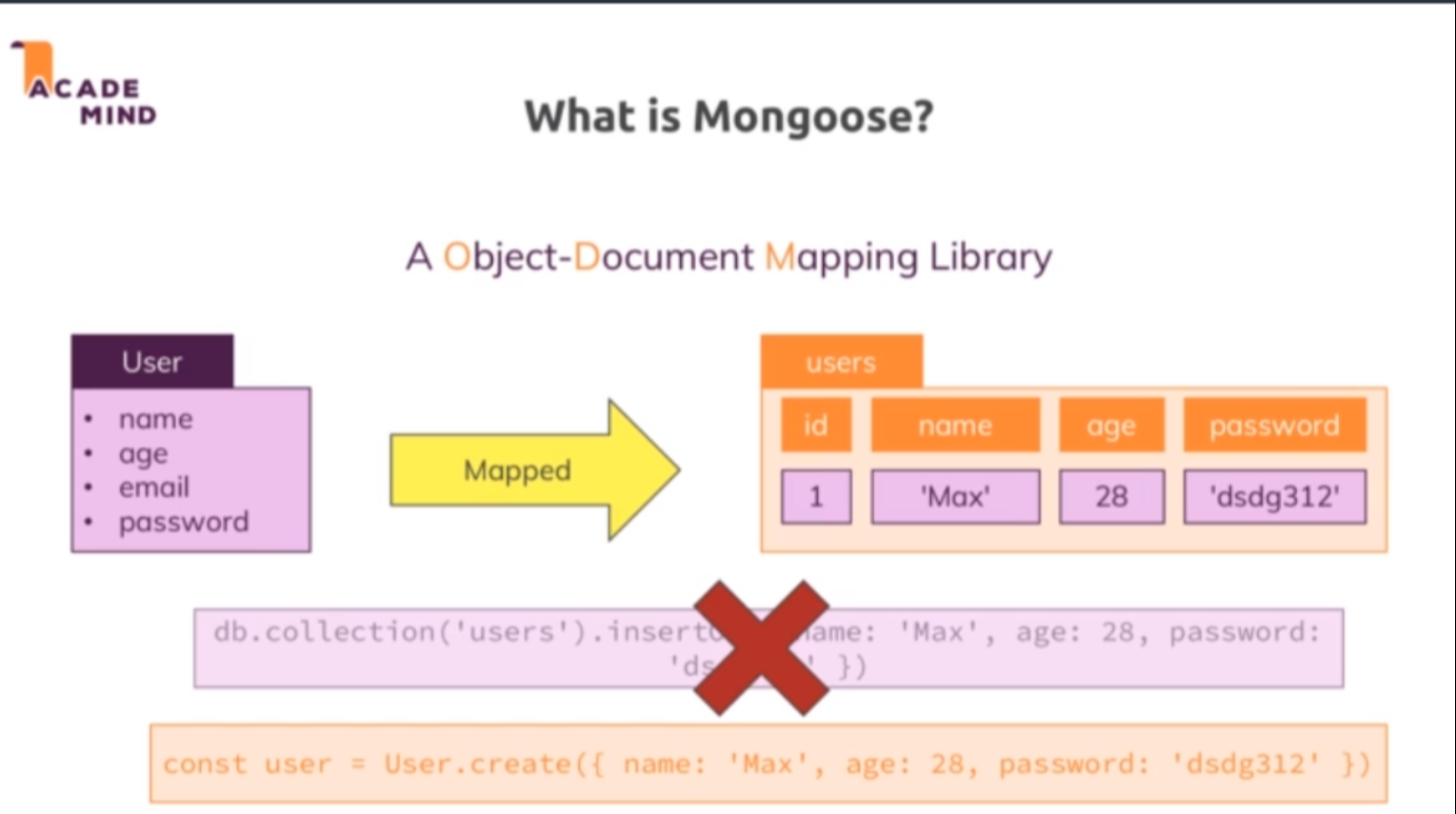
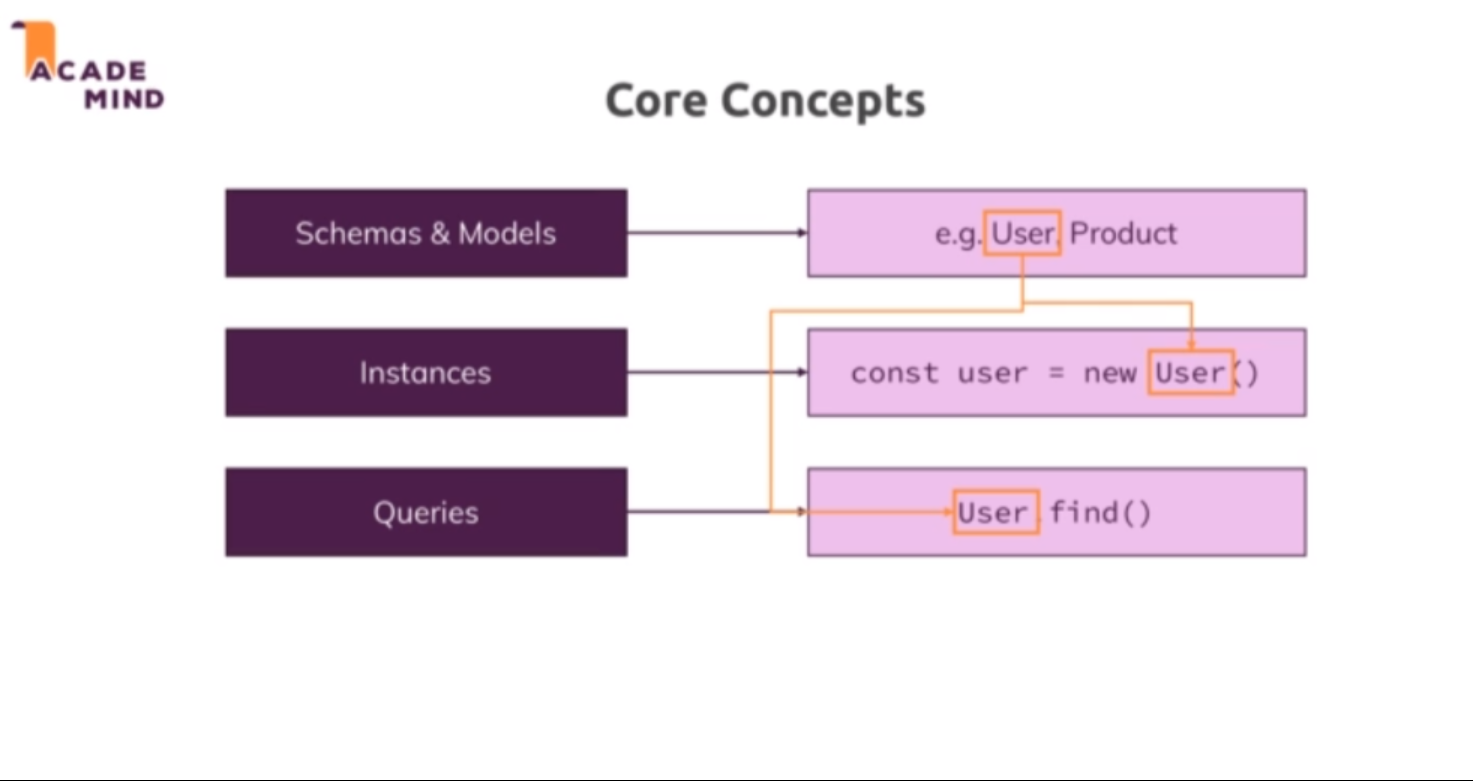
**Lecture 209**

**What is Mongoose?**





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

**Lecture 210**

**Connecting to the MongoDB Server with Mongoose**

* Refer code 01-connecting-to-the-mongodb-server
* App.js
* So with this, we already have everything in place we need to connect and mongoose as I mentioned will manage that one connection behind the scenes for us so that in other places where we start using mongoose from the mongoose package, we use that same connection we set up here, really convenient of course.

**Lecture 211**

**Creating the Product Schema**

* Refer code 02-saving-data-through-mongoose.
* Models/product.js
* mongodb is schemaless so why do we now start to create schemas? Well the idea simply is that whilst we have the flexibility of not being restricted to a specific schema, we often will have a certain structure in the data we work with and therefore mongoose wants to give you the advantage of focusing on just your data but for that, it needs to know how your data looks like and therefore we define such a schema for the structure our data will have. But important, we can still deviate from this by assigning a title like this, we could even work with a product and create a new one and save it to the database without setting a title because we still have the flexibility of not enforcing this, though what we can do is we can pass an object instead of just the type here as a value and then set a type property which could be set to string and then set required to true, this is basically a more complex way of configuring the value for this key and here we would say well the type of this is a string as before but it's also required.
* Please note that I don't add \_id here because this will still be added automatically as an object ID

**Lecture 212**

**Saving Data Through Mongoose**

* Refer code 02-saving-data-through-mongoose.
* Controllers/admin.js 🡪 postAddProduct method
* Save() method is provided by mongoose

**Lecture 213**

**Fetching All Products**

* Refer code 03-fetching-all-products.
* Find method in mongoose doesnot give a cursor , instead gives the result array itself
* It can be converted to cursor using Product.find().cursor()
* Controllers/shop.js 🡪 getProducts, getIndex method

**Lecture 214**

**Fetching a Single Product**

* Refer code 03-fetching-all-products.
* Controllers/shop.js 🡪 getProduct method.
* We can pass string id to findById method . Mongoose will automatically convert it to ObjectId

**Lecture 215**

**Updating Products**

* Refer code 04-deleting-products.
* Controllers/admin.js 🡪 postEditProduct method
* I will fetch a product and I'll fetch a product by ID with the prod ID, add a then block and in that then block, I know I have access to my product right, to the product which was fetched from the database. The cool thing is I can now move product save into my function here and call save on the product that was fetched from the database because thanks to mongoose, this will now not be a javascript object with the data but we will have a full mongoose object here with all the mongoose methods like save and if we call save on an existing object, it will not be saved as a new one but the changes will be saved, so it will automatically do an update behind the scenes. The only thing I need to do is I need to set my fields

**Lecture 216**

**Deleting Products**

* Refer code 04-deleting-products.

**Lecture 217**

**Adding and using a User Model**

* Refer code 05-adding-and-using-a-user-model.
* Models/user.js
* App.js
* findOne() method returns the first document if no arguments are passed.

**Lecture 218**

**Using relations in mongoose**

* Refer code 06-using-relations-in-mongoose
* Models/product.js, user.js
* Now obviously every product should be assigned to a user, so first of all we need to change our product schema a little bit. A product should also have a user ID field let's say, just as we had it before in the last module. Now a user ID field is of type and now which type is this? Well it will be a reference to a user, so this will actually be of type schema types objectID and now we can set something special here, we can add a special ref configuration and ref takes a string where we tell mongoose hey which other mongoose model is actually related to the data in that field. We know that we will store a user ID here but just because the type is objectid, this is not obvious, this could be any object ID of any object. So I will add user here and you use the name of your model to which you want to relate this, so since our model here is named user, I will name it user here, so I refer to my user model here and with that I got a relation set up.
* This also means that in my user model where I do store the product ID, I can also add a reference here and refer to product because I know that for every user in the cart items, I will store products where I refer to some ID and that ID happens to refer to a product stored or defined through the product model.
* Controllers/admin.js 🡪 postAddProduct method 🡪 adding user id while creating product
* When we add the user id while saving , we can specify either req.user.\_id or req.user , since req.user is a mongoose object and mongoose will automatically take the id from that.

**Lecture 219**

**One important thing about fetching relations**

* Refer code 06-using-relations-in-mongoose
* Controllers/admin.js 🡪 getProducts method

While fetching documents which has a relation to documents from other collections , we can call populate() method after find() method to populate those related fields with the document from other collections

e.g Product.find() -> in this case the fetched document only have the userId, which is a string

Product.find().populate(‘userId’) -> in this case the userId will be the entire document corresponding to that user.

* We can select the fields to be included in the result using the select() method following find() method.

Product.find().select(‘title price -\_id’) 🡪 only title and price wilil be included. Specifying ‘-’ before the field will exclude that field.

This can be done while using populate also by specifying as a second argument

Product.find().populate(‘userId’,’name’) 🡪 the user document will only have the name field

**Lecture 220**

**Working on the Shopping cart**

* Refer code 07-working-on-the-shopping-cart
* Models/user.js 🡪 adding addToCart method to user schema

**Lecture 221**

**Loading the cart**

* Refer code 08-loading-the-cart
* Controllers/shop.js 🡪 getCart method
* Populate method doesnot return promise. We have to call execPopulate() following populate()
* Views/shop/cart.ejs

**Lecture 222**

**Deleting Cart Items**

* Code 09-getting-and-displaying-orders
* Controllers/shop.js 🡪 postCartDeleteProduct method
* Models/user.js 🡪 adding removeFromCart method

**Lecture 223**

**Creating and Getting Orders**

* Code 09-getting-and-displaying-orders
* Controllers/shop.js 🡪 postOrder method

**Lecture 224**

**Storing All Order related data**

* Code 09-getting-and-displaying-orders
* Controllers/shop.js 🡪 postOrder method

**Lecture 225**

**Clearing the cart after storing an order**

* Code 09-getting-and-displaying-orders
* Models/user.js 🡪 adding clearCart method
* Controllers/shop.js 🡪 postOrder method

**Lecture 226**

**Getting and displaying the orders**

* Code 09-getting-and-displaying-orders
* Controllers/shop.js 🡪 getOrders method
* Views/shop/orders.ejs