**Lecture 176**

**What is MongoDB?**

So what is mongodb? Now let me first of all tell you that I do actually have a complete course for developers on mongodb, so you might want to check that out if you want to learn way more about mongodb than I'll be able to cover in this course but now with that let me start with a brief introduction to mongodb at least, what is it? Mongodb is both the name of the company which developed mongodb but also then of their most famous product, a database solution, a database engine you could say, a tool you can use to run very efficient NoSQL databases. The name stems from the word humongous because mongodb was built for one major purpose, that you could store and work and that really is important, the work part, that you could store and work with lots and lots of data. So mongodb is built for large scale applications, mongodb is built to quickly query data, store data, interact with data, so it's really fast and it's really awesome database philosophy that is behind NoSQL databases and therefore also behind mongodb. Now how does it work? Well just like in the SQL world, we spin up a mongodb server and there we can have multiple databases, for example a shop database. Now in such a database in the SQL world, we would have multiple tables, in the NoSQL mongodb world we have multiple collections like the users and orders collection for example. Now inside of each collection, we don't have so-called records but we have a couple of documents. Now documents also look different than records did and it's not just about different names being used, the core philosophy behind the database really is a totally different one. For example mongodb is schemaless, inside of one collection, your documents which is your data, your entry so to say don't have to have the same structure. In SQL that was totally different, we had a users table there and in that users table, we had an ID, a name, an email, a password. Now here that is different, here we can have any kind of data in one and the same collection. Often you will still end up with an at least similar structure but you're not forced to have exactly the same structure and this gives you more flexibility, also for your application to grow and to change its data requirements over time without that being difficult to depict in your database world. So this is one thing, a document in mongodb looks like this and this looks a lot like javascript object notation and to be precise it kind of is. Mongodb uses json to store data in collections, so every document you store looks something like this, it follows the javascript object notation format. To be very precise mongodb uses something which is called bson for binary json but that only means that mongodb kind of transforms this behind the scenes before storing it in the files but you don't have to worry about that, you will basically use it as json. Now such a json document is basically the same as a javascript object you could say and there as you see, we can have nested or as mongodb calls them, embedded documents, for example the address here would hold an embedded document. And you can also have arrays inside of that document and that array can like in this case hold other documents, other objects or it could also just hold strings, numbers, anything of that kind. So again for the data, you have great flexibility and the existence of these nested documents also means that relations are managed a bit differently in the NoSQL mongodb world. So let me come back to that in the next lecture.

**Lecture 177**

**Relations in NoSQL**

So we know how we store data and I mentioned that this gives you more flexibility, also regarding the storage of relations between different data. Now in NoSQL it would be pretty normal to have something like this, here are three collections and we have some duplicate data in there, we have a users collection which holds all the details about a user but then we might have some copy of that data or of a part of that data in an embedded or nested document in another document in another collection. So instead of just matching by ID as you do it in the SQL world, here you can also depict a relation by embedding data into other documents. You could embedded the ID which points at another document so that you still have to merge two documents manually and you will indeed have to do that pretty manually but you can also just take the information that is important for you in the context of another document. Let's say here, some user data for the orders and you copy that into the orders and then you have that data right there whenever you fetch all orders without you having to fetch all orders, then look for the fitting users and fetch them too and this is part of what makes NoSQL and especially mongodb so fast and efficient. It really is built to make sure that you query your data in the format you need it, that you store the data in the format you need it, that you don't have to do a lot of merging and so on but that you can really fetch data in the format you need it without having to combine multiple collections behind the scenes on the server. That being said, you can still do that, nested and embedded documents are one alternative for depicting relations, references are another one. So here's the embedded document example where the address is part of our customer document, so instead of having two collections, customers and addresses and then matching by ID, here we put the address right into the customer. There also are cases where you would have a lot of data duplication and where you need to work with that data a lot and hence it would change a lot and you would have to manually update it in all duplicate places, where using embedded documents is not ideal. So for example if you have some favorite books for every customer, well you would have lots of data duplication because a lot of customers might have the same favorite books and these books might then change a lot, maybe there is a new edition published and you would have to go to all customers who have these books as favorites and update the entries for each customer. In such a scenario it would be easier to still go with two collections and only store the references to the books in a customers documents and then manually merge that with the books which are managed in a different collection. And over time you'll get a feeling for which approach you want to follow, I'll show you some examples in this course and have a look at this complete mongodb course I did mention which drives much deeper into that and shows you way more examples that can be helpful to you. So this is the idea here though, we can embed or we can use references, whatever fits the purpose a bit better and with that, we know how mongodb generally works, that we have the important thing of not having a schema hence we have more flexibility, no structure is required and we have fewer data relations because we can relate by embedding. We can still build relations manually with references as you saw but you should always know if that is the best approach right now or if you can use an embedded document with too much work. So these are NoSQL characteristics and therefore the mongodb characteristics and these are also part of the reason why mongodb is so popular because of the speed and the flexibility it gives you.

**Lecture 178**

**Setting up MongoDB**

Now that you had a brief introduction to what mongodb is and what NoSQL databases are, let's install mongodb, . let's set it up and let's use it for our project here. Now to install mongodb, you can either do that locally on mongodb.com, you can click on get mongodb, download the installer or the tools when you click on community server here and follow the installation instructions and this will all work and it will give you a locally running mongodb server. Now I would go for a cloud solution here because it's the more realistic environment we would use for deployment anyways and it's really easy to set up and it's free and that will be Atlas, so mongodb Atlas. So I got here by clicking on get mongodb and then choosing atlas. Now again this is free, you don't need a credit card or anything like that, you just need to sign up and you can spin up your own in the cloud best practice managed mongodb server which is a great environment for practicing because this will follow all the best practices. So sign up here, now once you did sign up you should end up on a page that looks something like this. Maybe you need to create a new project first, give it any name you want and then you should be on that page or you can build a new cluster, you might also end up in that wizard right from the start. Now here you configure a mongodb cloud cluster, so basically a mongodb server and therefore a mongodb database or multiple databases, as many as you want running in the server. You can generally leave all the default settings, make sure that you choose a region where a free tier is available, behind the scenes mongodb uses one of these cloud providers but you don't need to be signed up with them or anything like that, so you can choose anything you want here, then you choose the cluster tier and there you should use the free one and zero where it says free and then this will not cost you anything. Under additional settings, you can well leave all the defaults, you can't change them anyways in the free tier but that's fine. You can change the cluster name if you want and then you can click create cluster and this will now create a new cluster, we'll take a couple of minutes which you then can connect to later and which you then can use as your backend. Now one important note, you might see that I have some cost here in this account and this is not coming from the setup I'm showing you here, I was simply using this for some other set up in the past which did cost something, this does not cost you anything. Now whilst this is getting set up, you can already click on security here and there you will probably not have as many users as I do. Now make sure you add at least one new user and I already did this which and that's important, which has read and write access to any database. You can turn it into an atlas admin but more realistic is this here because this will later be the role which our nodejs application assumes which should be able to read and write our databases but not to administrate them because we'll not do database administration through nodejs, that would be something our database admin does and therefore not our app. So make sure you create such a user, you can also sign your own password or auto-generate a secure one, make sure you do save that because you'll need it later and then also make sure you have a look at IP whitelist. There you see all the IP addresses that are allowed to connect to your mongodb server, now I already got a bunch here that will probably be less for you. Now one thing you should do here is you should add a new IP address and add your current IP address since the node app runs locally on your machine, your node app will have this IP address. Later when you deploy the node app this should of course use the IP address of your server where you deployed it to but here you can use your local one so that you can connect and your node app can connect to that server. That's of course a good security feature because this makes sure that no unauthorised access can happen to your database, so your database is now locked down both from a IP perspective but also from a user perspective. And now we can wait for it to set up to finish and I'll be back once it did finish. Now the set up did complete and now we can connect to our mongodb server from inside our node app and for this we can click on connect here and choose a connection method which in our case will be an application. Now here we can check I'm using this driver and we get this url which we'll need soon but first of all we need to install the mongodb driver, in our case for nodejs. You can click on an example down there but I will simply show it to you in our project.

**Lecture 179**

**Installing the MongoDB Driver**

So with the mongodb server up and running, let's add the mongodb driver which simply is a package we can use to connect to mongodb. For SQL or MySQL, we used MySQL too and also sequelize. Now in my case here or with mongodb, we'd run npm install --save and then it's just mongodb and this installs the official mongodb driver which we can use to connect to mongodb. Now with that installed, we can start using it and we can start using it in the first file that gets executed when we bring up our server which would be the app.js file of course. There we basically already do some database set up with sequelize and we can get rid of all of that. We did create a default user here but we did this in the MySQL world, so we'll start from scratch basically, we can get rid of all the associations and so on we set up here so let's get rid of that. Therefore we don't need these imports anymore, let's get rid of all of them including sequelize from the util database folder and with that all removed, also here where I used my user model to find the user by ID, I'll at least comment this out for now so that we really have a lean app.js file and now I'll go to my util folder and there, the database folder and I still want to connect to a database here but that will no longer be sequelize of course. So let me clear all of that here too and let's now set up some code that will connect us to mongodb. Now for that, I'll first of all import mongodb here, so I'll set mongodb equal to requiring mongodb, this gives us access to the mongodb package. In there, we can extract a Mongo Client constructor by simply accessing mongodb, the thing we're importing and then Mongo Client, with a capital M and C, the constant here can be named whatever you want of course. So now with that, we can use that client to connect to our mongodb database and we do connect by running Mongodb Client and Mongo Client connect, like this, so this is all we need to do to create a connection to mongodb. Now connect here simply takes a url to connect and that url is exactly that url you're have in the connect modal here on the mongodb Atlas cluster page, so you can copy that and paste it into here. Now one important thing, you need to make sure you are using the right user and in my case that should be Maximillian, the user you created in your mongodb cluster under security users and the fitting password for this user, so I'll quickly insert my password here too. With that we got all set update that we need to establish such a connection. Now the connect method here actually returns a promise, which either fails or throws an error if the connection fails and in such a case here, I of course want to output my error here so that we can diagnose it or we have a successful connection, so here I can say console log connected. And with that, we have a file which when we execute, it would connect to mongodb. Now of course you want to execute this together with app.js, so here we are we bring up our server or actually we're not doing this right now but where we will soon do this again, here I also want to connect to my database. So to do that, how can we achieve this? Well what we can do here is we can wrap our connection code here into a method and I'll simply create a method by using const here and I'll name it mongo connect, this will be an arrow function and inside of that arrow function here, I will then execute my mongo client connect code like this. I also expect to get a callback here and I will call that callback and pass result into it here inside of my then block once we did connect here because the result I can already tell you this will be the client, so basically a client object which gives us access to the database. And then here all I need to do is I need to export mongo connect, this function here. Now with that exported, we can go back to app.js and import that here, so here I'll also name that mongo connect but that name is totally up to you and I'll import from the util folder and there, the database file. There I will import my mongo connect constant which will be a function because I'm exporting a function here. So in app.js at the bottom, I will execute mongo connect and remember here, we have to pass a callback so a function that will get executed once we connect it and here I will get access to the client object. So in here, I will run app listen and bring up my node server because I want to do that once I know that I connect it to the database and I will also console log my client here so that we can have a look into that. With all of that, let's run npm start and bring up our server and the problem I now have is simply that when I bring up my server, we do actually register all our routes here and in the routes files, we dive into our controllers and in our controllers, we are using that sequelize object which simply does not exist anymore because we're importing the models here and in our models, we require sequelize and well that simply does all not work as before anymore. So in order to make this work, for now to simply see how we connect, I will comment out my routes here and therefore also comment them out down there. So now we got no working routes for the moment but this means that we can now at least connect and indeed with the automatic restart, this does not look like an error instead we see connected here and then we see this Mongo Client object which we got with some details about the connection and this is in the end the object which we'll be able to interact with, to work with, to well create data in our database for example. So now we are connected but of course our setup is broken now because it still relies on sequelize which we're not using anymore, we're not even using SQL anymore. So let's now implement mongodb in our app step by step and whilst we do this, you will learn how to insert data, find data and so on.

**Lecture 180**

**Creating the Database connection**

We connected to mongodb and this is nice but it also means that we can't do anything else right now, the rest of the app is not working anymore and of course it should work. So let's get there step by step and let's start by re-adding the admin routes, so let's add this middleware again and for this, I also need to import admin routes again and if we now have a look into our admin controller, here we do of course use the product model, in the product model if we have a look into that does rely on sequelize of course because we define the product model with the help of sequelize. Now that is all something which we won't do anymore instead we now want to use mongodb. So to get there, I will remove these two imports up there and instead in here, I want to create a new class again because I'll create my own model again and this is something we did before already in the pure MySQL module. So I'll create a product class here, like this and of course I will also export this, so module exports will still be my class here. Now in that class, I'll have a constructor and in that constructor here, I want to store the title, price, image url and description of the product when it gets created. So here I will simply get the title, the price, the description and the image url and then in the constructor I'll say this title equals title, this price equals price, this description equals description, well and so on, you know the game, so here image url equals image url. Now we can create a new product in javascript, a new object which follows this form and now to save it in the database, I will also add a save method here, so a function which can be executed on this class and in here, I now want to connect to mongodb and save my product. Now to do that, to be able to connect, I'll need to import mongodb or mongo connect, so I'll import mongo connect from my utility folder and there from the database file. So I will simply import that method, the function I created here where you pass a callback to, where we do connect to mongodb inside and then we basically execute the callback and return the connected client so that we can interact with it. However if we would do this, we would have to connect to mongodb for every operation we do and we would not even disconnect thereafter, so this is not really a good way of connecting to mongodb since we will want to connect and interact with it from different places in our app. So it would be better if we could manage one connection in our database and then simply return access to the client which we set up once from there or to the different place in our app that need access. And to do that, I'll tune my set up here a bit. Let's tune it together in the next lecture.

**Lecture 181**

**Finishing the Database connection**

So I still want to have a function which I can call to connect and therefore, this function still is looking good to me in general. One thing I want to do is I'll add a throw error here so that I also throw the error again when we fail here but besides that, I'm happy with having mongo connect here but I'll change something, I'll not return the client in my callback here instead I'll add a variable, \_db, the underscore is only here to signal that this will only be used internally in this file but you don't have to name it like this, you could also have normally, just like db and initially this will be undefined. Now here in the then block, I will store a value in there, I will store access to the database here and if I leave it like this, what we will do is we will connect to the test database by default because that is what we specify here in our connection string. So I will connect to shop here and then this will give us access to the shop database to which we automatically connect and you could also enter the database name here to overwrite this and connect to a different database than you were connected to initially but I'll not enter anything and just connect to that database. As a side note, unlike in SQL we never need to create that database or the tables, the collections in there ahead of time. It will be created on the fly when we first access it which is again fitting that flexibility theme mongodb gives us. In SQL we had to prepare everything in advance, at least when not using sequelize which also had to do that but it did it for us, here we don't need to do anything, we're just telling mongodb hey connect me to the shop database and if that database doesn't exist yet, mongodb will create it as soon as we start writing data to it. So that's just a little side note, here I do store a connection to my database in the db variable. And with that stored, now in my callback I don't need to return it but I will add another function here, I'll name that get db and here I will simply check if db is set, so if it's not undefined and in this case I'll return db, so I will return access to my database otherwise I'll essentially not do anything here, so I'll return undefined, we could also throw an error, no database found, something like this. Now here, I also want to export that so I'll not just export Mongo connect but instead, I'll just use a different syntax you learned about, exports mongo connect equals Mongo connect and I'll also have exports get db which equals get db, like this. So now I'm exporting two methods, one for connecting and then storing the connection to the database and therefore, this will keep on running and I have one method where I return access to that connected database if it exists and mongodb behind the scenes will even manage this very elegantly with something called connection pooling where mongodb will make sure it provides sufficient connections for multiple simultaneous interactions with the database, so this is really a good pattern we should follow. Now with that changed, we still can connect, one thing I need to adjust though is in the app.js file where I do connect, I'll not get declined anymore because we don't return it in the callback anymore, so now we just know we are connected but there is nothing else we can do. But one thing I will be able to do is in product.js where I create my product model, there I don't just can connect the connection now but I can import get db and that of course is very useful because that means I do import this by accessing get db here because this means that I now can call this function to get access to my database and therefore I can use it to well interact with the database. Now let's use that database connection starting in the next lecture.

**Lecture 182**

**Using the Database connection**

So now we're importing something which allows us to get access to the database connection we set up initially when starting our server which now is a concept that we can reuse. So in the save method of the product model, I can now get access to my database by calling get db, remember get db does simply return that database instance we connected to, so now we have a connection which allows us to interact with the database and then here in mongodb, you can call collection to tell mongodb into which collection you want to insert something or with which collection you want to work because remember in mongodb you have databases, collections and documents. We have a database connection here, so the next level is a collection. So here we can connect to any collection and just as with the database, if it doesn't exist yet, it will be created the first time you insert data. So here I'll connect to a products collection. Now on that collection, we can execute a couple of mongodb commands or operations. Now a full list can be found in the official docs and of course you'll learn all about them and all the details in my full mongodb course which I mentioned. In the official docs, you can click on mongodb server and here for example click on mongodb crud operations to learn how to insert find, that is what query is, update and deletes documents and you'll see all the examples here and all the commands and how to configure them and so on, so this is a great place for you to dive deeper besides the other course I mentioned but of course here's a short introduction too. If you want to insert data, you can do this with insert one or if it's multiple documents at once, insert many. Insert many then takes an array of javascript objects you want to insert, insert one which I'll do here because I want to insert one product only takes the object you want to insert. Now what do you mean with object? Well you would simply pass an object where you have like for example name, a book, price 12.99 and so on, so this is something which would be valid code, that would insert such a document into the database. By the way this is not json, this is a javascript object but it will be converted by mongodb . However here in our case, essentially it's the product which we want to insert right, so we could just say this and see how that works if we try to insert this into our database. Now this is all we need to do right now, insert one then returns us a promise, so we have then and catch and we can log any error we found or we can have a look at the result we get back, so we can console log the result. With that out of the way, let me remove the sequelize model and export our class with the save method and now with that, we fixed the model file, let's now go to the admin.js file. And here obviously we got a couple of methods or a couple of functions which still use parts of the model that won't work anymore, like this one here, request user create product for adding a new product. This will not work anymore, for now we'll have to create a product without storing something about the user, we'll reintroduce this later again. So for now, we can overwrite this here though and I also want to comment out my other methods because these will not work anymore, so let's comment them out. We'll comment them back in later once it works and therefore in the admin routes, here routes admin.js, there I also want to disable all routes that will not work anymore. So essentially that's all but this post route. So here for getting all products, this will not work anymore and all these routes down there will also not work anymore but these two routes should still work for add product. So now with that, I need to fix one more thing in the app.js file, here I need to import .mongo connect here from my database utility file. And with that now we're not only connected but we should also be able to insert a new product already, so let's try this out. Visit localhost 3000/addproduct and this won't load because in app.js when I commented out the code here in app use, well I should at least call next otherwise every incoming request will die here. So with that changed, now if we reload, we are back to our admin add product page. Now let's simply check this and see if it works by adding a product. Now I get an error here regarding create product and that of course makes sense because in the admin.js file, I didn't change any code. So let's work on that code next, so let's make sure that we can add a product.

**Lecture 183**

**Creating Products**

So let's make add product work again and for this we'll not use the user anymore but instead here before we use then, we'll re-use that so no need to delete it, before we do that we'll use our product model. So I'll create a new product constant by initialising product again and here, I will simply pass title, price, description and the image url, so we'll pass all that data into the constructor of the product. Then on this product, I can call save and then here to call then, I'll go into my model again and I will return this here, so I'll return my collection and then this entire command chain because that will allow me to well treat this overall as a promise and chain then thereafter and then therefore also redirect. So now with that if I save that and I go back to the form and resubmit that, I'm actually redirected to a page which is not found because I commented out all the shop pages but if we go back to our server side console, we see something interesting. We see that this here has to be the output of this console log line in the product model where I print the result of the insert operation and there, we see a lot of data about that operation and for us important, if we scroll down to the bottom, we see one document was inserted. It received an ID and such an ID is managed automatically by mongodb because every document needs to have such a \_id, this is a must have and mongodb creates it on the fly automatically if the object you entered does not have it, so we'll use that auto-generated ID and then you see the data which we entered also was stored. So whilst we can't look into database yet because we're not fetching anything, we see that our insert one operation was successful and did successfully add a product into the collection which is of course amazing.

**Lecture 184**

**Understanding the MongoDB Compass**

Now in the last lecture, we did insert a product into our mongodb database which is awesome. Now I also want to see it and before we actually fetch it in our node application, let me show you another tool called compass. If you click on get mongodb on mongodb.com, you can choose that and you can download and install it for free as well. It's available for Windows, MacOS and Linux and once you get that installed, you can start your compass application on your machine and this essentially is a tool that gives you a graphical user interface to connect to your database and to interact with it. So let's wait for it starting up here and once it did start up, you can connect to it. Now to connect to it, let's go back to our mongodb cluster and click connect here and then click connect with mongodb Atlas, choose your operating system and then copy that url down there. Now one cool thing is if you now quickly close compass again and you restart it after you copied that url, it should tell you that it detected a connection string and if you click yes, it will insert the most important pieces here for you. Now you still need to choose how you want to connect, so make sure your username is correct and also enter the password for this user, you need to do this manually. All the defaults can be left as they are and you should be able to now connect to your database and here it is and interesting enough, you are to be precise, you are now connected to the database server and here you see a couple of databases, two default ones which you don't need to touch and which you shouldn't touch but then also your own one, shop. And the shop database here actually has a products collection as you can see and if we look into that products collection, in there we can see the documents that are stored in there and here is that one document we inserted, so it is one product we added in the last lecture. Now you can also insert documents and edit them inside of compass and check out the official compass docs to learn all about the features you can use here. I just want to use this as a nice visual support so that we can see our data before we fetch it in our node app. That being said obviously the data is not that useful to us here, so let's go back to nodejs and write some code so that we can use the product data on our shop pages again which we commented out at the moment.

**Lecture 185**

**Fetching All Products**

So let's work on our shop pages now and let's make sure we can use our data there. For this, let me go to my product model again and here, besides being able to save data, I of course also want to be able to get my products, so I'll add a static method again and I'll name that fetch all. Now here I want to interact with my mongodb database to fetch all products. For this, I will again return and then use db collection to tell mongodb to which collection to connect to and here the collection I want to connect to is products of course and then there, mongodb has a method for finding data which is called find. Now find could be configured to also use a filter, for example we could find all products that have a title of a book and there are more elaborate filters than just equality filters available too, again something I cover in great detail in my mongodb course if you want to learn more about that. Here I of course want to find all products for now. So I want to find all products which I can do by just calling find like this. Now the important thing about find is find does not immediately return a promise though, instead it returns a so-called cursor. A cursor is an object provided by mongodb which allows us to go through our elements, our documents step by step because theoretically in a collection, find could of course return millions of documents and you don't want to transfer them over the wire all at once. So instead find gives you a handle which you can use to tell mongodb ok give me the next document, ok give me the next document and so on. That being said, there is a toArray method you can execute to tell mongodb to get all documents and turn them into a javascript array but you should only use that if you know that we're talking about let's say a couple of dozens or maybe one hundred documents otherwise it's better to implement pagination which is something we will implement at a later point of time in this course. So for now, let's use toArray and then this returns a promise and here we can again catch any errors we might be getting but most importantly here, we should have our products and we can log our products here and then also let's return our products in here and let's see if that works the way we want. So now we have a fetch all method that hopefully works. One important thing is I of course need to get access to my database by calling get db in fetch all, just as I do it in save and now let's head over to our shop controller and there, we have the get products function. Now this is the function I want to work on now and instead of find all, let's now use fetch all because that is how I name the method and there I should get my products and hopefully I get them in a format that works, let's see. Let me also go to my shop routes now and here, get index and get products, that should be fine, let me comment out all other routes for the moment because we can't work with them right now because we have no code that would support them. By the way in the shop controller, you also want to make sure that get index works by also using fetch all in there, so just as we did it in get products, we want to do the same in get index. Ok so now I got some code in place, let me go back to app.js finally and comment in that shop routes import and not just comment it in of course but also uncomment this middleware down there. And now we hopefully have a set up where we can actually reload our root route let's say, so just localhost 3000 and here we go, here is our product just like that and of course it works automatically because I didn't change any property names. If you suddenly store the title in a field named name, you would have to adjust your view and so on. Well and there is something where I want to adjust my view and that is related to how we work with the IDs but we'll see it in the next lecture.

**Lecture 186**

**Fetching a single product**

So we're able to fetch all the products, let's now implement the code to fetch a single product and for that, I'm back in the shop.js controller file and here it is the get product function I want to work on. Here we get the product ID as part of the url and then we want to use our product model to somehow find the fitting product and therefore we of course need to work on the model again. And feel free to implement this on your own if you got some mongodb knowledge, it's a great challenge then otherwise of course we'll just do this together. So how would I fetch a single product? Well besides having my static fetch all method, I'll add another static method, find by ID and you can name this however you want of course. Here I expect to get a product ID as an argument and then just as before I'll call get db to get access to that database connection we have and I then want to return the result of an operation where I use my collection and there, I will now pass products again because it's still the same collection I want to interact with and I will find a product here but I'll find only one product. And to do that, I'll narrow down the result set with find and then I'll pass a javascript object to it which allows me to configure a filter and here, I want to look for a product where \_id is equal to prod ID because that's the ID of the product I'm looking for. So with this, I'm returning theoretically all products which have this ID but I know it'll only be one so I can use find like this and I'll only get back one product or do I? Well actually find will still give me a cursor because mongodb doesn't know that I will only get one and here we can use next, the next function to get the next and in this case here also the last document that was returned by find here. So here I can then add then and catch and as always, log any error I might have and then in then where I will have my one product, there I will log it to the console for one and then I will return my product. So now this should hopefully yield my product here. Now with that, if we go back to the shop controller with find by ID, I either have an error or I get my product and I try to render the product detail page. Now let's go to the routes and there to the shop.js file and we need to comment in that route here for getting the product details. If we now save everything, let's click on details here and one thing you can see is that this doesn't seem to work right, we always get redirected to /products page. So if I go back to the starting page and we can tell the difference by the distance between the dollar sign and the character, on the starting page if I click details I just get forwarded to products, now why is that? Now this actually happens because no product ID is passed when I click details and therefore we end up in this route. Now why is no product ID passed? If we have a look at our view and there add our index.ejs file in the shop folder, in there we got our details button and I do actually add the product ID here but what is the error here? Well I access product.id here but with mongodb, it's \_id, so I should use product\_id and the same of course on the product list page, there I should also use product \_id. So in all the places of the app where we used product.id, we should now use product\_id. And with that if we reload that page here, the starting page or products page doesn't matter, if I now click details, now something else happens. Now we get stuck here because we have an error but that is better than before because now at least, we do have an ID which we try to find. So what's the issue here now, why do I get cannot read property title of null? Well for one it's worth noting that null is printed here as well and that null should be stemming from my product model from find by ID when I console log the product. So it looks like we didn't find any product for that ID and what could be the reason for that now? The reason for that is that is that the ID in mongodb is actually stored a bit differently and we can see this in compass, the ID is actually such an object id thing. Now I did mention that mongodb stores data in bson format and this binary format of json is not just used because it's a bit faster to work with, it is but also because mongodb can store some special types of data in there and object id is such a type. It's a type added by mongodb, it's not a default javascript type, actually it doesn't exist in javascript at all and it's simply an ID object which mongodb uses because this generates and manages IDs which look random but actually aren't, so IDs for example are created in a way that if you create an ID now and an ID one second later, the ID one second later will alphabetically be a higher value than the previous one but that's just one thing. So object ID is an object provided by mongodb and if we look for equality, we can't compare \_id which in the database will only hold object id values with a string because a string is not equal to the object id and the string in here does not count, mongodb will not compare this, it compares the entire object, the entire object ID. So to fix this, we simply go into our product model and in there, I'll import mongodb by requiring mongodb from the package and now I can use mongodb to get access to that object id type. So here when I compare, I can use mongodb.objectid and I can create a new, this is a constructor, a new objectid to which I pass a string which will be wrapped by that. So now if I save that and I try reloading the page for that ID, now you see this works because now I create such an objectid object and therefore here when I'm telling mongodb find me all documents where the ID stored in the database is equal to that, mongodb will now succeed because we now pass on objectid object to the comparison instead of just the string. And now this works too. It's very important that you keep in mind that mongodb stores IDs and \_id and that it uses the special objectid type for that.

**Lecture 187**

**Making Edit and Delete buttons work again**

Let's complete the crud operations by ensuring that we cannot just find all products and add them and find a single one but that we can also edit and delete our products and that is of course something which would happen in the admin area and therefore, we need to work on the admin side again. For that first of all make sure that our admin page here where we list all products works again, feel free to pause the video here to try this on your own before we will implement it together. Were you successful? Well let's do it together. First of all in the admin routes, I want to comment in my get route here again so that we are again able to get all products onto our admin overview page. Now here we use admin control in get products to fetch and render all products, so if we go to the admin controller here, I of course need to re-add that get products function by simply commenting it in again. In there I use request user get products and this will not work anymore, so instead let me here simply use my product model and then use fetch all and this also returns a promise so we can still use then on it and we should still get all products but now through our own modified model. So if I now reload this admin products page here, I see all the products and now we can try editing and deleting them. Now for that, first of all we need to go back to our views and adjust them to take account of the new ID format, so here in products.ejs in the admin folder in the views folder, in there whenever we reference the product ID like we do here, it's products.\_id, the same here, product. \_id, that is really important. Now with that in place, if we reload that page, our edit and our delete button should work again and should make sure that we edit or delete the right product. So let's work on editing the product in the next lecture.

**Lecture 188**

**Working on Product Model to edit our product**

Time to edit products and for this, I'll first of all work on my admin.js file in the controllers folder. There we have two routes that are or two functions that are related to editing products. To get edit product page which is responsible for fetching the product that should be edited and for rendering it and the post edit product page which is responsible for saving these changes to the database, I'll comment both back in and let's start with get edit product. There the majority of the code can stay as it is but when we then fetch the fitting product, we of course do this differently, we use product find by id and we find the product with Prod ID like this and then we still get back products however, well actually we don't get back products anymore, we get back one product here so no need to store it in a constant like this and extract it from an array, we get back one product automatically because we wrote that method in such a way and we therefore can render this product, so this should work. Now let's quickly have a look at the view that belongs to this controller action, so to this edit product.ejs file and in there, you know what I'm looking for, we want to make sure we're always using product\_id when using the product ID just to make sure that we extract the right thing from the document we get back from the server. So now with that, we get everything in place to hopefully make this edit button work partly, if I click on it, well almost, let's leave it on this page. The controller should work but in the routes in the admin.js file, I of course have a comment this get route back in, so let's make sure this is done. And now that if we now reload that page, we indeed see the form with the default values filled back in. So now this is working but of course it's not that useful to only see the data and we were able to fetch data before, the interesting part now would be to update it. Now to update it, let's go back to the product model. How could we update our product which is stored in the database? Well let's go to the constructor and let's add a fifth argument here, the ID and then I'll say this \_id is equal to the ID I'm getting here, you could name this \_id here too if you wanted to. So now we accept a kind of optional fifth argument, we're not passing it in the other places of the app but here I do at least create the option of passing this too. An ID will therefore be undefined and be auto generated by mongodb or we did set it and then when we called save, I don't want to insert one, I want to create a new product instead. So for this what I'll do is I'll simply check if this \_id, if that is set and if it is set, we'll update the product otherwise we'll insert it, so I'll move my const db function up there, I always need access to the database in both cases and now I'll create a new variable, let dbOp for db operation and in the else case where we insert a document, dbOp is equal to my connection to the database and then connecting to collection and then inserting one and then down there, I can use dbOp to follow along and actually return that. So now I have one case where dbOp is simply my insert command and then I return my result or dbOp is access to my database and to the products collection still but then here I use update one and as the name suggests, update one will update exactly one element. There also is update many where you can update multiple elements at once but here I know I only want to update one, so I can use update one and now update one takes at least two arguments. The first one is that we add a filter that defines which element or which document we want to update, so here again I'll pass a javascript object and we can filter for equality also or run more complex queries which you again can learn about in my mongodb course if you want to and here I only want to find a document where the \_id is equal to and now again I'll create a new mongodb objectid to which I pass this \_id, so the ID which is part of that object here. So I'm looking for a document where the ID matches the ID I have here in my product I'm currently working with and for that document, we now as a second argument to update one, we now specify how to update that document. This again is a javascript object where we describe the update and this is now not the new object, so we don't say this here as you could imagine that we tell mongodb find me the existing document and replace it with this, update one does not replace. Instead we have to describe the operation and we do this by using a special property name which is understood by mongodb, kind of a reserved name you could say, $set. This again takes an object as a value and here we describe the changes we want to make to the existing document which we found with this filter. And here you could actually say this and you would instruct mongodb to simply set these key value fields which you have in your object here to the document it found in the database and therefore since these are only key value pairs which exist in the document in the database, it will update the values of the document in the database with your new values. You could also write this in a more verbose way and explicitly say title should be equal to this title and so on but since we want to replace all fields, we can just say this here. And now with that we have a database operation that will update an existing database object, by the way the ID will not be overwritten or anything like that, only the other fields are. So with that let's go back to our admin.js controller and and in post added product, that is the part I want to work on in the next lecture to make sure that we actually are able to save our product.

**Lecture 189**

**Finishing the update product code**

So we worked on the product model to make the save method more flexible and now in the admin.js controller in post edit product, there we already have code where we find a product by id and this will work because find by ID is a method I created in my model, so here we have the product or maybe we should name it product data because here, I will now create a new product constant by using my product constructor, so my own product class and there, I will pass my updated title, my updated price, my updated image url and my updated description and my ID. Now actually, we don't even need find by ID then anymore, we can just get rid of that and for now also of that and simply create a new product with all the updated information and we now also need to pass the product ID, there you just need to make sure that you create a new mongodb objectid object. So make sure that at the top of the file, you do actually require mongodb and now you can of course also create a new constant, name it object ID or whatever you want and extract that objectid constructor out of mongodb like this and then you could just write new objectid and reference this basically, so we can now go down to post edit product and pass a new objectid here with new objectid and wrap that Prod ID you're extracting from the url here and wrap that or from the body of the request and wrap that into object ID. Now we have that new product and now we can simply call product save because we just modified the save method to support both creation and updating. So let me save all of that and let's give this a try. Let's reload that view here maybe, change the price to 19.99 and click update product, now we get a page not found, let's ignore that for now and let's instead go back to mongodb compass and click that refresh icon here. Now that doesn't look like it worked though right, the price is unchanged but I also did not get an error, so what's wrong here? Well you need to go to your admin.js routes again and uncomment this post edit product route otherwise editing products will not be possible. So now if we load that product to edit again and we change that price, now this looks much better or at least partly. If we have a look at mongodb compass, we see now it looks like I made a mistake with the assignment of values, so my description and I can simply edit this here, my description now simply was the url and the other way round, so let me change both and click update here. Now if we load this page again, it should work but the most important thing is our method worked. Now what did I mess up? If I go to the admin controller, I'm assigning price, image url and description, well that is in the wrong order, it should first be the description and then the image url but that was only a little logical mistake I made. If I edit this again back to 12.99, you see now this is working and this also works, let's see how it works if I edit two fields at the same time and this also works. So this is how we can update a product with the help of mongodb and with the help of update one.

**Lecture 190**

**One note about updating products**

Now one important note about the update video, what I did there is I actually passed an object id in admin.js controller to my product constructor. Now actually I could also just pass product id as a string and therefore remove my object id creation up there in the mongodb import, all of that is happening in the controller now and if I do that and I try editing this and change the price to 16, you see now it fails. Now if we go to the product model and we have a look at the save method, there I am actually looking for the right object but I'll have a problem with updating it because there, I'll try to set my object ID to a different object id to a string because here I'm just referring to this which will hold the unmodified objectid. So what I should do is I should automatically convert the objectid, the ID here which is a string to an object and the object in the constructor so that I can remove it down there because \_id will now always be an object id field, no matter if I'm using it in a filter or if I'm using it for updating. So now with that if I edit this again and I change the price to 18, now this works again because now I'm not trying an invalid update by trying to change the ID to some invalid value. So this is just an important note I wanted to add, you don't have to convert the ID in the controller file, you can leave that untouched but you can do a general conversion in the product.js model file which is the better approach of doing that.

**Lecture 191**

**Deleting products**

So now that we're able to add, fetch and update products, let's work on deleting them. Now for deleting, you could implement different approaches, you could add a method to the product class so that you can create a new product object and call delete on it just as you called save on it but I will go with a static method and I'll add a static method which I'll name delete by id and where I expect a product ID as an argument. Now in here, of course again we need access to the database, so let's call get db to get access to that one central database connection we configured at the start and then on the db, again it's time to reach out to the products collection and on that collection, we want to delete an element. Now as always, you'll find more in the official docs in the crud operations, if you click on the documents you see that just as we had insert one, insert many and so on, we also have delete many and delete one. Now I don't want to delete many products but exactly one, so I'll delete one product here and you need to specify a filter now, so again pass an object where you describe how to find that product and again it will be our boring ID equal to check here. Now here product ID is an argument, so we need to convert it to an objectid manually again by passing it to the objectid constructor and now mongodb will go ahead and delete the first element it finds that has this criteria fulfilled. So I will return here and of course just as before, we can add then and catch here if we want, we can catch any error and console log it and we can also work with the result of this operation, we could console log it, I'll just console log deleted and that will be it. So now we have the delete by id method, now we just need to add it. So in the admin.js controller file, here the post delete product route, let's comment it back in or not route, the action I mean of course. We extract the product ID and then here we had a different flow, I first of all found the product here, now we don't do that anymore, we just call product delete by ID instead, we pass the prod ID as a string and then we just have our then block here which won't receive an argument and yeah that is it, we redirect thereafter. So let's go back to our application, let's quickly add a new dummy product with totally invalid and uninteresting data and that fails, that's an interesting behavior, let's fix that in a second. Let's first of all check if deleting works, I get redirected to a page that is not found, yeah makes sense because obviously working in a controller is not enough, I need to go to my admin routes and comment in the post delete product route. So after changing this, let's now click delete and now I get no products here. Let's confirm in compass by quickly refreshing that page and our product is gone, so deleting works. Let's now see what's going wrong when I tried to add that product.

**Lecture 192**

**Fixing the add product functionality**

Deleting works but when I try to add product and I'll just enter some dummy data here, that fails and it seems to fail because I would imagine that in save here, we somehow make it into this if block. So that \_id is the problem and that makes sense because I actually do initialize \_id here at the beginning by creating a new mongodb object id. So even if we pass no ID and this therefore is undefined, this will create an object and store it in \_id, so \_id down there will always be defined and if it's just such an empty or automatically generated object id object, this should be the issue here. So how can I overcome this? Well for example with a ternary expression, we can check if ID exists basically, so if this returns true in an if statement and if it's the case, then I want to create my object id object otherwise I'll store null and null will be treated as false down there. So with this tiny change in place, with this ternary expression added in the product model, we should be able to add a product real quick with some dummy values and now it's created. So now this is working and it was just the fact that we were always creating such an objectid object which we shouldn't.

**Lecture 193**

**Creating new users**

We get the basic crud operations with mongodb covered, how you can insert, find, update and delete elements with mongodb and you also learned how to connect to mongodb. Now let me show you how you can now work with relations and for that, let's work with some users again. Now with sequelize, we didn't have an authentication flow and we will not have one here either but now I want to create new users at the beginning here. And for that, I'll first of all work on my user model of course, I'll keep it real simple but of course I'll again a normal javascript class again. So let me create a new class, user and add a constructor and in that constructor, I'll get a user name and an email let's say and then I'll just store this in a name property and in an email property. And you could add more, you could work with this however you want. I'll then add a save method to be able to save that user to the database and of course I'll also add a static method for finding a user by ID and here I'll get the user ID as an argument. Now here's your challenge, try this on your own, pause the video here and try implementing this on your own. Try using mongodb and that central mongodb connection we have to save a user with the help of the save method and to find it by ID Good luck, after a short break which gives you the chance to pause to video, we'll solve this together. Were you successful? Well let's start by first of all importing our mongo database client, so I'll name this db again and I'll import it by requiring our database file in the util folder and there I'll use or I'll access get db and I'll name this get db up here too. Now let's start with the save method and there I'll create a new constant, db and I'll call get db to store my database client in that constant. Now on that client, on this constant here, we can reach out to a collection and obviously this will now not be the products collection but we could name it users collection if we wanted to. Now in this collection, I want to insert one new element here and that new element will be this, so this javascript object we're in, so an object with a name and an email property, this is what I want to store as a user for now. Now with that, we can again use the then and catch if we want to or we just return that and let whoever calls this listen to that if there is need for that. Now we can also work on find by ID already and there I again will get access to my database client and then I will return db collection users and here, I now want to use find again to find a specific user. We can narrow the user down by comparing the \_id and now the important thing here is of course that you need to convert user ID which I expect to be a string here to an object ID. So let me import mongodb by requiring mongodb like this and now here I'll follow a slightly different approach which I also already showed before and I'll create an objectid constant and simply store access to it by accessing it up here but I'm not calling it, I'm not creating an object in here, I'm just storing the reference to the objectid class in my objectid constant and then down there, I can just write new object ID thanks to my constant up there and pass user ID to it. And this should find me all fitting users and I therefore get back a cursor and now I can call next to get the first and as we know only element that matters to us, so now I'm returning this here. As a side note, you could also use find one if you are sure that you only find one element and this will now not give you a cursor but immediately return that one element, so then this would be an alternative and I will use that here but of course this is something I haven't shown you before, so if you used find and next, that's perfectly fine. So here I'll use find one and now I got two methods in place that should allow me to create new users and to then find that user. Now with that user model added, let me go back to the app.js file and in there I will import my model. So here I'll import user by requiring user from /models/user like that. With user imported, first of all we had some code here which I want to comment back in where I want to find a user with this ID here and then store it in the request, so I will still use that here, however I'll change the ID in a second. First of all when connecting, I also want to add some code to see if a user with a certain ID exists and for that, I will actually create that user here in compass. So I'll connect to the shop database and I'll create a new collection here, users and of course this collection name here should be the collection name you chose in your code. I'll create a new collection and then go into that collection and now if you have a look at your user model, here I'm storing a name and an email for a new user. So if we want to create one behind the scenes in compass, we can insert a new document here and here is the automatically generated ID and I can enter a name and you can use any name you want here and of course an email and insert that document. And now it's this ID, this part here between the quotation marks that matters to me, I'll take that and I'll use that in app.js. So here I could now create that user if it didn't exist but of course I now know it exists because I created it behind the scenes, later we'll add a real authentication flow where users can sign up by themselves, no worry. But here in my middleware where I find a user by ID, I can now search for that ID and I convert that in the user model, remember that is why I can use a string here. Ok so now I have that in place again, I get access to my user and now we're at least prepared to set up a connection between our product and our user which we can use later. So let me save that, that should work and in the next lecture let's use that user object and store a reference to that user in our database.

**Lecture 194**

**Storing the User in our Database**

Ok so time to use our user object and the question of course is where do I want to use my user object? Well I want to use that user object when creating a new product, right, when saving a product, I want to store a reference to a user here or embed the entire user data as you learned. However for products in users, you could actually find arguments for both approaches here, you certainly don't want to enclose all the user data in an embedded document because that would mean that if the user data changes, you need to change that data in all products but if you do include something which is unlikely to change very often, like the username for example, well then you could certainly go ahead and embed that together with the ID so that you always have that ID to fetch more data about the user if you need to, you've got to find by the method in the user model after all or that you have at least some snapshot data like the username available immediately, if that should change, you need to update it everywhere. The alternative to this is that you just store the ID, so just a reference and therefore if you need connected data, you always have to fetch it manually from two collections but on the other hand you might not do that too often and therefore here indeed when I fetch the product, I don't really need the user data, we're not displaying the user name anywhere in our app, so I actually just want to store the user id so that we know who is connected even though we're not fetching that a lot. Now what does this mean for our application here though? Well it means that when creating a new product, we can pass the user id, we can accept the user id here, so this user ID is equal to user ID, we store this as a property of our product now and with that, we have all we need in the product model. Now we need to make sure that we do pass that user data when creating a product, so in the admin.js controller when adding new products, here I want to pass null for the product ID because we dont have that when creating a new product but for the user id, I want to pass the ID of the user which as you know we now store in our request. We do this in the app.js file right, we have that middleware which we set up in an earlier module where we find that user and where I then store that user in our request. Now this is a bit of a constructed example because I'm storing it just to extract the ID which I hardcoded here, so we could just hardcode it into our code anyways but the idea here of course is to show you how you can extract the user in one central place and then reuse it in any other route and that's the idea here. Later once we get an authentication, we'll manage that user a bit differently. So for now, we just have that user object in the request and therefore in the admin.js controller, here I can access request user and then \_id which will be just a string here because when retrieving data, the object id is converted to a string for us, so we have just the string here. Now with that, we should actually store the user ID when creating a new product. Let me first of all delete that old product and then let's quickly add a new product here and I get an error that I can't read Property ID of undefined and this should be coming from inside our admin.js file here when I try to access my user \_id, so something seems to be going wrong here. I'm storing the request user here though, let's go to the user model, find by ID, let's add then and catch here and catch any error we might be getting when fetching that user and here I'll also console log the user object I'm getting right before I return it again. And with that let me try to reload the admin product, add product page and I now get an error. We did retrieve the user here but then I sent duplicate headers by the look of it, yeah because I'm calling next too often here in app.js, . since I added the other code again, I don't need to call next at the bottom of it. So let's save that to not call next too often. Let's now reload this page here and now I just fetch the user with the ID which is a string here, so that is actually valid. Now let me try adding this again and now this works, so it was that error with the double next. So now if we have a look at our products here, we see that products now also have a user ID which is just a reference pointing at the user who did create that product which is one way of establishing relations the way you already know from MySQL of course, this is not an embedded document but here, the argument simply is when we're fetching products, we don't really need any user information hence we do it just like that. This of course will change once we start storing orders, there you could say it does make sense to store information about the user, for example the e-mail at least and for the product you want to store the title and the price maybe. So there it makes sense to suddenly aggregate this together. Speaking of that, why don't we work on cart items and orders next.

**Lecture 195**

**Working on Cart Items and Orders**

So we worked on products and users, it's now time to work on the cart and the cart items again and that will now change now that we're using mongodb. So let me start with the cart model, now what is, what's the overall goal we have here? Well obviously for every user we have, we want to store a cart right and that user will have a cart and that cart will then hold the products. So actually with mongodb, this is a great place for embedded documents because we have a strict one-to-one relation between a user and a cart and therefore there is no need to manage this with a reference, I could actually get rid of my cart items, my cart model so that I just have product and user and for now, also the orders and now on the user model, there I also want to store my cart items. Now the question is how, the answer is we get our shop.js controller and in there we got code to store something in a cart, here for the user we got the cart and then we got the products to see what's in there, update it and then save it back. Now we can do that with the help of the user model. Let's say on a user object, instead of just having save I also add a new method, add to cart, something like this. Now here add to cart would be a product I want to add to the cart of course, essentially what we're doing here in shop.js already, here we already fetch or we got everything we need to fetch a product which we could then add to the cart. So I expect to get a product here which I can add and then in add to cart, I can have all the logic I need to find out if that product is already inside of the cart and therefore if I just want to increase the quantity or if it is not and therefore I want to add it for the first time. Now to understand how this works you must not forget that add to cart will be called on a user object and we'll create that object with data we fetched from the database with the help of find by ID, here we will return a user. So therefore we just need to be able to accept more data in a constructor, for example the cart so we can also store the cart in our javascript object here which will be based on the data stored in the database. So now we can assume that we'll have a cart property on our user and now we just need to find out if that cart contains a certain product already. Now the cart will essentially be an object let's say that looks something like this, an object with some items, so an object which has the items array in there. So what we could do if we assume that this is the case, we could create a new constant, cart product and then use this cart items and find, let's say the index of a product in that cart with the same ID as the product we're trying to add again. So I'll pass a function to find index which simply is a function javascript will execute for every element in the items array and then here I want to return true if I found the right product in my items array and this will be the case if cp which is the product in the items array, if the \_id there matches the \_id of the product I'm trying to insert. If that is the case, so if this returns a valid index, so something else than minus one which would be the default otherwise, then I know this product already exists in a cart and then I just need to find out what its quantity is. Now we'll do this a second step, first of all we can assume that there will be no products in the cart because we're just starting from scratch, so let's add the code to add a product without checking whether that product exists yet, so we can comment this out for now and instead create a constant, maybe named updated cart which is an object where we might have an items property which is an array where I will now include my product. However not just the product as I'm getting here, I also want to add a quantity field, so I'll actually say product quantity and this is how you can add a field on the fly in javascript equals one. Now another approach, maybe a bit more elegant than that is that in items, you create an object with curly braces because we'll add an object here and then you use the javascript spread operator, three dots to pull out all properties of this object so of the product object and then with a comma, you can add or overwrite a property, so here I'll add the quantity property and set it to one. Whatever approach you choose, this will create an object which holds an items property which is an array of in our case only one object, one product with an additional quantity information. And now I want to update the user to store that cart in there, so I will get access to my database with get db and I'll reach out to my users collection and I'll update one user in there and I'll update the user with this user ID here, so what I need to do is I need to accept that ID as an argument. So this \_id will be equal to ID let's say and then here I can say I want to find the user where \_id is equal to new and I'll use object ID which I extracted up here because I need that for the comparison where this is equal to this \_id and once I found that, I'll describe how to update and I'll use $set where I pass an object which holds all the information about which field to update in which way. And here I essentially want to keep everything as it is, I dont want to change the user name or anything like that, I'll just set cart equal to updated cart, that is it. So cart which I expect to have in a user in the database will now receive updated cart, so this object as a new value which will overwrite the old one and this is important, it will not merge this with the old one, it will not merge the elements in the items array, it will simply overwrite the old cart with the new cart. Now I can return this here, this update one call and what this should do is well, it should update our user to add a product to the cart, for now it will always overwrite any existing products in the cart, we'll fix this later but for now, this should work. Now let's wire it up in the next lecture.

**Lecture 196**

**Adding the ‘Add to Cart’ Functionality**

So we're making progress on the add to cart functionality, at least a very basic first version of it, let's now go to app.js to wire it up. When finding a user, I'm storing it in the request like this and this actually should be all right, however it's important to understand that the user as I'm storing it here will just be an object with the properties, so the data we have in the database. All the methods of our user model will not be in there because the user I'm getting here is data I'm getting out of the database and the methods aren't stored there, they couldn't be stored there. So to have a real user object with which we can interact, I should actually create a new user here and then simply pass user.name which is my username, how it's a stored, user email, user cart and user\_id, so I should create such a user object and store that in the request because now this enables me to really work with well all the user data or with the whole user model and this now allows me to also call all these methods like add to cart on it. So with this changed to app.js, we can go to the shop.js file in the controllers folder and then there, I want to work on the post cart method for now which allows me to add an element to the cart. I do get my product id here which is nice and then we have all the logic here for fetching what's inside the cart and for updating this, I will actually comment this out because for now we don't need that. So instead what I want to do is here I want to fetch the product I want to add here, so I will use my product constructor, my product model and I will use find by ID to find a product by the product ID I'm extracting here and I'll just add the then block here and in here, I should have the product that I want to add to the cart. Therefore in here, I will now use request user which now is our full user model and call add to cart and pass that product I fetched as an argument because in the user model, add to cart does expect a product and then I return the result of update one which will be a promise, so back in the shop.js controller, if I return that promise instead of then here, I can simply chain another then block where I get the result of in this case my update operation, so I'll just console log that result. Now with that, let me go to the shop.js routes and let me add that cart post route again. Let's now go back to products and let's give this add to cart button a try and I get an error here. The problem here should be that if we have a look at our view, at our product list.ejs file, that for add to cart, we have that add to cart ejs snippet that include where I pass my product to but in add to cart ejs, I still access product.id instead of \_id, so I need to change this in order to pass on the ID correctly. So now if I reload that products page and I click add to cart, now we get stuck here but actually we were successful in a way. We did actually modify something as our result tells us here and if we quickly go to compass and have a look at our users, you see there is an embedded document, a cart document with items with an object which holds product data. Now that user ID here is a bit redundant because were already are in that user, we could strip that out and only store what we want but it also that doesn't matter too much. The important thing for us here simply is that we now store a whole product which we do also store in a separate collection as part of an embedded document in our user, so we clearly have duplicate data here, we have the same product here as an embedded document and we have it in products. So this is maybe something which we should change because if we change the product right now, if we change the title or the price, this will not be reflected in the cart and in the cart we should have correct data because if the price changes, we can show the wrong price in our cart, right. So actually we'll need to tweak this a little bit more. In our user model where I store product in add to cart, well I actually don't want to store all product data in this object and then the quantity, I just want to store the product ID by creating a new objectid here and passing product.ID, .\_id as an argument and the quantity, so now only the reference and the quantity and not the rest of the data. If I now save this and I click add to cart again, it gets stuck again, it doesn't matter too much right now. If I now update my users, I get an error here, that should be Product\_id, product\_id, I forgot the t. So now again if I update this and I click add to cart again and I reload my users document here or my collection, now you see I'm only storing the reference and the quantity and this is exactly the information I want. If I now want to display information about the product, I have to fetch that manually because I only have the ID, that on the other hand is all I need for fetching information, so that is what it is. But on the other hand if I now do change my product, I don't have to change it in the products collection and my users because that would be a lot of work for data that might change quite a lot.

**Lecture 197**

**Storing multiple products in the cart**

So I'm now able to store some data in the cart, at least in a very basic way but obviously we don't just want to overwrite the existing cart all the time, we want to be able to store multiple products in there and increase the quantity in case we already do have a product in there. So we need to finteune our code a little bit and we already started, so I'll comment this back in where I do check whether a certain item already does exist. Now I just need to tweak that code a little bit, here I need to look for Product ID because that is where I store my product IDs in the items in the cart. So I'm looking for product ID because I'm storing the ID in productid down there and now if that is something else than minus one, we know that this product already exists in the cart. So I can add a new quantity field again and set this to one by default but if cart product is greater than zero or greater equal than zero, so if it's anything else but negative basically, then this means this product already exists. So then new quantity is cart product, that is actually the index so let's maybe name it cart product index, so this is then actually equal to this cart items for that given index we just identified and there we'll have a quantity plus one. So this is the new quantity if that item already exists, so if it already is part of our cart, if not we'll go with the default of one and therefore here, I will always update to new quantity. However I also don't want to update by always overwriting items with a new array with exactly one object, instead I want to add a new object to that array if the product does not exist in there or if it does exist in there, I want to update that one product, so how do we do that? Well one of the simplest ways and you could use other approaches where you leverage some functions mongodb has but one of the clearest approaches you can use is that you get the updated cart items, that you create such a constant and then, you access your cart items and you create a new array where you copy in all the existing elements with the spread operator with the three dots. So this gives me a new array with all the items that are in the cart and they are now stored here and I can now edit this array without touching the old array due to the way javascript works with reference and primitive types. So now I can edit my updated cart items and now I just need to differentiate, do we already have that item in the cart or not. So I'll actually move that up here before my if check and then here if I make it into this if statement, I know that we have this product already. In that case, I can access updated cart items for the cart product index I found, so now I have access to that item I'm interested in, I know that it already existed so I can set its quantity equal to the new quantity like this. Now else if the item did not exist before, I'll take my updated cart items and simply add a new one with push. I'll add a new cart item and I'll add a new cart item which looks exactly as described down there, so I'll just grab that code and add it here. So now I either update the quantity of an existing cart item or I add a new one and then down there for updated cart, I can always set my items equal to the updated cart items because that will always be an array with all the old elements because I copy it first and then with the update added, so either with the quantity increased for the existing element or with a new element added to the cart. So then I can safely have my updated cart down there and save that to the database with all the updated items in there. So if I save that now, I'll first of all add a second product real quick so that we, whoops, so that we have some alternatives here and then I'll go to products and I'll add this first one with $12 to the cart. Now if we update this in compass and we look into the cart there, we see this was added again even though it already existed. So our logic is not too convincing, if I press this one more time, we now see it in there three times I'd imagine, yeah, three times the same object so clearly our logic fails. Let's see what's wrong and I found the issue, it was this comparison up here. The problem we have here is the product I'm getting here, right, the product I'm getting as an argument is a product I just retrieved from the database. Now the \_id we have in there actually is treated as a string in javascript but is not exactly of type string, since I'm using three equal signs in my check here however, I am telling javascript that this should only return true if these two do not only match by value but also by type and technically this is no string even though we can use it as such. So one solution is to use two equal signs or to use toString on both here, so here and here to make sure we only work with strings here in both cases. And with this adjustment made, I can edit my object here by double clicking into some field and then we can mark the latter two objects here in the array for deletion by clicking on the cross on the left, click update thereafter, now this is updated, we only have one item in there with quantity one but now if I go back and I click add to cart and I do update this in compass, I should still only have one object but now with quantity two. If I do add my other element here though by clicking on add to cart here and I do update, now I should have two elements in the cart, one with quantity two, one with quantity one. If I click add to cart again and I update this again, it should still be two objects, now both with quantity two and so on. So now this is working, now I am updating the cart, this is now an add to cart functionality, a basic one implemented on our own. Now of course we want to be able to also display the cart items.

**Lecture 198**

**Displaying the Cart Items**

Now whilst we're able to add product to the cart now with this code, let's make sure we can also display them when fetching them, when going to the cart's page here, to this page because right now we don't support this page because we still have that route commented out and we've got no logic to fetch all cart items anyways. So time to go into the controllers folder into the shop.js file where we do have the get cart route. Now this request user get cart thing will not work anymore, so let's make it work. Let's go back to the user model and in there we have add to cart, let's now also call or add a get cart method, you could name it however you want of course. The idea here is that we do indeed return the cart items, so that in the controller we have the cart items there and we can start outputting them, in the end what we need as you can tell is just the products, right, so we want to have a list of the products with the respective quantities. Now get cart exists on the user who already has this cart property, this is the mongodb way of thinking about relations, we don't need to reach out to a cart collection because there is no such collection, instead here we can simply return this cart and that's it, this gives us access to the user cart. Obviously we could have directly accessed the cart property on the user if we wanted to. You could add more logic to transform it or anything like that but in the shop.js controller, I can now already call get cart and have the cart therefore. Now on that cart I can't call get products though but I don't need to because the products are also already included in the cart or at least their references, right because what I'm storing in the user model with add to cart is the ID of every product, this is what we store. Now and that is exactly something we can change in get cart, instead of just returning this cart, it would be interesting to return a fully populated cart, so a cart with all the product details which we also require. And to do this, we of course have to reach out to the database again, so let's get access to our database client and let's return the result of a database operation where I do reach out to the products collection now because I have all the user data, I have all the cart data, now I need to fill it with some live from the products database and there I want to find all products that are in my cart. Now how can I do that? Well for this, we can use a special query syntax mongodb supports. In find I can tell, I can say I want to find all products where \_id is equal to and now I don't pass an ID here because I'm not looking for a single ID instead I pass an object because this allows me to use some special mongodb query operators of which there are many covered in detail in my mongodb course or in the official docs of course but we are looking for the $in operator. And this operator takes an array of IDs and therefore every ID which is in the array will be accepted and will get back a cursor which holds references to all products with one of the IDs mentioned in this array. So all we need to do is construct such an array, so product IDs can be constructed by using this cart items and remember, this is an array of objects that look like this, so objects that have a product ID and the quantity. Now we're only interested in the product ID here, so I'll simply map this, this is a default javascript function, I'll map this to transform every item in there and I simply want to return the product ID. So what I'm doing is I'm mapping an array of items where every item is a javascript object into an array of just strings, of just the product IDs and this is then stored in this new product IDs constant. This is then my array which I want to use here to tell mongodb give me all elements where the ID is one of the IDs mentioned in this array here. So this gives me a cursor with all the matching products, now I'll again use toArray to quickly get that converted to a javascript array and then I'll add a then method. So in this then method, I'll have all my product data for the products that were in my cart. Now of course we want to add the quantity back to every product because that is something that is important to us, now how can we get that information back into there? Well in this then block, I'll again return a mapped version of my data, so a mapped version of my products array where every product will be converted a little bit. I'll return a new object for every product which is fine because every product is an object and I'll distribute all the existing properties, so I want to keep all the data I retrieved but then I'll add a new quantity property and that quantity property of course needs to be populated with data I know or I have on that product. Now we of course have the products stored in the cart of this user, so what I can do is I can use this cart and make sure you use arrow functions here to ensure that this inside of this function still refers to the overall class, with normal functions it would not, so use this cart in here, access my items and simply find the item with that ID at hand here. So here I'll have my cart item again and I'll return true if that item has a product ID toString, that is equal to the product \_id to string of the product we just fetched from the database. So this can look confusing but in the end I have an array of products here fresh from the database, then I want to transform this which I'm doing with map, map takes a function that executes on every element and products which describes how to transform this element, here I'm basically returning the new value which is an object where I still have all the old product properties but I add a new quantity property and to get the right quantity for that given product, I reach out to my cart items which exist on that user and I again use a built in javascript method, find to look at all elements in cart items with this function here and then identify the one product where the product ID I'm storing in my cart items matches the ID of the product I have fetched from the database and since with map I'm going through all these products, this will also vary for every run. Now the last thing is this whole thing here, the whole find method in the end just gives me the product object, I want to get the quantity though. So now from the cart items I have, I extract a quantity for the given product. And now with that code, get cart should return products which are enriched with all the data that is stored in a product's collection because in users in the cart we'll only store the reference and this is what we need to do in mongodb if we then have a connection between two collections with a reference, we need to merge them manually as we are doing it here and with that merging being done manually here, we can now use that data. So get cart should now return a cart with all the information we need, so on the user we can call get cart and then we know we get back some products here, so we can delete that here as well as this and then render our view with the products we fetched. Now on that view, so the shop cart view here, we just need to make sure we output our products correctly because we cycle through them but now due to the way how we merge this, every product will itself already have the quantity here and whenever we use .id, we should have \_id as you learned. Let's see if that works by going to our routes folder and there the shop.js file and let's add that get cart route again. So I'll reload this cart page here and this is looking pretty good, now let me add a new product, third with some image and some description. Now this exists, not in the cart though unless I add it to the cart. Once I do that and I go to the cart, we see it there too. Now the one thing that is still not working is that if I do add a cart item, we're stuck, let's fix this in the next lecture.

**Lecture 199**

**Fixing a Bug**

So everything works but when clicking add to cart, it would be nice if we don't get stuck here, so something seems to be wrong. If we have a look at post cart, well the thing is we never send a response here, we do add our product to a cart but that's it. Now in the old code, we redirected to /cart and that is what we should do here too, so instead of returning this result which doesn't matter, no one's interested here, I just want to call res redirect here or not here but here and then I guess we can return this to see the result of that operation here if we wanted to but the important part is that we redirect at some point. With that, now if I click add to cart here, I'm redirected to the cart page, awesome. Now let's make sure we can delete cart items. As always feel free to try this on your own, in the next lecture and thereafter, we'll do this together.

**Lecture 200**

**Deleting Cart Items**

We're nearing the end, let's make sure we can also delete cart items now. For this I'll first of all get rid of that code I previously had for updating the cart, we outsourced this into our cart model for now and now let's work on the post cart delete product action here in the shop controller. I of course want to be able to delete items from the cart and to delete items from the cart, the user model is again a great place to work on. There I'll add a new method, delete item from cart, the name is up to you and I only need the product ID here to remove the entire product from the cart. With that ID passed, we can create a new constant, updated cart items and first of all copy all existing cart items, again with this spread operator. However we can even take an easier route and just use this cart items and then the built in filter method, this is again a method provided by vanilla javascript. Filter allows us to define a criteria on how we want to filter the elements in that array, so in this case the elements of the items array and then it will return a new array with all the filtered items, so all the items that make it through the filter. The filter is a function here which runs on every item and then we return true if we want to keep the item in the new array or false if you want to get rid of it. Now I want to keep all items except for the item which we're deleting, so I will look into each item and these cart items are objects of the structure we define up here, they are objects with a product ID and a quantity. So I'm looking for the product ID now, down there I'm looking for the product ID and if that is equal to the product ID I'm getting here, then I know this is element I want to remove, so I'll check for the opposite because remember, I return true here if I want to keep the item, so I want to return false if I want to get rid of it. So this should return false if I want to get rid of it and I want to get rid of it for this condition. As before with toString, we can guarantee we're working with the right type of data here and now we have the updated cart items which we just need to save back to our cart and therefore to the database because these is already are the cart items with the one item we wanted to get rid of removed. So I can simply copy my code from up there where I update the database, like this and when I say updated cart here, well I simply mean I want to assign this to an object with an items property because that is what our cart has, right, we have items in there and items is equal to our updated cart items and that is it. This now will update the cart to have all cart items except for the one we deleted. So back in the shop.js controller, here I don't need to call get cart instead on the user, I can call my new method, delete item from cart which you just added, we can call that here and pass the product ID as an argument and that we can get rid of all that code and simply redirect once we're done, so let's see if that works. If we now save this, let's go to the routes and add this post route for deleting cart items, reload this page here and let's delete the second item and this is looking good. It's gone here, no errors here so this is looking good and if I update this in compass, we see that for this user we indeed well obviously only have the other items. So this worked and that is how we can delete cart items with these.

**Lecture 201**

**Adding an Order**

We're coming closer to the end of this module, only the order part is missing and I want to include that too because this will show you another way of creating relation. With the cart, we had like the middle way, the cart is an embedded document but the items on the cart then are a combination of references and extra metadata. Now for the orders, this will change. So let's work on the orders now and for that I'll go to my shop.js file where we post the order. Again I want to store orders on users, so in the user model, we can add an add order method and this doesn't take any arguments because the cart which will be passed as an order or as the data for the order is already registered on this user, so all I need to do here is I need to add the orders to my user or the other way around. You could also argue that you want to store the orders in a new collection because you might have thousands of orders and you don't want to embed them all into user objects because these objects will then get pretty quick, carts don't get that big but an order history, well often that can get very long so I will actually work with a new collection here. I'll still create order here as a method on my user though, so I'll reach out to my database client and then reach out to a new collection, orders. There I want to insert one new order now and I'll return the entire thing and that one new order will be well the cart I currently have, right, that is essentially what I did before too when I added an order, I added my products in the order model, we had no special fields for that, we just had our products and then the quantity of every product and that is still the same I want to add here. So you could say I'll insert one and I'll insert my cart, so this cart which refers to the users cart. I insert this as an order and then when this succeeds, I set this cart equal to an object where items is an empty array, so I basically empty my cart at this point. Of course I don't just want to empty it here, I also want to clear it in the database, so we'll also copy that code for updating my user in the database and I will return that here in this then block and there, I will simply search for that user and set the cart equal to an object where items is an empty array. So now I cleared both in the user object as well as in the database but I also insert the cart into the orders collection before I clear it and that of course is the important part. With that, we already are adding the order. Now in shop.js, in post order where the order gets placed, I can therefore remove all the code up here and simply say add order, simply execute this method and then redirect once this is done and in add order, I will store my cart as a new order and you could of course also add some new fields like the total of value for example if you wanted to and that is it for now, so thereafter I redirect. Now let's see if that works by going to the shop.js file and re-adding that create order route and let's click order now here. Now I have page not found because orders, the orders page the route wasn't added again but let's refresh the entire page here and my items in the cart certainly are gone but I don't see new orders in the shop database, I don't see that collection here but here it is after updating this once with the update arrow in the top left corner and in orders, I have one order with indeed the items I had in my cart previously and the cart is now empty. So let's now make sure in the next lecture that we can see our orders page again.

**Lecture 202**

**Adding Relational Order data**

Now that we're able to store orders for the user, let's add a new method to the user model to also get orders and this will be a method we need to make our orders page work again. Now getting orders is pretty straightforward, we reach out to our orders collection here and then we need to find all orders for the given user and oh, how do we do that? Well there's one thing which we forgot right when adding orders, our orders right now contain the content of the cart but no information about the user to which they belong and that is something we need to change first of all. So let's pause getting orders for now and let's tweak our add order function here. We insert one order and right now the orders only are a cart. Well this is not the entire truth, right, we also need information about the user, so let's create a new constant order in here which is the object we actually want to save and in there we certainly want to have the items, so we want to have the cart items in there, that is for sure. But we also want to have some information about the user and for this I'll add an embedded document where I add the id and here, I want to create a new objectid based on this ID, so on the ID of the user we're working with but I also want to store the name which we have as a property here and the email. So here I will duplicate data because this will then end up in the orders collection and in the users collection but I don't care too much about this because the data I have been here, the user data I have in here might change for sure but it doesn't need to be updated on all the orders because if you had like processed and open orders, for all processed orders, you wouldn't care too much if the user email changed because you might not need to touch it there and of course if you do care, you can always get rid of all the data. So now even if the user name would change, I could be fine with not changing it here and only in the users collection. I also want to store more information about my products by the way because right now, my items are really just the product IDs but the idea was that we have more information about our products than just their IDs. To be precise when we have a look at our orders.ejs file, here I also output the product title for example and you could argue you also want to show the price. So storing some extra information would be useful too, therefore we also need to work on this cart items again and we need to fetch some data from our products database, so we need to tweak that add order method a bit more. Now we learned how to fetch product information with get cart, there we actually get a cart with enriched information about all products. So actually what we can do is in add order, I can first of all call this get cart and then add then to work with the data get cart gives me, so with my updated products. So inside then here, I get my cart products and these products will have all the product information along with the quantity and I then create my order inside of that then block, so once I know that I have that products data because outside of that then block, the code would execute too early. And then my items here will be my products, so an array of products with the product information and the quantity, so now the product information will also be part of the order. And here I really don't care about that information changing because if it should change, for orders we need a snapshot anyways, if the price of a product changes, that doesn't affect the past order, so there we wouldn't want to update the price even if it would change. So for orders, such a snapshot and therefore an embedded document is a great way of relating the order and the product because the product data might be duplicate but it doesn't need to change in the orders collection because there, we want the snapshot. So now I have my products in there and some user data, not all but some and now I'll take my insertion code here and move that into this then block after my const order thing and move this then block up. So now the order is we get the cart which is essentially an array of products, we create an order with that data, then we insert this order into our orders collection, that's new, we need to insert that, we'd return the result of that and then here, we know that we were successful with inserting this and we clean up our existing cart. With that, let's test this. Let's go back to our application and add a couple of products to the cart, let's say like this and now let's click order now and I get an error, yeah because I need to return the result of this add order thing so that outside of add order, to be precise in my controller, I can call then on that. So that needs to change, so let me go back to my cart here, it's actually deleted because we did write it to the database, so if I do update orders, we actually can see it there. This was our old order, we can delete that with the trash icon here, the second document is the new order we just added and there you see items indeed does have all the enriched product information, the snapshots of our products and the quantity in there and we have some user data. So this works and now with that little change I just made, let's also see if that works again. If I order now, yes now we get no error. So now we can work on that cart page and on getting our order information.

**Lecture 203**

**Getting Orders**

So now that we added some relational information to our orders, we can go back to getting all the orders in our user.js file in the models folder. There I have get orders and now I want to reach out to my orders and simply find them all or at least find all for that user. Now how do I find all orders for that user? Well remember, each order has a user object and in that user object, we have the ID of that user, so we need to compare that ID to the current user ID. Now to do this, we add a filter and now in mongodb, you can also check nested properties by defining the path to them, the only important thing to know here is that you need to use quotation marks around the path and then you can say check user and then the ID for the user. You do that by specifying user.\_id and this will look for \_id in the user property which holds an embedded document and then here I can compare it is to a new objectid for this ID and this should give me all orders for that user and this will now be more than one, so again we can use the toArray shortcut and return that data to return an array of orders for that user and our user here for example has two orders because we got two orders for that user ID. So now we want to output that order information, so let's go to the shop controller, get orders, now I named my function get orders here too, so calling that here should work. We don't need that anymore, that include thing was related to sequelize. I then get back my orders and I passed them to the orders view, so let's now just inspect that orders view here and there I loop through all my orders and for every order, I have an \_id, that's important and then in the orders, I don't have products, we could name it as such but I named it items here. So I'll loop through all the items now, there will be product information in there though, for example the title and the quantity which is now a top level field in product so we access it like this. And with that out of the way, let's save that and let's go to the routes folder and there the shop.js file and comment in that one orders route you removed earlier. And let's now click on orders up there, get db is not defined, that should be a typo in get orders in the user model, yeah this should be get db with a lower case b. So let's reload that page and this looks much better, now we got our orders with the ID and with the items we ordered. So this is now working, now we got that basic shop functionality working again which we had work earlier with sequelize, now we're doing it with mongodb. Now obviously just as we used sequelize for SQL to make certain things easier, we can do the same with mongodb and there also, we can find an alternative that makes our life a bit easier.

**Lecture 204**

**Removing deleted items from cart**

Now one problem I just noticed when I clicked around and deleted some products here is of course that when I delete a product that wasn't a cart of a user as I have it here, I added one product behind the scenes to a user product with this ID and if we have a look at products and I don't want to update anything, if we have to look at products, this product ID is not the product ID I have in my cart here. So I deleted a product which was part of the carts of different users, now what would you do about that? For one it's important to notice that our app works correctly because when we try to load the cart, when we don't find the product data for an item in the cart, we just go ahead and well ignore that basically which is what we want here but how would you solve this? Well what you would do or what you could do, one approach that would makes sense is that you add some kind of worker process which is something a bit more advanced and not directly related to building web applications with node, basically a script that runs on the server and checks for such cases in your database once every 24 hours or something like this, where you basically scan your users, your carts and look for products which you don't find in the product collection anymore and then you clean up these carts, that would be one thing you could do in an application to clean this up. Depending on your requirements, you could even have a cleanup script for the entire cart which resets the entire cart every 24 hours or every seven days. An alternative approach that you could of course look into would be that when you load the cart page as we are doing it here, so when you're calling get cart on the user and you know that there are cart items on the user object and still the products you get back is empty, then you know there's a mismatch between what you have in your cart and what's in the database and in such a case, you could then issue some behind the scenes request, so basically with exactly the tools you learned about to update the cart of that user to match the product you got back from the database. So if you got an empty product array and you have items in the cart, you want to reset your cart. If you've got less items in the data you get back from the database then that's in your cart, you want to find out what the difference is and then update your cart accordingly. So this would be strategies you could employ here. For now, I will move on and not focus on that as this of course also depends highly on your application needs but especially the second strategy is certainly something you can implement with the knowledge you learned in this module, so this would be a great challenge you can take to add the functionality to clean up your cart on the user collection whenever you find out that in get cart, there is a mismatch between the products you get back and the products you think you have in your cart.