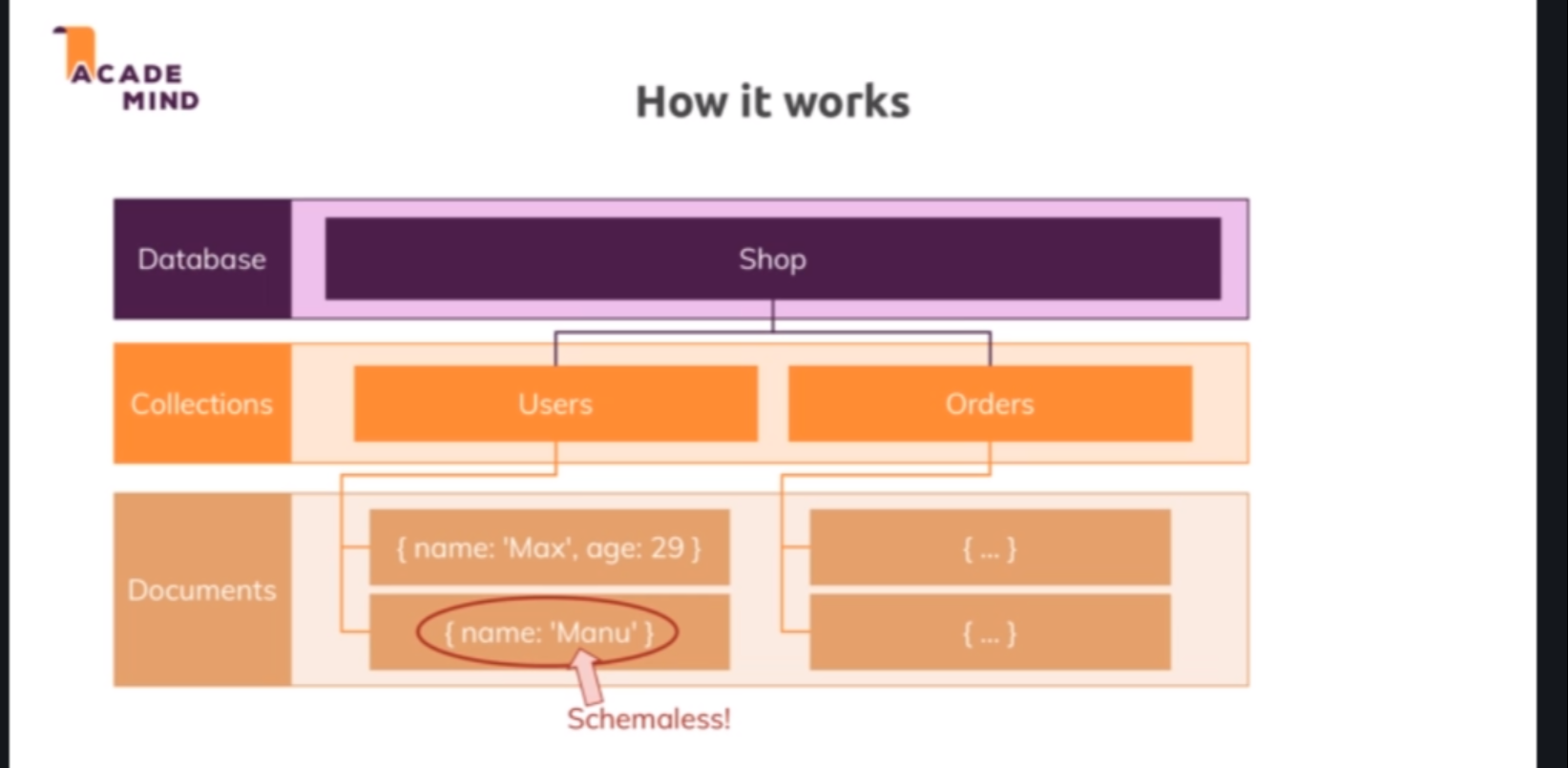
**Lecture 176**

**What is MongoDB?**

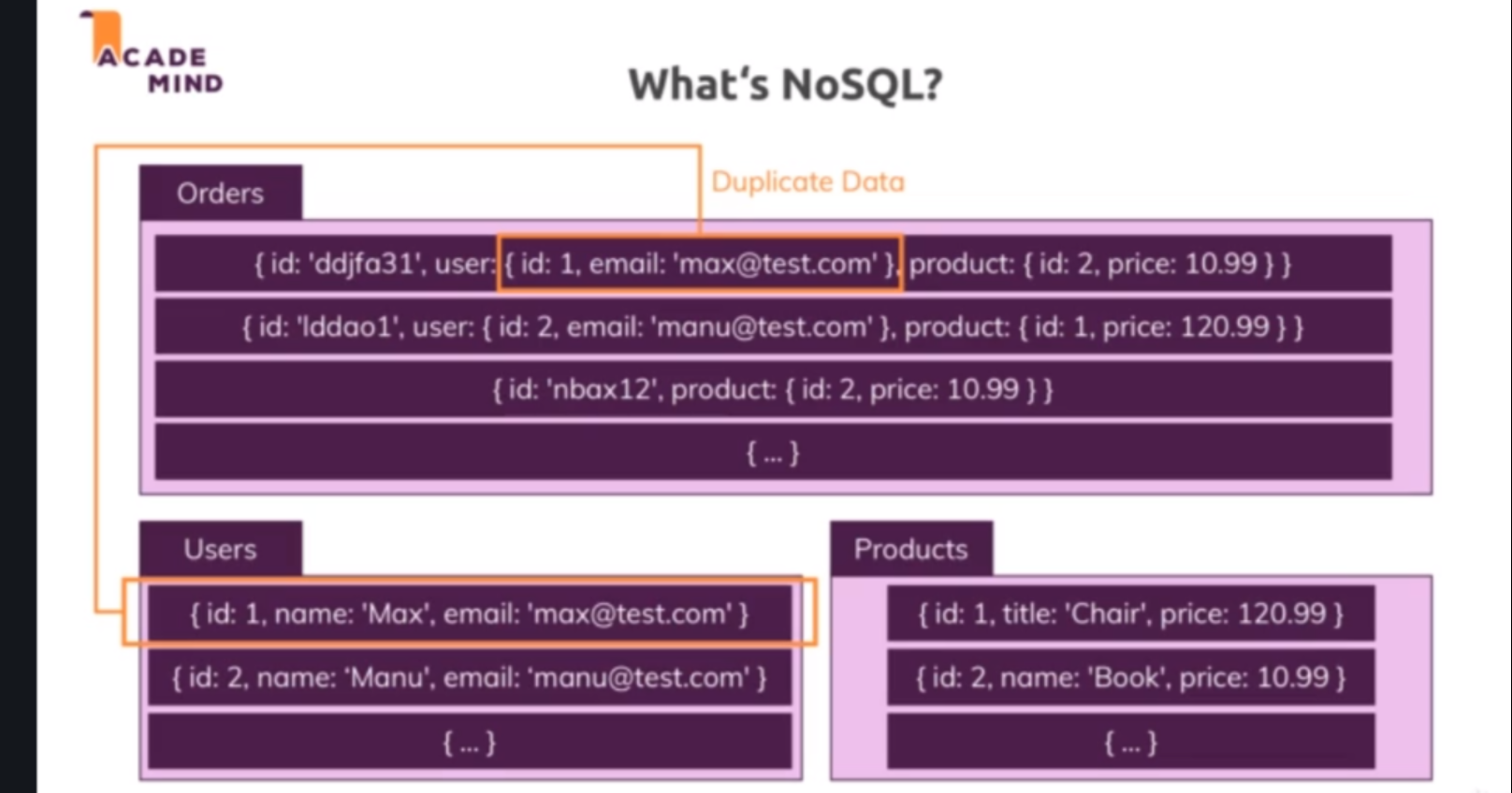


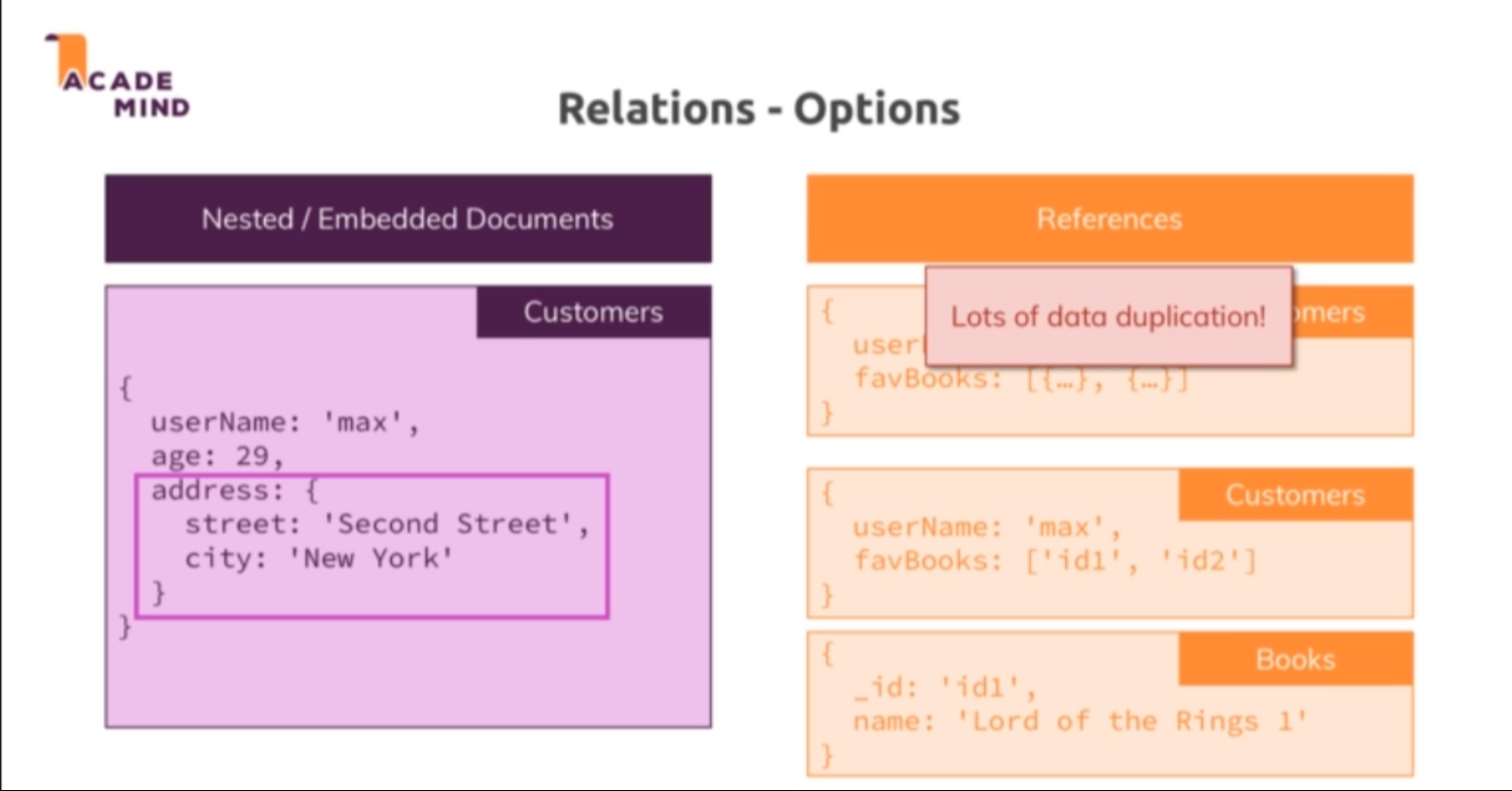


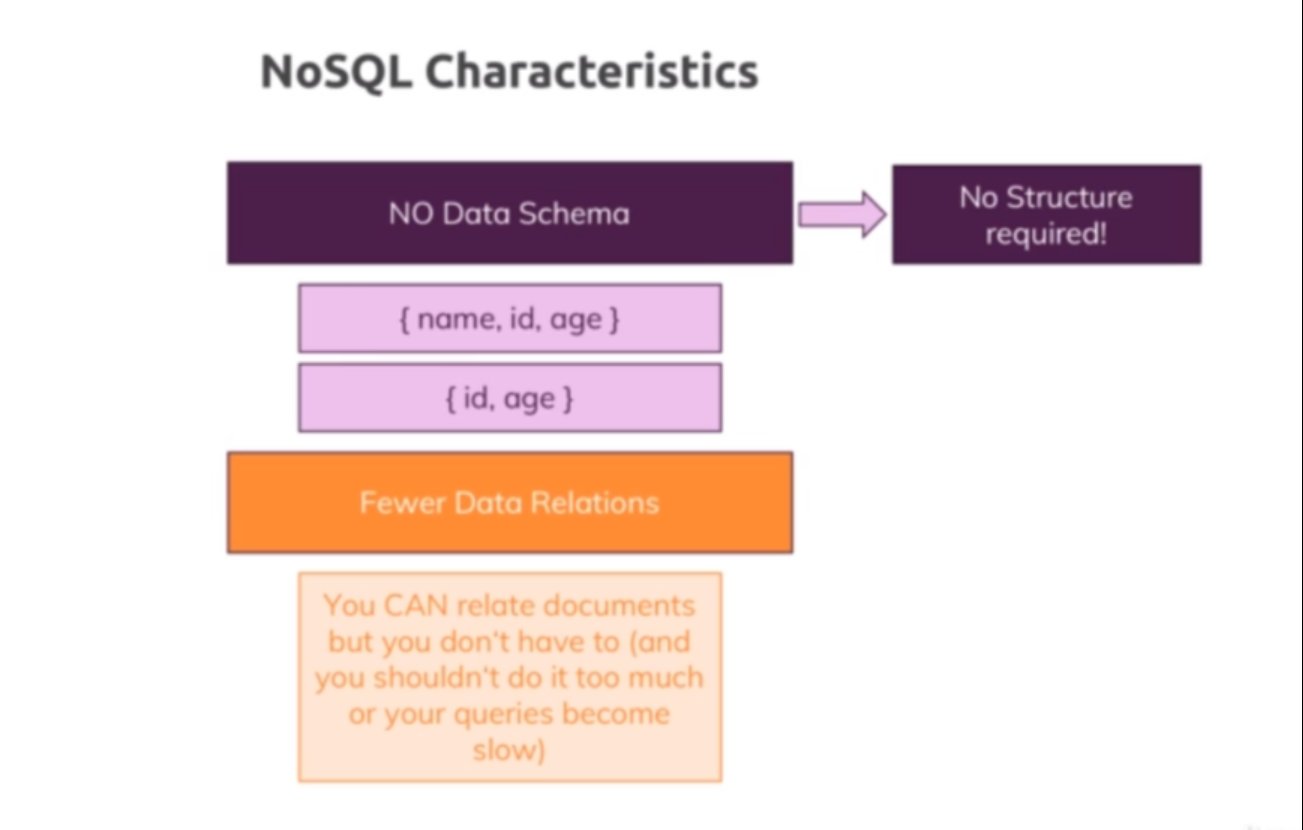
* Mongodb is both the name of the company which developed mongodb but also then of their most famous product, a database solution, a database engine you could say, a tool you can use to run very efficient NoSQL databases. The name stems from the word humongous because mongodb was built for one major 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

**Lecture 177**

**Relations in NoSQL**







* 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**

* We are using MongoDB Atlas – which is cloud based mongodb service
* Create new project, new cluster
* Create new user , add current ip to whitelist
* 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.
* 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.

**Lecture 179**

**Installing the MongoDB Driver**

* Npm install –save mongodb
* Refer code 01-using-the-database-connection.
* Util/database.js
* App.js

**Lecture 180**

**Creating the Database connection**

* Refer code 02-fetching-all-products
* Models/product.js.
* 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.

**Lecture 181**

**Finishing the Database connection**

* Refer code 02-fetching-all-products.
* Util/database.js.
* 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

**Lecture 182**

**Using the Database connection**

* Refer code 02-fetching-all-products.
* Models/product.js 🡪 save() method
* If collection doesnot exist, it will be created when you insert data for first time.

**Lecture 183**

**Creating Products**

* Refer code 02-fetching-all-products.
* Controllers/admin.js 🡪 postAddProduct method
* Every document inserted as a \_id property which is added by mongodb automatically if the user has explicicty specified that property

**Lecture 185**

**Fetching All Products**

* Refer code 02-fetching-all-products.
* Models/product.js 🡪 fetchAll method
* Controllers/shop.js 🡪 fetchAll method
* Find() method returns a cursor through which we can go through the results step by step.
* 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.

**Lecture 186**

**Fetching a single product**

* Refer code 03-fetch-a-single-product
* Controllers/shop.js 🡪 getProduct method
* Models/product.js 🡪 findById method

Next() method is used to get the next document , which in this case , is also the last document

* Shop/index.ejs,product-list.ejs 🡪 changing product.id to product.\_id.
* 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.

**Lecture 187**

**Making Edit and Delete buttons work again**

* Refer code 04-finishing-the-update-product-code.
* Contollers/admin.js 🡪 getProducts method
* Views/admin/products.ejs 🡪 changing id to \_id

**Lecture 188**

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

* Refer code 04-finishing-the-update-product-code.
* Controllers/admin.js 🡪getEditProduct and postEditProduct methods
* Views/admin/edit-product.ejs 🡪 id to \_id
* Models/product.js 🡪 adding \_id property, save() method

**Lecture 189**

**Finishing the update product code**

* Refer code 04-finishing-the-update-product-code.
* Controllers/admin.js 🡪 postEditProduct

**Lecture 190**

**One note about updating products**

* Modified code to convert the id to object id in product class constructor.

**Lecture 191**

**Deleting products**

* Refer code 05-deleting-products
* Controllers/admin.js 🡪 postDeleteProduct method
* Models/product.js 🡪 deleteById method

**Lecture 192**

**Fixing the add product functionality**

* Refer code 06-fixing-the-add-product-functionality
* new mongodb.ObjectId(id) -🡪 even if the argument ‘id’ is null or undefined , this expression will return a valid ObjectId.

**Lecture 193**

**Creating new users**

* Refer code 07-storing-the-user-in-the-database
* Models/user.js
* App.js

**Lecture 194**

**Storing the User in our Database**

* Refer code 07-storing-the-user-in-the-database
* Models/product.js 🡪 adding support for accepting user id
* Controllers/admin.js 🡪 postAddProduct

**Lecture 195**

**Working on Cart Items and Orders**

* Refer code 08-added-the-add-to-cart-functionality
* Models/cart.js 🡪 deleted
* Models/user.js 🡪 added cart attribute, addToCart method
* We are storing cart as embedded in User Model.

**Lecture 196**

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

* Refer code 08-added-the-add-to-cart-functionality
* App.js
* Controllers/shop.js 🡪 postCart method
* Models/user.js 🡪 addToCart method

**Lecture 197**

**Storing multiple products in the cart**

* Refer code 08-added-the-add-to-cart-functionality
* Models/user.js 🡪 addToCart method

**Lecture 198**

**Displaying the Cart Items**

* Refer code 09-displaying-the-cart-items
* Models/user.js 🡪 getCart method
* Controllers/shop.js 🡪 getCart method

**Lecture 199**

**Fixing a Bug**

* Refer code 09-displaying-the-cart-items
* Controllers/shop.js 🡪 postCart method

**Lecture 200**

**Deleting Cart Items**

* Refer code 10-deleting-cart-items.
* Models/user.js 🡪 deleteItemCart method

**Lecture 201**

**Adding an Order**

* Refer code 11-adding-relational-order-data
* Models/user.js 🡪 addOrder method
* Controllers/shop.js 🡪 postOrder method
* 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.

**Lecture 202**

**Adding Relational Order data**

* Refer code 11-adding-relational-order-data
* Models/user.js 🡪 getOrders method ,

addOrder method 🡪 storing userinfo in order

**Lecture 203**

**Getting Orders**

* Refer code 12-removing-deleted-items-from-the-cart
* Models/user.js 🡪 getOrders method
* We can retrieve documents based on the nested properties

**Lecture 204**

**Removing deleted items from cart**

* Refer code 12-removing-deleted-items-from-the-cart
* When a product is deleted by admin , it should also removed from the carts of users
* 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.
* We are ignoring that case

Two Adjustments (behind the scenes)

Behind the scenes, two files were deleted:

* order-item.js
* order.js

Why? We simply don't need them anymore, the way we now structured our models.