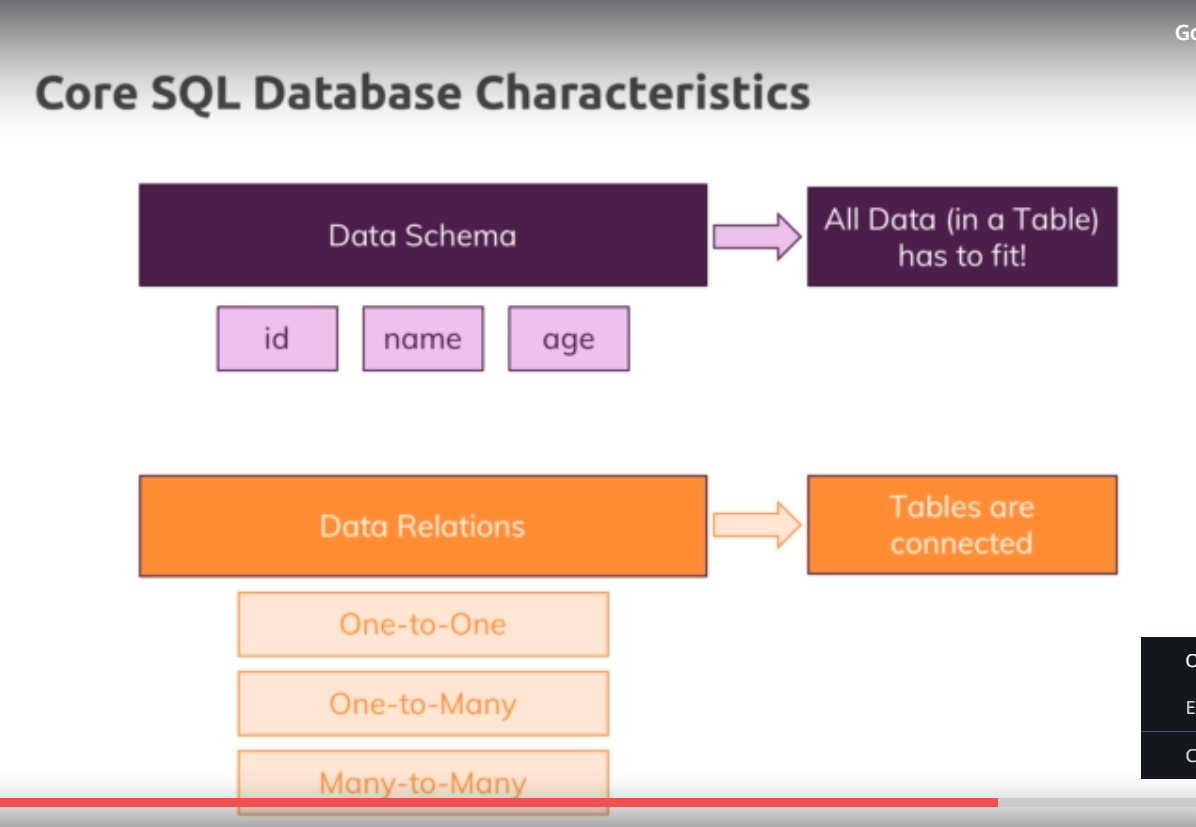
132)Choosing Database

Characteristic of sql, it has strong schema, for each table we define which columns this table will have, which data type can that column can have. So we have this strongly and strictly defined schema and **all data in table has to fit that schema** , this is really important. So this schema thing, this definition of how data has to look like is one core thing in SQL database.

Another core characteristic of sql is that we also have realtons between tables. These can be of 3 types. So tables are connected that is another important thing.

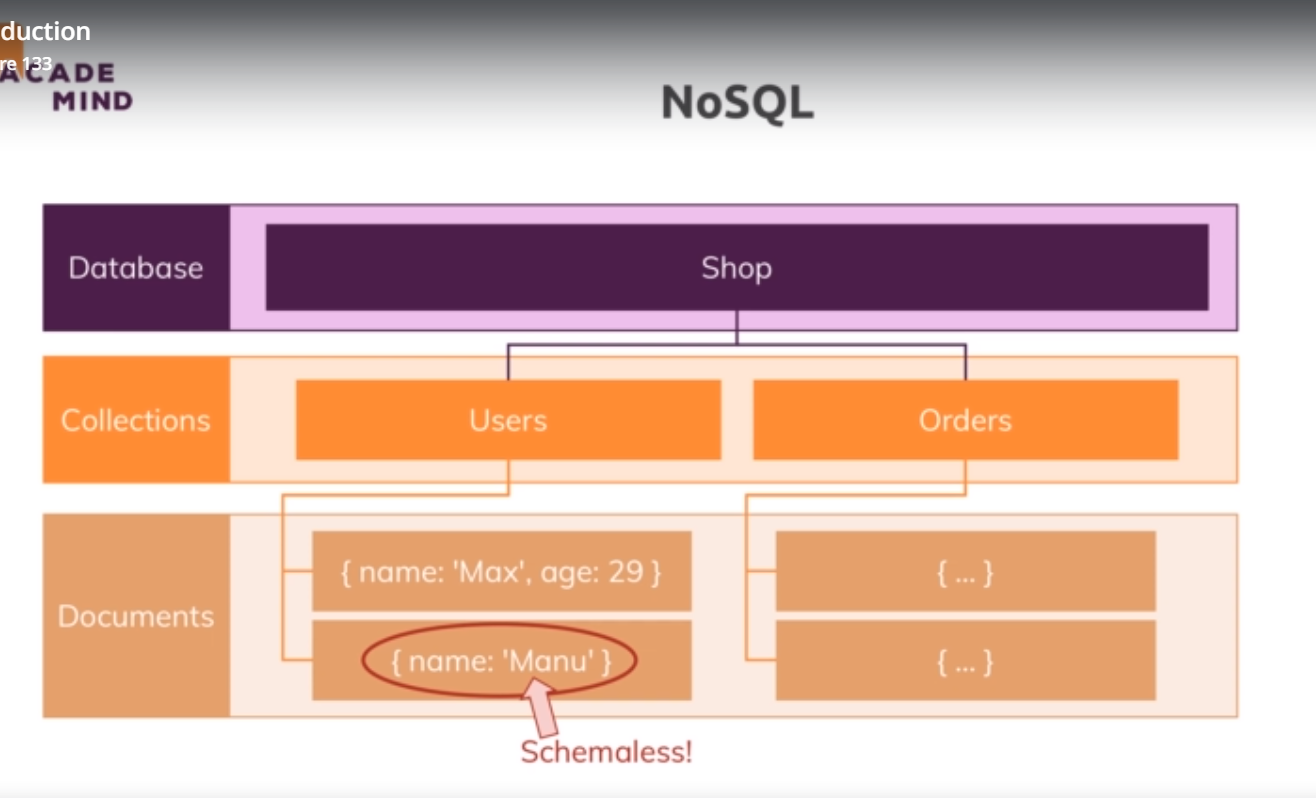


133)NoSql Intro

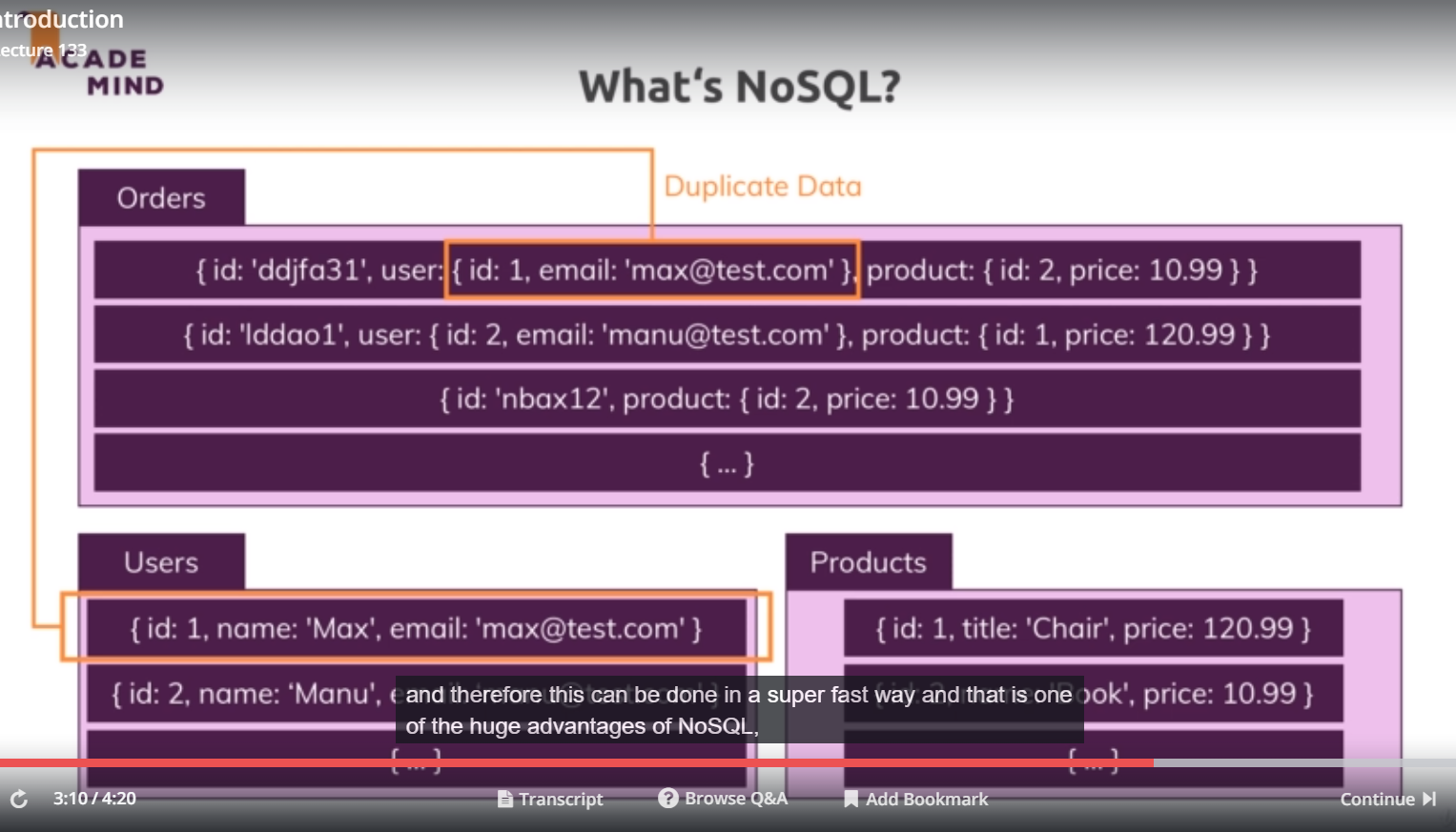
Tables = collections

Records = documents

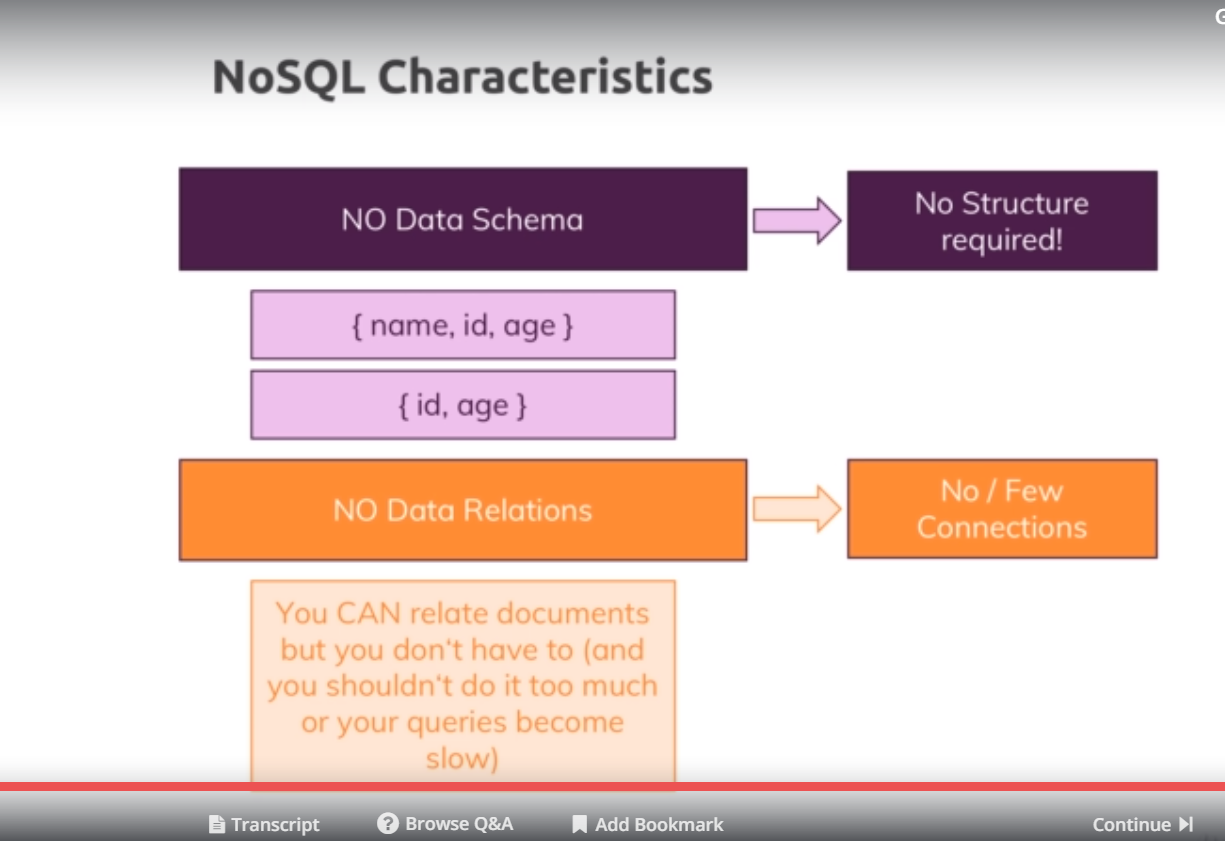
No sql does not have strict schema. Here we got 2 documents in same collection. But second one does not have age and that is perfectly fine in no sql. You can store multiple doucments with different structures in same collection.



Other thing is nosql word we got no real relations. Instead we go for duplicate data. now that means that we have an orders collections here, we have a nested document, user which is also stored as separate document with more details may be, in users collections and we dnt connect that through some id or behind the scenes setup relation isnteda of dulplicate data, to be precise, the data thatw eneed in orders collections here. this also means that if that data changes, we have to update it multiple places,if all places need latest data. but this can be ok, because on other hand it gives us huge sdvntage that if we ever retrieve data, we dnt have to join multiple tables togather, which can lead to very long and difficult code and which can also impact performance. So instead of making quersues in other tables, herew e got all data that we need in orders collection. So this can be done in super fast way and it is one of huge advantages of no sql, it can be very fast and efficient.



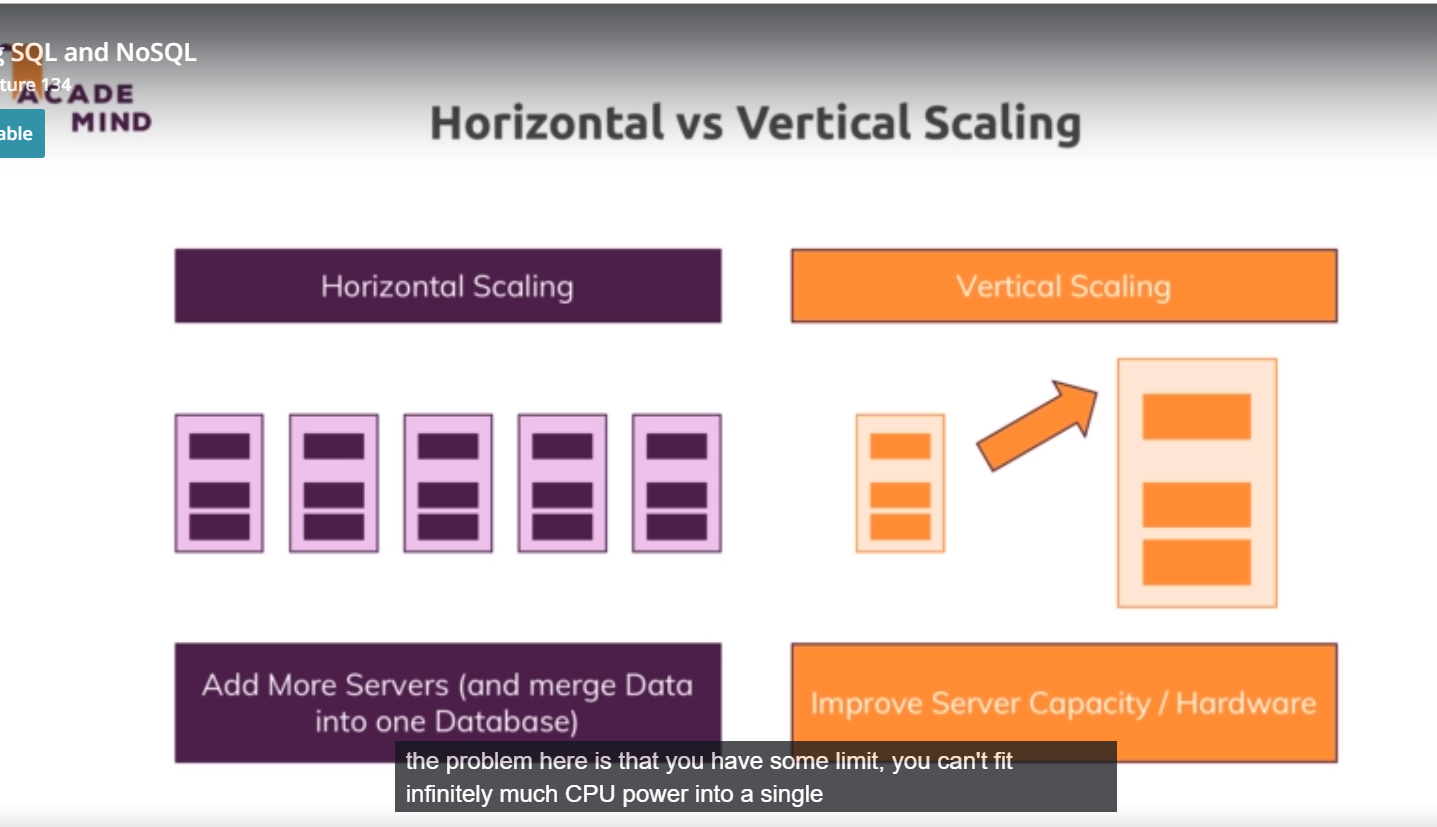
So characteristic of no sql is-



There is also difference in scalability.

134)Comparing sql and no sql

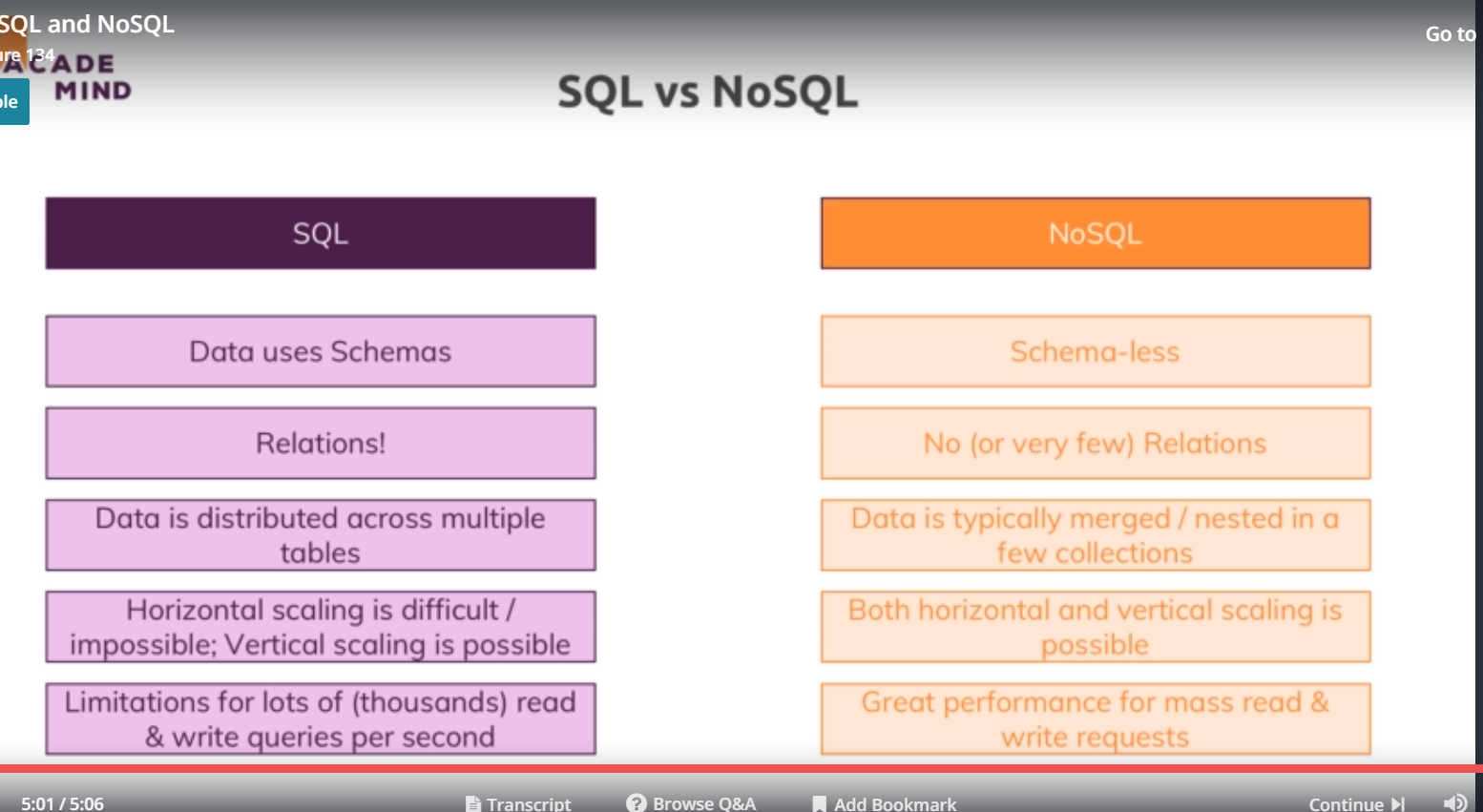
Hese are 2 approchs thatw ecan use to scale our db.



Horizontal scaling means that we add more servers. Advantage is that we can do it infinity, we can always buy more servers, be that on cloud provider or in data center and connect them to our database and split our data across all these servers. Of course this emans thtw e also need some process that runs queries on them and merges them togather inteligentally. So this is general something which is not that easy to do but this of course a good way of scaling.

Vertical scalling means that we make our eiting server stronger by adding more cpi or something like that, especially with cloud providers, this is very easy, you just choose another option from dropdown. Te problem is here is that you have some limit,. You cnt fit infinetly much cpu into single machine.

Lets compare 2 db’s regarding scaling and other things.



In sql because of below reasons, horizontal scaling is not possible. So this is a problem, possibly if we have multiple or thousands of read and write queries per second, then may be ouer sql database especially if we if we do very complex joins between related tables can reach limits or cannot be the best choice.

On other hand scaling is easier in nosql. So nosql can be very performant in an application with high throughput.

Now that makes sql look very bad but the full truth is that it always depends on kind of data, you are storing. If you are storing where relations are really important and where you want to have split up across the tables and where you want to have strong schemas, SQL can be perfect and also not every part of your data is accessed multiple times per second. you can have parts of your applications where you manage general data,lets say user data which does not change that often and therefore , SQL might be very good there. Other parts of applications, lets say orders or shopping carts that do change frequently could be stored with NoSQL and there, relations might not be that important because you can always put all the information that belongs to shopping cart or to an order in one single document and even if you do store some user data there, you might not need to touch that document just because user changed photo because you probably did’nt store that along with order anyways.

Here I will tech you to use both of them.

135)Setting up MySQL

Go to mysql website, there go to download, then to community edition. Now we need

1)MySQL Community addition

2)MySql work bench

On windows you can use combine installer. On window, choose **mysql on windows** , whch allows you to setup everything in one go.

Now during installation, in authenticate=ion methid, select **use legacy authentication method** , this may sound insecure, but is completely valid. The newer version is simply not supported by the node sequel package we are using yet.

136)Connecting to database

We created a new file database.js in utils folder. install tthis package-

**npm install –-save mysql2**

it allows us to write sql code and execute sql code in node and interact with a database.

Now we need to connect to our database our inside our app. here we will write code to connect to db and also it will give us connection object that we can use to write queries.

Now there are 2 ways of connecting. One is tht we setup one connection which we can then use to run queries and we should close the connection when we are done with query and the downside is that we have to rexecute same code to create connection for every query. There will be lot of queries. So creating connections all the time quickly becomes very inefficient both in our code and also regarding the connection to the database which is established. So the better way is to so create so called connection pool. You can leran all about this package in official docs.

So we use createPool method on this packagewhich gives us a pool. We also have createConnection methofd which will give us single connection. But here we want pool of connections. This pool will allow us to always reach out to it whenever we have query to run and then we get a new connection from that pool which manages multiple conncetions, so that we can run multiple queries simultaneously because each query needs its own connection and once query is done, the connection will be handed back to into the pool and it is available again for new query and the pool can be finished when our application shuts down.

To CrettePool we ass js object, which has details of db we want to connect to.

Then we export this pool, we export it in special way actually, I will call promise on pool because this will allow us to use promise when working with these connections.

We can import this pool and execute couple of methods on it. One such method is execute, which allow us to execute queries. We also got query methods for it but execute is a bit safer.

We also have end metod, whih ends the connection when applications shuts down.

Database.js-

const mysql = require('mysql2');

const pool = mysql.createPool({

host: 'localhost',

user: 'root',

database: 'node-complete',

password: 'sumeet'

});

module.exports = pool;

now create a table producst and enter one row in it in db using workbench. Now lets execute a query in app.js-

const db = require('./util/database');

db.execute('SELECT \* FROM products').then();

we are getting back promises as a result of these queries, this is because we used .promise I database.js

138)Retreiving Data

We can also use callbacks with this packages. We just need to pass a anonymous function as second argument to execute fucntiion. By using promises we do this-

db.execute('SELECT \* FROM products')

.then((result) => console.log(result))

.catch(err => console.log(err));

Now lets see what we got. We get object on console, in this object we also see the data of row. we got back an array, a nested array infact. First array has row data and next array then

Has some metadata about the table. Sow e have 2 arrays inside results. Letslog them separately.

db.execute('SELECT \* FROM products')

.then((result) => {

console.log(result[0]);

console.log(result[1]);

})

.catch(err => console.log(err));

So row data is in first console. Now lets see how we can adjust our model to work with database instead of file system.

139)Fetching products

Models/product.js-

static fetchAll(cb) {

// getProductsFromFile(cb);

return db.execute('SELECT \* FROM products');

}

controller/shop.js-

exports.getProducts = (req, res, next) => {

Product.fetchAll()

.then(([rows, fieldData]) => {

res.render('shop/product-list', {

prods: rows,

pageTitle: 'All Products',

path: '/products'

});

})

.catch(err => console.log(err));

Here we used array destructuring to extract value from incoming from array.

141)Inserting data into database

Now to safely inject the values and not face the issue of SQL injection which is attack pattern where users can inset special data into your input fields in your web page tht runs SQL queries,we should use approach where we sue ? , one for each values. Then we pass argument to execute method, values feo second argument will be inserted in place of question marks.

Here mySQl pattern parses this second argument to look for any sql commands and removes them if found any.

Models/product.js-

save() {

return db.execute(

'INSERT INTO products (title, price, imageUrl, description) VALUES (?,?,?,?)',

[this.title,this.price,this.mageUrl, this.description]

);

}

Controllers/product.js-

Controller/admin.js-

exports.postAddProduct = (req, res, next) => {

const title = req.body.title;

const imageUrl = req.body.imageUrl;

const price = req.body.price;

const description = req.body.description;

const product = new Product(null,title, imageUrl, description, price);

product

.save()

.then(() => {

res.redirect('/');

})

.catch(err => console.log(err));

};

Here we redirect only when data has been inserted.

142)Fetching a Single Product with the ‘where’ Condition

Model/product.js-

static findById(id) {

return db.execute('SELECT \* FROM products WHERE products.id = ?', [id]);

}

Controllers/shop.js-

exports.getProduct = (req, res, next) => {

const prodId = req.params.productId;

Product.findById(prodId)

.then(([product]) =>{

res.render('shop/product-detail', { product: product[0], pageTitle: product.title, path: '/products' });

})

.catch(err => console.log(err));

};

Here products is array of rows, althrogh we have only one row there. We still need to pick up first element.

143)Wrap up

You can have your application do all things by these queries. But as your app grows these queries will grow complex. If you are sql master, you can use only uries in your app.

But for rest of us there is a way easier approach to that which allows us to not to write all these queries, we dnt have to write any native sql code but instead only work with native js objects that gives us certain functionalities that we need for connecting them, deleting them, adding them and so on. So in next module we will find a easier approach.