there will be a name property and we also want to store the ID of that user in the order, we also have like the user ID, so we should store that as well, so we'll add user ID here and that will simply be, well we can just use the entire request user object and as I mentioned mongoose will pick the ID from there. Ok so this is the easier part, now we also need to add the products for these users cart. However it's not that difficult because we already implemented something similar. For getting the cart, we use this approach for fetching all the products that are in the users cart. So in the end we can just copy this here, all the way up to the part where I extract the products and reuse that in post order at the beginning here, just make sure to close that then function here and here we now got the products that are in the users cart. Now important is just products is not entirely correct, you should remember that here we have an array of elements were we have the quantity and then the real product data was nested in a product ID field but we can work with that of course. We can now create our order inside of this then block and here, we can now add products which should be an array of products and that should be an array of products which is a bit transformed, so not the structure of products we get back here but after some transformations, to be precise products for us need to have a product field with the product data and a quantity field, that is almost what we have but we have the product data in a product ID field here, not in a product field but that is easy to fix. I will simply not just extract items like this, I will also map the items so that I store the changed items in my products array and a mapped item should still have its quantity so I'll keep that, the i simply refers to the item since this function goes through all items in that array, so we'll have the quantity. And then we'll have a product field and the product field should have all the product data so that we'll store everything that I had in i product ID before because that was the old structure we had in there, now we have this structure, we have an array of products which just have a quantity and then the product detail data which is exactly the structure we expect to get in the order model and now we can simply store these products in the order. And with that we got everything we need almost, now we just need to call save on that order, so order save now saves that order to the database. And then we can chain then on that once we return order save so that we can also redirect and so on. Let's see if that works by going to our shop routes again and commenting the create order route back in and now back in our project, let me first of all add a second product so that we get something to play around with and see that this does not get added, this is just a dummy product obviously. So now in our cart we got no products, let's add a nice book to our cart, maybe also from the details page and let's not add the other product and let's click order now. Now we don't find the page we want to redirect to, that's fine and the cart also was not cleared because we haven't added that logic yet but let's have a look into compass and let's see, if we refresh the entire setup, we got the orders collection and in there I got an order with some user data, that looks pretty good, let's quickly confirm the user ID, it ends with 8A8, that looks good and let's have a look at the products there too and there I got a quantity of two and I got my product ID. Well the product ID, is that what I wanted though? I wanted all the product data right, let's fix that in the next lecture.

**Lecture 224**

**Storing All Order related data**

Now let's see what was causing that issue and for that let's console log user cart items to see what's in there when we place an order. So if I click order now now and I increase the console, we see product ID indeed does hold a full object and not just the ID which is what gets stored though, if I refresh my orders, now I got two but both orders actually just have the product ID in there, not the full product, I want to have the full product data though. Now one thing we can do is here when I store the product ID, I can wrap that in curly braces to create a new javascript object, use the spread operator and use that not directly on the product ID but on a special field mongoose gives me, \_doc. I can access this here because product ID actually will be an object with a lot of metadata attached to it even though we can't directly see that when console logging it but with .doc we get really access to just the data that's in there and then with the spread operator inside of a new object, we pull out all the data in that document we retrieved and store it in a new object which we save here as a product. And with that if we save that and we go back to the cart and I click order now, I get no error and if I go back to my compass interface and I have a look at this new order, there I see indeed I got all the product detail data in there too. So this is now working and this now allows me to store all the data I want to store with every order. Now with that being stored, of course the missing part is that we also clear the cart.

**Lecture 225**

**Clearing the cart after storing an order**

So let's clear the cart after storing an order and for that, here once I'm done with saving the order, I want to clear the cart and one way to do that would be to go to the user model and add a new method there. We can go to the user schema, to the methods key and then add a clear cart method, so simply a function we can call to clear the entire cart and how would we do that? Well we would simply set this cart equal to an object where we have an empty array of items and then we just return this, whoops, return this save and that should be it. That updates the carts to have no items in it anymore and we then save this and therefore we update this. With this, let's go back to the shop controller and here once we know that the order was placed, let's simply call request user clear cart, this method we just added and then I will return that and move my redirection into a new then block after the previous one which will only execute once the cart was cleared. So now with that, let's save that, let's now also go to products and maybe add another product to our cart. Let's order that cart now, let's click on cart again, no products in cart, this is looking great. Let's quickly check our orders in compass and this new order should now have two products, that is looking good, one with quantity two, that was our nice book and one with quantity one, that was the second product and in the users, the cart is empty. So now we got an elegant way of managing orders and the cart through mongoose. However let's of course also add the functionality to get the cart in the next lecture.

**Lecture 226**

**Getting and displaying the orders**

Let's move towards the end of the module by making sure that we can also get the orders and display them on our orders page here. So for that, we want to work in the shop controller with get orders and there I had a get orders function on my user model we could call. If we have a quick look at this function, get orders simply reached out to the orders collection and found me all orders for this user. Well that shouldn't be too hard right because what we can do of course is in our controller here, we know the user ID because we got it here in a request object, so in the end what I can do is I can use my order model which we're importing at the top of the file and I can find all orders where and let's have a look at the order model real quick, where the user, user ID so this nested object here, this nested key is equal to the user id of the logged in user. So here we have user.userID, that should be equal to request user \_id, that is the check I want to make and this will give me all orders that belong to that user. Well and then we can just use these orders here in the then method, so essentially I can also just reuse my old then method where I already expected the orders and I can render the orders page. If we now save that, we just need to make sure the route exists, so let's quickly comment that in here and now if I click on orders, I get an error because the structure of our data changed a little bit, so we need to work on the view to adjust it to our data in my orders page here. How did it change? We loop through the orders and each order has an ID, that will still work but then we have no order items anymore, we've got order products now, we can confirm this with our schema or in compass of course, we got order products and each product has quantity and then the product data in a product field. So inside of here, we have the product and there we have a nested product field, so we could also name this just P to avoid confusion, this will be p and there we have the product field with the title but directly on the top level p object, so directly in the object that is stored in that products array, we have the quantity so we can still access this directly on p which is the part directly in order products but then the product data itself is nested in one additional embedded document product. But with these changes to our view, if I now reload that view page, this looks good, now I see all my orders here as we did before. Now something is wrong about the first two orders and the thing that is wrong is that simply these orders were created at a time where we only store the object ID and no special object data, so let me quickly delete these two orders here so that we don't have them in our view and in our app anymore, so now we only have valid orders in here. So this does now all work, feel free to play around with that, try adding totals for the orders or anything like that. For now this is the state I want to work with, I want to continue with and this is what I wanted to show you with mongoose and mongodb and you therefore now saw a lot of different alternatives for working with databases and how to work with the data then.