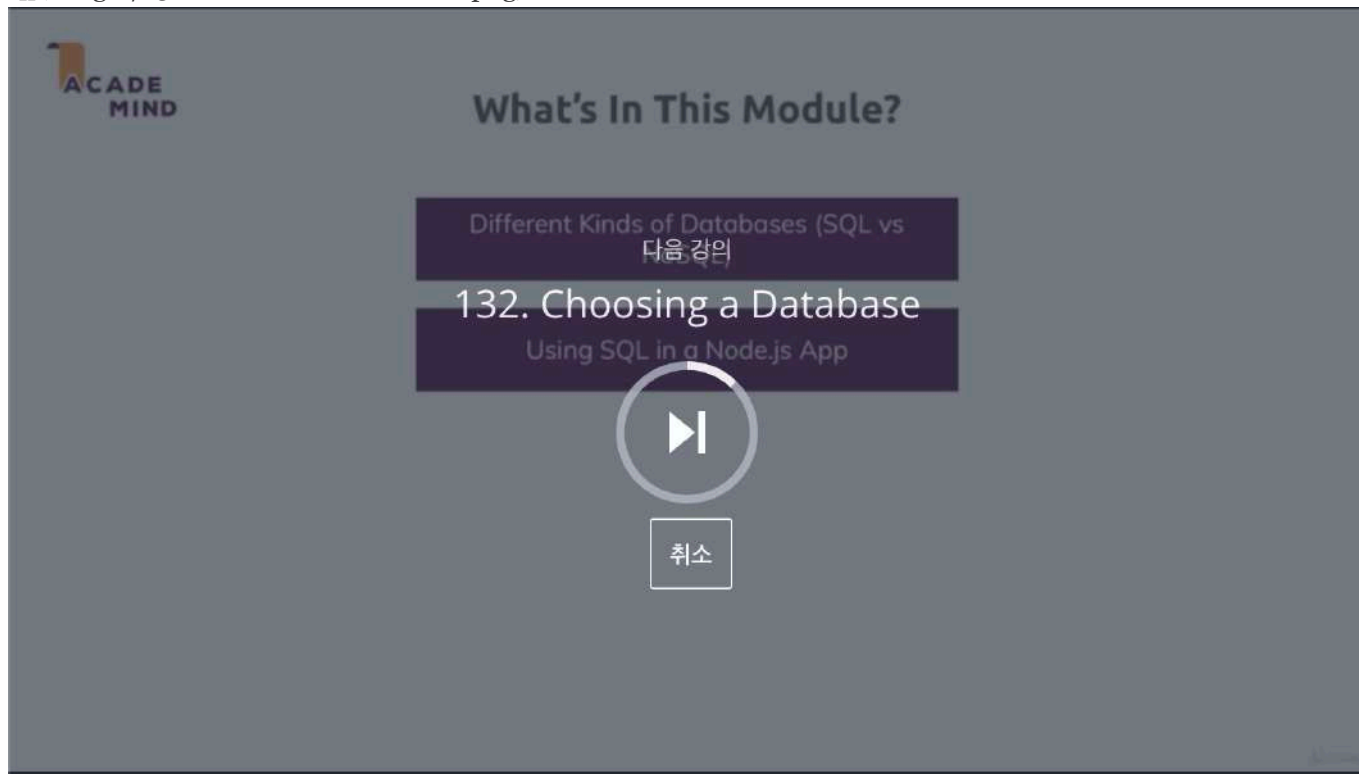


## 10. SQL Introduction

### \* Chapter 131: Module Introduction



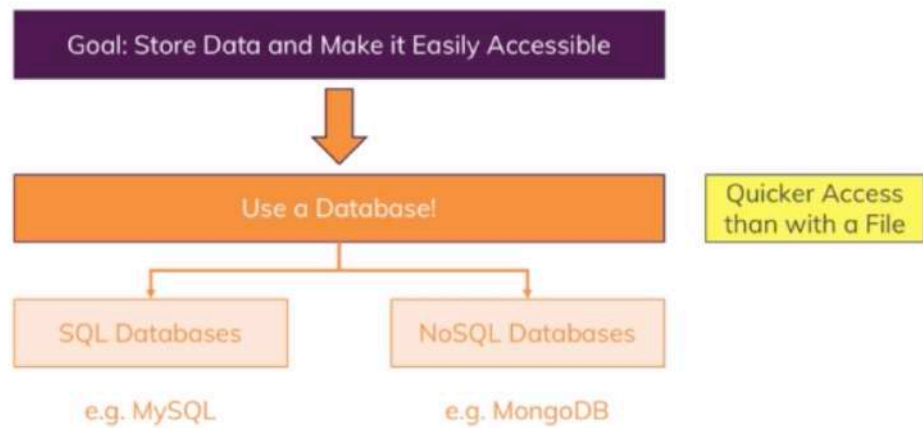


### \* Chapter 132: Choosing A Database





## SQL vs NoSQL



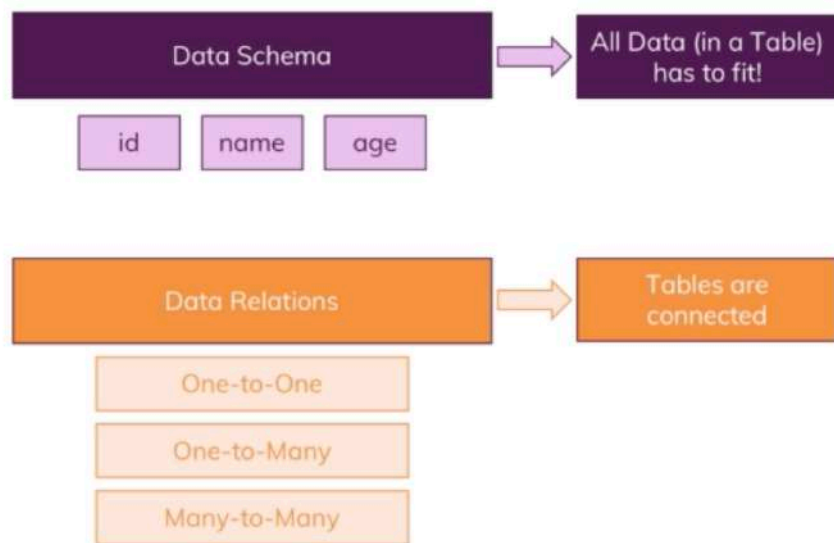
## What's SQL?



- SQL database thinks in so-called tables. so we might have a users, a product and let's say an orders table and in each table, you have so-called fields or columns. for example, a user could be defined by having an ID, email, a name and product could have an ID, title, price and a description.
- now we fill in data for these fields, so-called records. so basically the rows in our tables. for example, we got a couple of users with their data and we get a couple of products too.
- SQL-based database also have one important thing. they allow you to relate different tables. for example, an order could simply be described as a connection of a user and a product. because a user might order a couple of different products and a product might be ordered by a couple of different users.



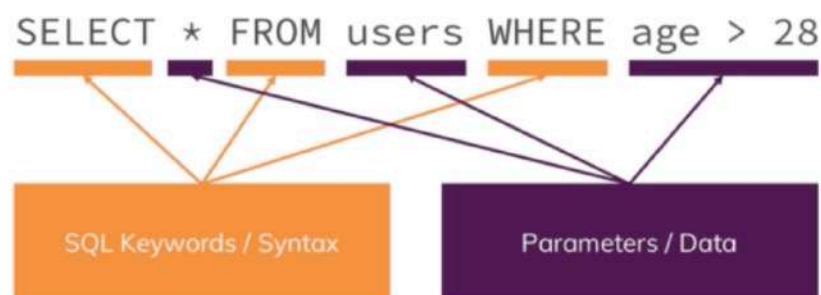
## Core SQL Database Characteristics



132

- the core SQL database characteristics are that we have a strong data schema. so that for each table, we clearly define how the data in there should look like. so which fields do we have, which type of data does each field store. is it a number? or string or text and so on.
  - so the schema, this definition of how the data has to look like is one core thing in a SQL database.
  - we relate our different table with 3 important kinds of relations. one to one, one to many, or many to many. this means that we can have 2 tables where each record fits one other record, a record might fit multiple other records or multiple records in table A can fit multiple records in table B.
- 

## SQL Queries

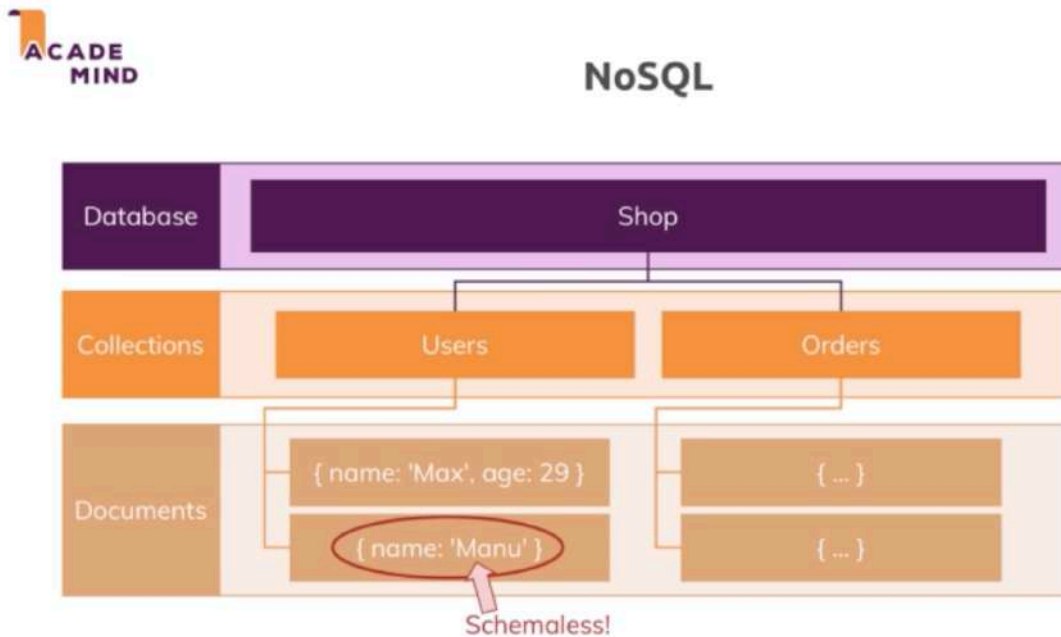


133

- SQL stands for 'Structured Query Language'. Queries are commands we use to interact with the database.
- this command would be a command that selects all users, so all entries, all records in the users table where the age is greater than 28. so this is so-called 'query'

# \* Chapter 133: NoSQL Introduction

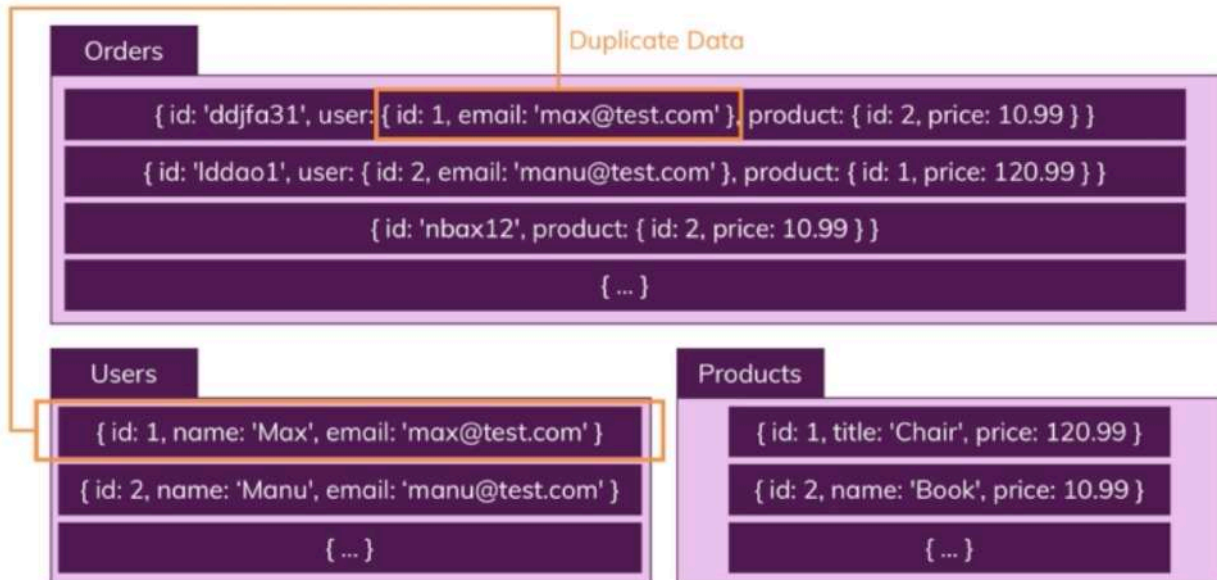




- NoSQL means that it doesn't follow the approach SQL follows. it also uses a different query language. but instead of having schemas and relations, NoSQL has other focuses or other strength.
- in NoSQL, we can still have a database and we can give this database a name, shop. that's the same for SQL. there we also have database
- in SQL, we had tables, users an orders and also products. these are just examples here.
- in NoSQL, tables are called 'collections' but you can think of them as tables, so as the table equivalent. but we call them collections in the NoSQL world. in collection, we don't find records but so-called documents which look like this. documents are very close to how we describe data in javascript.
- that are the documents in our collections and what you can already see in the users collection example is that NoSQL doesn't have a strict schema. we got 2 documents in the same collection. but the second document `{name: 'Manu'}` doesn't have a age and that is perfectly fine in NoSQL. you can store multiple documents with different structures in the same collection. you still try to have kind of a similar structure, but it's also not uncommon for some application that you don't always have exactly the same fields available for the data you're storing in the database and that is ok in NoSQL. so you can store documents which are generally equal but where some fields might differ.

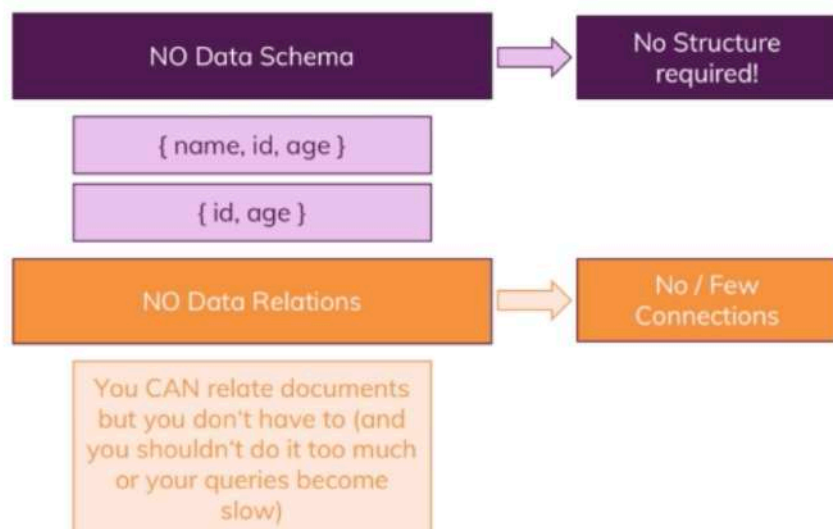


## What's NoSQL?



- One other thing is that in the NoSQL world, we got no real relations, instead we go for duplicated data. that means that if we have an orders collection here, we have a nested document, the user which also is stored as a separate document with more details maybe in the users collection. we don't connect that through some ID or behind the scenes setup relation.
  - instead we duplicate data we need in the orders collection. that means that if that data changes, we have to update it in multiple places. if all these places need the latest update or the latest data change but that can be OK because on the other hand, this gives us the huge advantage that if we ever retrieve data, we don't have to join multiple tables together which can lead to very long and difficult code and which can also impact performance, instead we can read the data from the orders collection and we probably got all the data we need to display on the orders page without having to reach out to other collections.
  - and therefore this can be done in a super fast way and that is one of the huge advantages of NoSQL.
- 

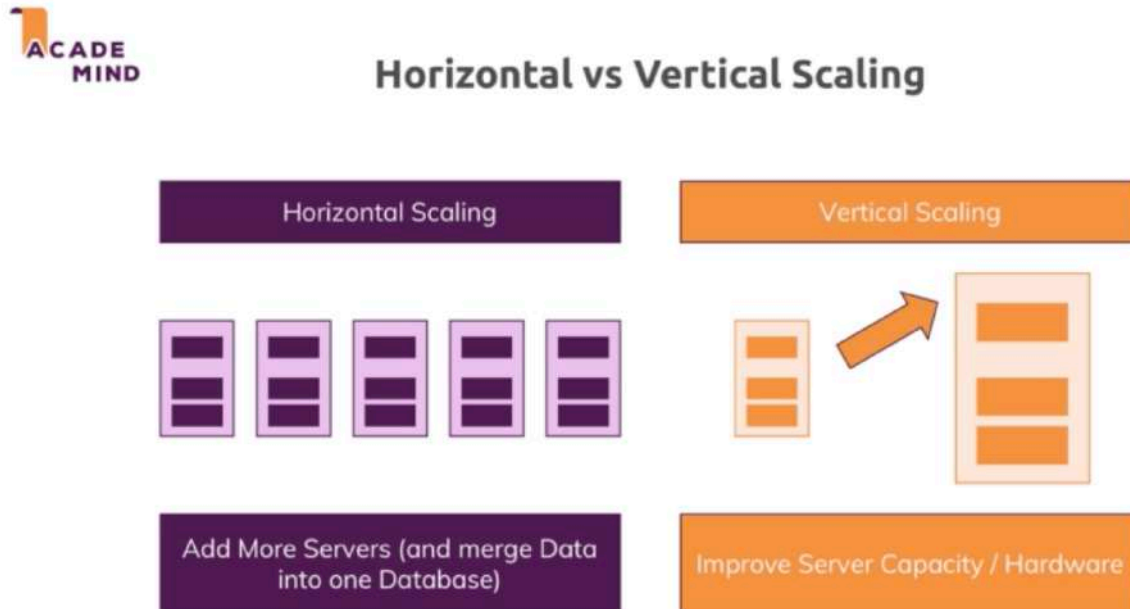
## NoSQL Characteristics



- we have generally no data relations. now we can relate document in some way.
- we can differentiate between horizontal and vertical scaling.

## \* Chapter 134: Comparing SQL And NoSQL





- in horizontal scaling, we simply add more servers. the advantage is that we can do this infinitely. we can always buy new servers, be that on a cloud provider or in our own data center and connect them to our database and split our data across all these servers. this also means that we also need some process that runs queries on all of them and merges them together. so this is generally something which is not that easy to do but this is a good way of scaling.

- Vertical Scaling means that we make our existing server stronger by adding more CPU or memory, especially with cloud provider, this is very easy, you choose another option from the dropdown. the problem is that you have some limit, you can't fit infinitely much CPU power into a single machine.



## SQL vs NoSQL

SQL	NoSQL
Data uses Schemas	Schema-less
Relations!	No (or very few) Relations
Data is distributed across multiple tables	Data is typically merged / nested in a few collections
Horizontal scaling is difficult / impossible; Vertical scaling is possible	Both horizontal and vertical scaling is possible
Limitations for lots of (thousands) read & write queries per second	Great performance for mass read & write requests

## \* Chapter 135: Setting Up MySQL







The world's most popular open source database


MySQL

MYSQL.COM DOWNLOADS DOCUMENTATION DEVELOPER ZONE

Contact MySQL | Login | Register

f t in g

Products Cloud Services Partners Customers Why MySQL? News & Events How to Buy



**New! MySQL 8.0**

MySQL Document Store

SQL and NoSQL!


LEARN MORE



**MySQL Enterprise Edition**

The most comprehensive set of advanced features, management tools and technical support to achieve the highest levels of MySQL scalability, security, reliability, and uptime.

[Learn More »](#)



**Oracle MySQL Cloud Service**

Built on MySQL Enterprise Edition and powered by the Oracle Cloud, Oracle MySQL Cloud Service provides a simple, automated, integrated and enterprise ready MySQL cloud service, enabling organizations to increase business agility and reduce costs.

[Learn More »](#)



## Contact Sales

USA: +1-866-221-0634  
Canada: +1-866-221-0634

Germany: +49 89 143 01280  
France: +33 1 57 60 83 57  
Italy: +39 02 349 59 120  
UK: +44 207 553 8447

Japan: 0120-065556  
China: 10800-811-0823  
India: 0008001005870

[More Countries »](#)

[Contact Us Online »](#)



# MySQL Downloads

## Oracle MySQL Cloud Service (commercial)

Oracle MySQL Cloud Service is built on MySQL Enterprise Edition and powered by Oracle Cloud, providing an enterprise-grade MySQL database service.

[Learn More »](#)

## MySQL Enterprise Edition (commercial)

MySQL Enterprise Edition includes the most comprehensive set of advanced features and management tools for MySQL.

- MySQL Database
- MySQL Storage Engines (InnoDB, MyISAM, etc.)
- MySQL Connectors (JDBC, ODBC, .Net, etc.)
- MySQL Replication
- MySQL Partitioning
- MySQL Utilities
- MySQL Workbench
- MySQL Enterprise Backup
- MySQL Enterprise Monitor
- MySQL Enterprise HA
- MySQL Enterprise Security
- MySQL Enterprise Transparent Data Encryption (TDE)
- MySQL Enterprise Firewall
- MySQL Enterprise Encryption
- MySQL Enterprise Audit

[Learn More »](#)

[Customer Download »](#) (Select Patches & Updates Tab, Product Search)

[Trial Download »](#) (Note - Select Product Pack: MySQL Database)

[Learn More »](#)

- MySQL Enterprise Encryption
- MySQL Enterprise Audit

[Learn More »](#)

[Customer Download »](#) (Select Patches & Updates Tab, Product Search)

[Trial Download »](#) (Note - Select Product Pack: MySQL Database)

## MySQL Cluster CGE (commercial)

MySQL Cluster is a real-time open source transactional database designed for fast, always-on access to data under high throughput conditions.

- MySQL Cluster
- MySQL Cluster Manager
- Plus, everything in MySQL Enterprise Edition

[Learn More »](#)

[Customer Download »](#) (Select Patches & Updates Tab, Product Search)

[Trial Download »](#) (Note - Select Product Pack: MySQL Database)

## MySQL Community Edition (GPL)

[Community \(GPL\) Downloads »](#)



### Contact MySQL Sales

USA/Canada: +1-866-221-0634 ([More Countries »](#))



### PRODUCTS

Oracle MySQL Cloud Service  
MySQL Enterprise Edition  
MySQL Standard Edition  
MySQL Classic Edition

### SERVICES

Training  
Certification  
Consulting  
Support

### DOWNLOADS

MySQL Community Server  
MySQL NDB Cluster  
MySQL Shell  
MySQL Router

### DOCUMENTATION


MySQL Reference Manual  
MySQL Workbench  
MySQL NDB Cluster  
MySQL Connectors

### ABOUT MYSQL

Contact Us  
How to Buy  
Partners  
Job Opportunities

- we need MySQL Community Edition.  
  



The world's most popular open source database

MySQL.COM **DOWNLOADS** DOCUMENTATION DEVELOPER ZONE

Contact MySQL | Login | Register

f t in S+

Enterprise **Community** Yum Repository APT Repository SUSE Repository Windows Archives

- MySQL on Windows
- MySQL Yum Repository
- MySQL APT Repository
- MySQL SUSE Repository
- MySQL Community Server
- MySQL Cluster
- MySQL Router
- MySQL Shell
- MySQL Workbench
- MySQL Connectors
- Other Downloads

## MySQL Community Downloads

### MySQL Community Server (GPL)

(Current Generally Available Release: 8.0.12)

MySQL Community Server is the world's most popular open source database.

DOWNLOAD

### MySQL Cluster (GPL)

(Current Generally Available Release: 7.6.7)

MySQL Cluster is a real-time, open source transactional database.

DOWNLOAD

### MySQL Router (GPL)

(Current Generally Available Release: 8.0.12)

MySQL Router is lightweight middleware that provides transparent routing between your application and any backend MySQL Servers.

DOWNLOAD

### MySQL Enterprise Edition (commercial)

MySQL Enterprise Edition includes the most comprehensive set of advanced features and management tools for MySQL.

Learn More »

Download from Oracle eDelivery »

### MySQL Cluster CGE (commercial)

MySQL Cluster is a real-time, transactional database designed for fast, always-on access to data under high throughput conditions. Plus, it includes everything in MySQL Enterprise Edition.

Learn More »

Download from Oracle eDelivery »

- MySQL Router
- MySQL Shell
- MySQL Workbench
- MySQL Connectors
- Other Downloads

### MySQL Cluster (GPL)

(Current Generally Available Release: 7.6.7)

MySQL Cluster is a real-time, open source transactional database.

DOWNLOAD

### MySQL Router (GPL)

(Current Generally Available Release: 8.0.12)

MySQL Router is lightweight middleware that provides transparent routing between your application and any backend MySQL Servers.

DOWNLOAD

### New! MySQL Shell (GPL)

(Current Generally Available Release: 8.0.12)

The MySQL Shell is an interactive Javascript, Python, or SQL interface supporting development and administration for the MySQL Server and is a component of the MySQL Server.

DOWNLOAD

### MySQL Workbench (GPL)

(Current Generally Available Release: 8.0.12)

MySQL Workbench is a next-generation visual database design application that can be used to efficiently design, manage and document database schemata. It is available as both, open source and commercial editions.

DOWNLOAD

### MySQL Connectors

MySQL offers standard database driver connectivity for using MySQL with applications and tools that are compatible with industry standards ODBC and JDBC.

DOWNLOAD

### New Releases

- Connector/Python 2.1 (2.1.8 GA)
- Connector/J 5.1 (5.1.47 GA)
- Connector/NET 6.10 (6.10.8 GA)
- Connector/ODBC 5.3 (5.3.11 GA)
- Connector/C++ 8.0 (8.0.12 GA)

- and we need 'MySQL Community Server' and 'MySQL Workbench'









On Windows, you can use the combined installer. On macOS or Linux, you need to install both tools separately.

MySQL Workbench is a next-generation visual database design application that can be used to efficiently design, manage and document database schemata. It is available as both, open source and commercial editions.

(5.3.11 GA)

Connector/C++ 8.0  
(8.0.12 GA)[DOWNLOAD](#)

### MySQL Connectors

MySQL offers standard database driver connectivity for using MySQL with applications and tools that are compatible with industry standards ODBC and JDBC.

[DOWNLOAD](#)

### MySQL on Windows (Installer & Tools)

(Current Generally Available Release: 8.0.12)

MySQL provides you with a suite of tools for developing and managing MySQL-based business critical applications on Windows.

[DOWNLOAD](#)

### MySQL Yum Repository

MySQL provides a YUM software repository to simplify installing and updating MySQL products on a variety of Linux operating systems.

[DOWNLOAD](#)

### MySQL APT Repository

MySQL provides an APT-style software repository for installing the MySQL server, client, and other



Please report any bugs or inconsistencies you observe to our [Bugs Database](#).  
Thank you for your support!

#### Generally Available (GA) Releases

### MySQL Community Server 8.0.12

Select Operating System:

macOS

[Looking for previous GA versions?](#)

ⓘ Packages for High Sierra (10.13) are compatible with Sierra (10.12)

**macOS 10.13 (x86, 64-bit), DMG Archive**

8.0.12

177.2M

[Download](#)

(mysql-8.0.12-macos10.13-x86\_64.dmg)

MDS: ee79241a8393226c9f42f00e3a476e61 | [Signature](#)**macOS 10.13 (x86, 64-bit), Compressed TAR Archive**

8.0.12

126.7M

[Download](#)

(mysql-8.0.12-macos10.13-x86\_64.tar.gz)

MDS: 90956f7d454cf846ea93f5a55cf9f1923b | [Signature](#)**macOS 10.13 (x86, 64-bit), Compressed TAR Archive Test Suite**

8.0.12

49.7M

[Download](#)

(mysql-test-8.0.12-macos10.13-x86\_64.tar.gz)

MDS: 046e06e038932fd15bec79e04cd7923bd | [Signature](#)**macOS 10.13 (x86, 64-bit), TAR**

8.0.12

176.4M

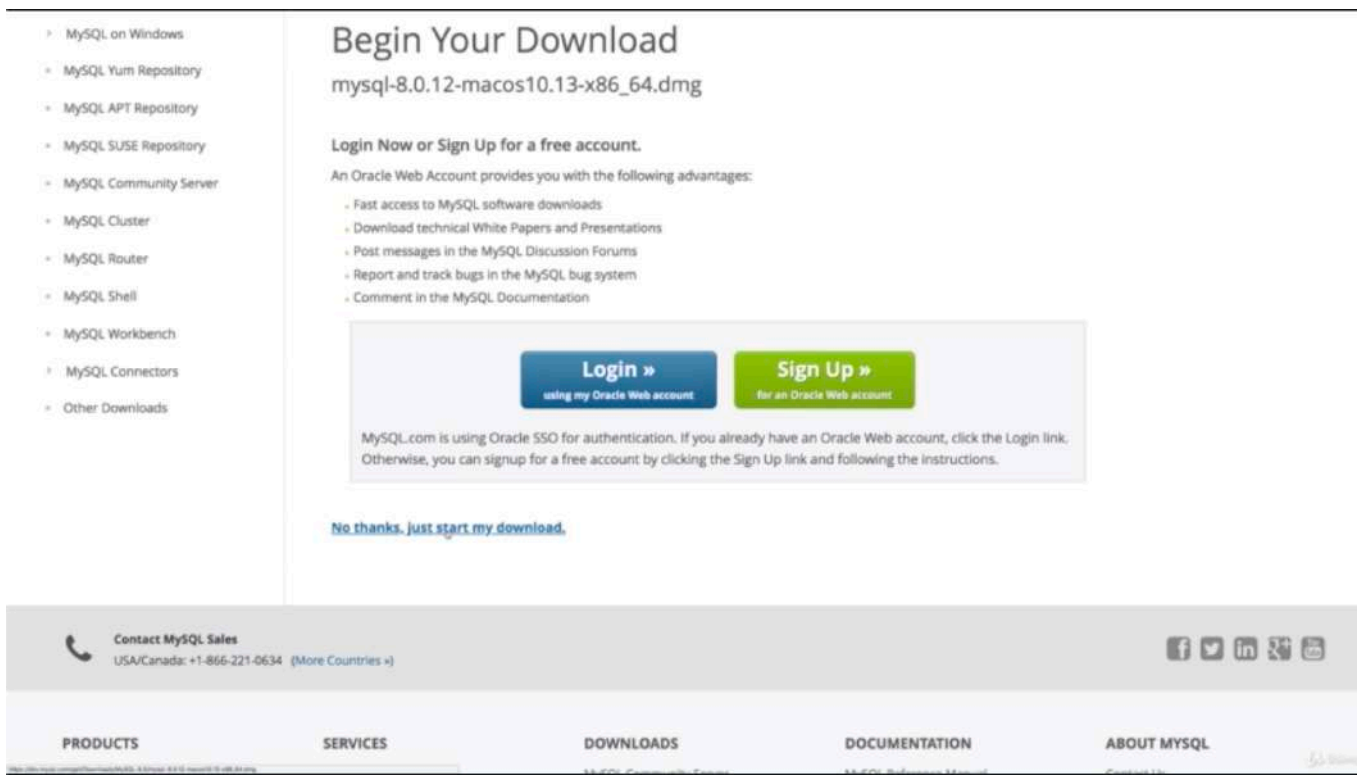
[Download](#)

(mysql-8.0.12-macos10.13-x86\_64.tar)

MDS: 1c5bc32fdbdfe4146850b4e84c8c192 | [Signature](#)

ⓘ We suggest that you use the MDS checksums and GnuPG signatures to verify the integrity of the packages you download.

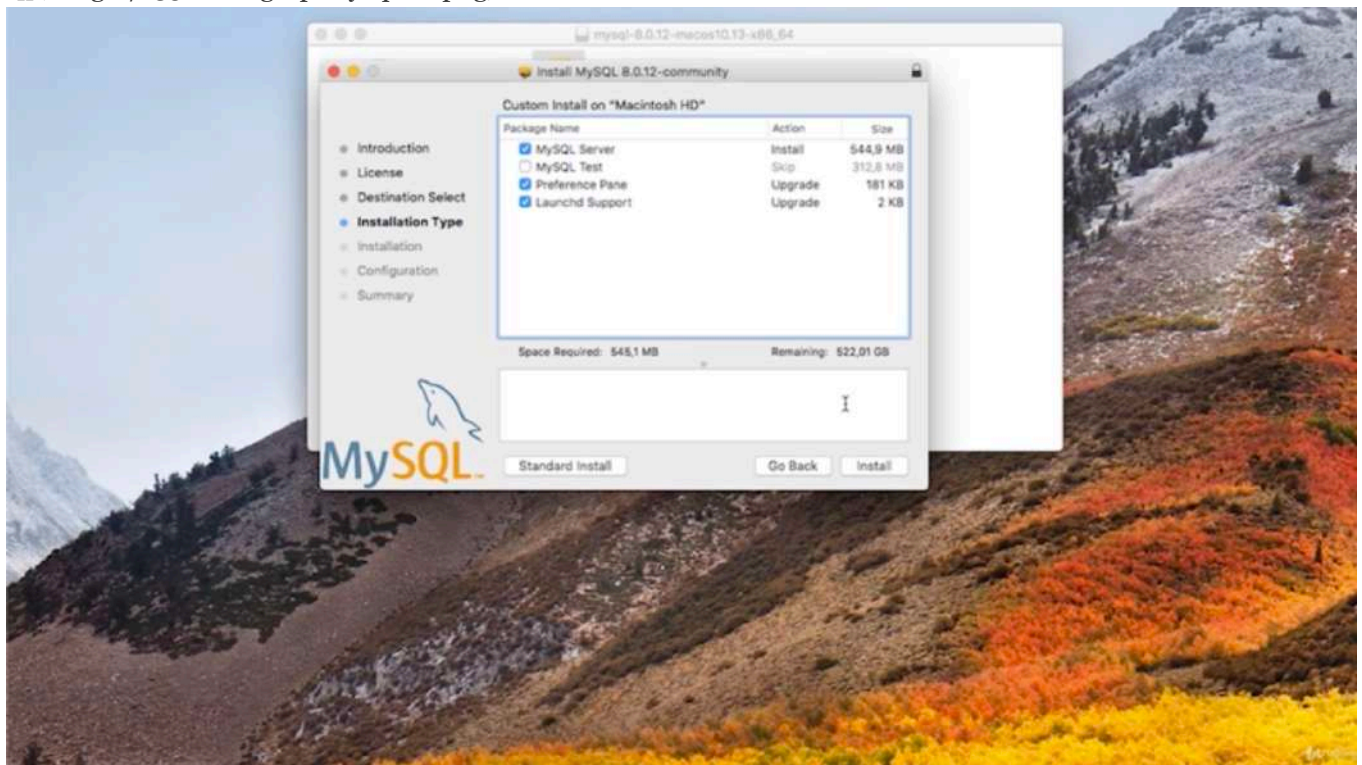




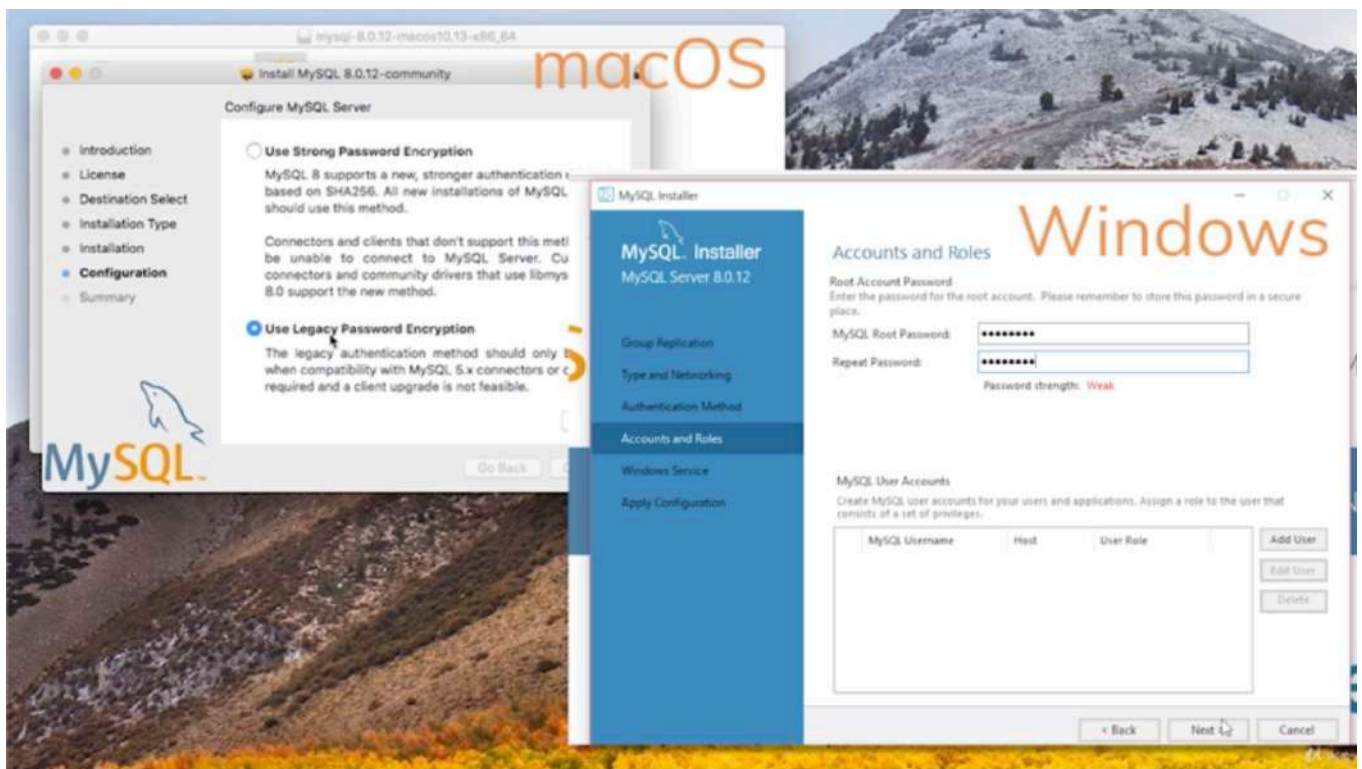
- you can choose 'No thanks, just start my download'





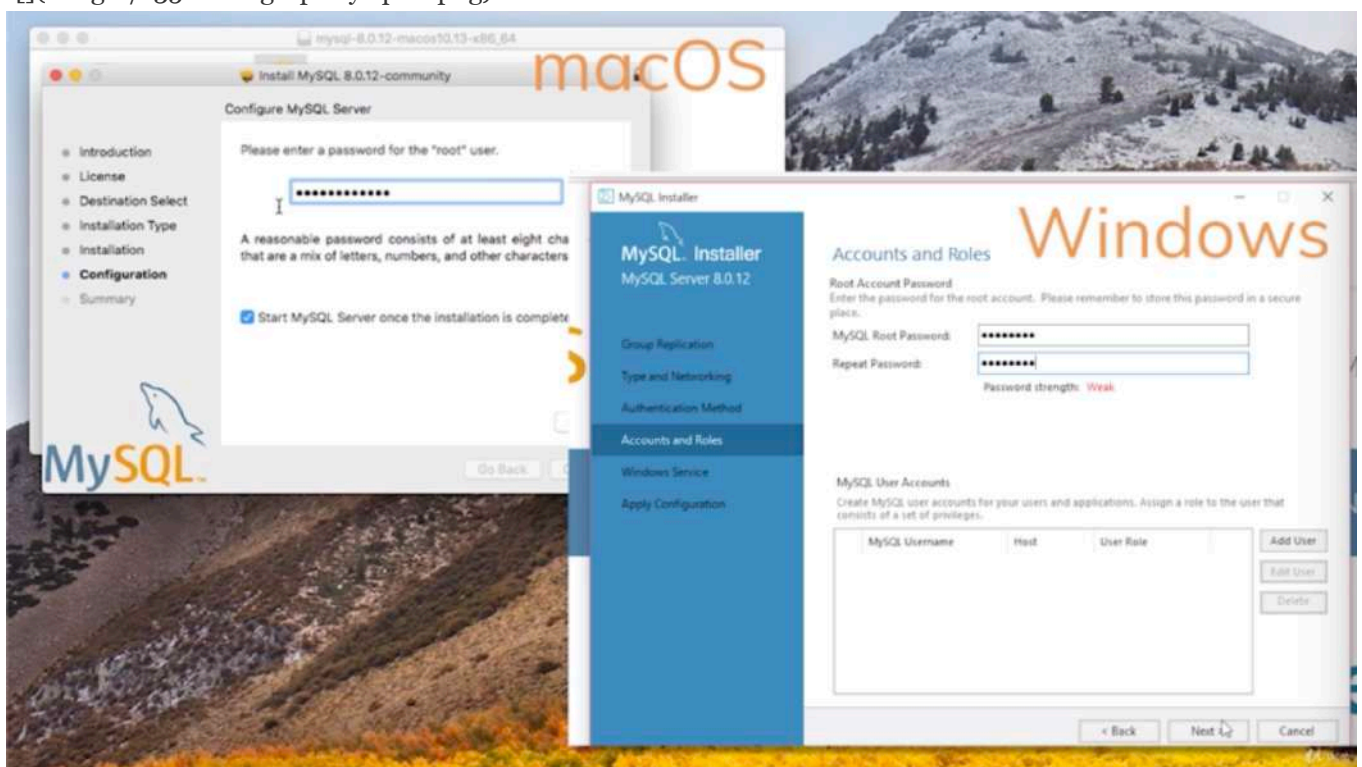







- you should make sure that during this configuration process, you do choose the 'legacy password encryption' which sounds insecure but which is perfectly fine and the newer version simply is not supported by the Node SQL package we are using yet.





- next, you have to choose a password for your root user.





The world's most popular open source database

MySQL.COM   DOWNLOADS   DOCUMENTATION   DEVELOPER ZONE

Contact MySQL | Login | Register

f t in s+ y

Enterprise   **Community**   Yum Repository   APT Repository   SUSE Repository   Windows   Archives

- MySQL on Windows
- MySQL Yum Repository
- MySQL APT Repository
- MySQL SUSE Repository
- MySQL Community Server
- MySQL Cluster
- MySQL Router
- MySQL Shell
- MySQL Workbench**
- MySQL Connectors
- Other Downloads

## Download MySQL Workbench

MySQL Workbench provides DBAs and developers an integrated tools environment for:

- Database Design & Modeling
- SQL Development
- Database Administration
- Database Migration

The Community (OSS) Edition is available from this page under the GPL.

Download source packages of LGPL libraries: [ + ]

MySQL Workbench Windows Prerequisites:

To be able to install and run MySQL Workbench on Windows your system needs to have libraries listed below installed. The listed items are provided as links to the corresponding download pages where you can fetch the necessary files.

- Microsoft .NET Framework 4.5
- Visual C++ Redistributable for Visual Studio 2015

To learn more about MySQL Workbench:

MySQL open source software is provided under the GPL License.

OEMs, ISVs and VARs can purchase commercial licenses.

- The workbench is a client, a visual client we can use to connect to our database to inspect it and play around with it outside of our node application which makes debugging and developing a bit easier.







Looking for the legacy MySQL GUI Tools Bundle (Administrator, Query Browser, Migration Toolkit)?

- Read the EOL Announcement for the MySQL GUI Tools Bundle
- Download Archives for the (EOL) MySQL GUI Tools Bundle

Please report any bugs or inconsistencies you observe to our [Bugs Database](#).

**Thank you for your support!**

**Generally Available (GA) Releases**

### MySQL Workbench 8.0.12

Select Operating System:

macOS

Looking for previous GA versions?

ⓘ Packages for High Sierra (10.13) are compatible with Sierra (10.12)

<b>macOS (x86, 64-bit), DMG Archive</b>	8.0.12	105.1M	<a href="#">Download</a>
<small>(mysql-workbench-community-8.0.12-macos-x86_64.dmg)</small>			
<small>MD5: f8e6499da5a09dad8b455487b15b06e8   <a href="#">Signature</a></small>			

ⓘ We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.

**Contact MySQL Sales**

USA/Canada: +1-866-221-0634 [\(More Countries +\)](#)

f t in s+ y

- MySQL on Windows
- MySQL Yum Repository
- MySQL APT Repository
- MySQL SUSE Repository
- MySQL Community Server
- MySQL Cluster
- MySQL Router
- MySQL Shell
- MySQL Workbench
- MySQL Connectors
- Other Downloads

## Begin Your Download

mysql-workbench-community-8.0.12-macos-x86\_64.dmg

### Login Now or Sign Up for a free account.

An Oracle Web Account provides you with the following advantages:

- Fast access to MySQL software downloads
- Download technical White Papers and Presentations
- Post messages in the MySQL Discussion Forums
- Report and track bugs in the MySQL bug system
- Comment in the MySQL Documentation

Login »

using my Oracle Web account

Sign Up »

For an Oracle Web account

MySQL.com is using Oracle SSO for authentication. If you already have an Oracle Web account, click the Login link. Otherwise, you can sign up for a free account by clicking the Sign Up link and following the instructions.

[No thanks, just start my download.](#)



Contact MySQL Sales

USA/Canada: +1-866-221-0634 [\(More Countries »\)](#)



PRODUCTS

SERVICES

DOWNLOADS

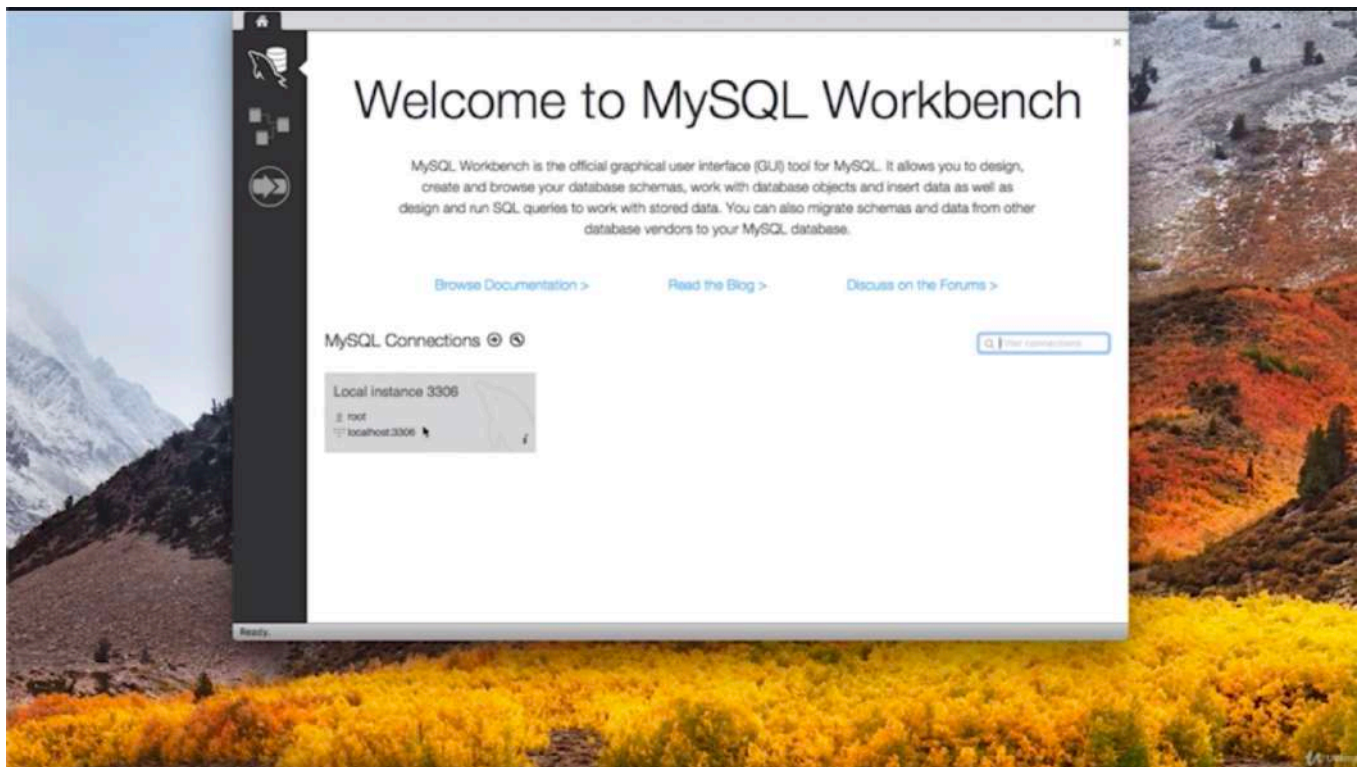
DOCUMENTATION

ABOUT MYSQL



- you can now test your setup by starting the MySQL workbench you just installed  



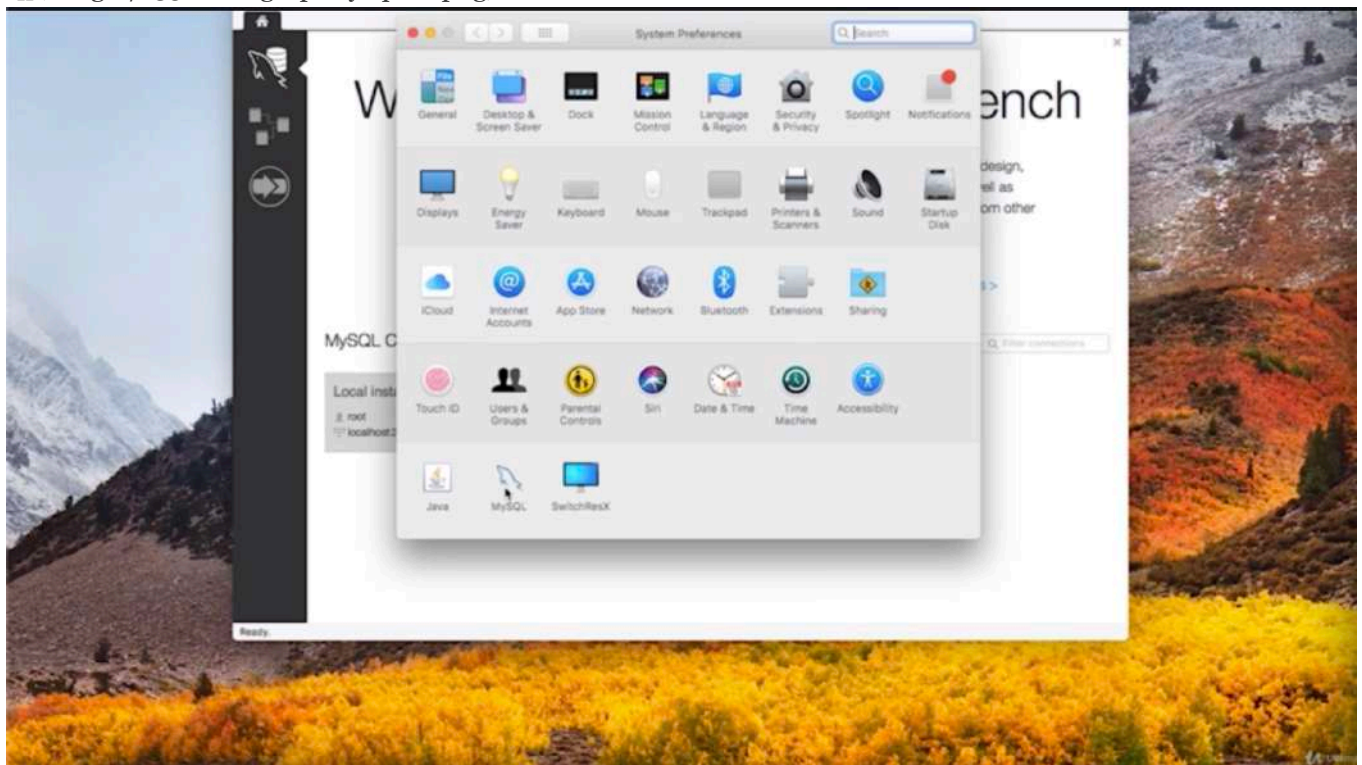



- and then there you should already see your MySQL instance running. if not the case, have a look at the attached document where i describe some common issues or in general give you some links on how to make this work and how to bring this up.

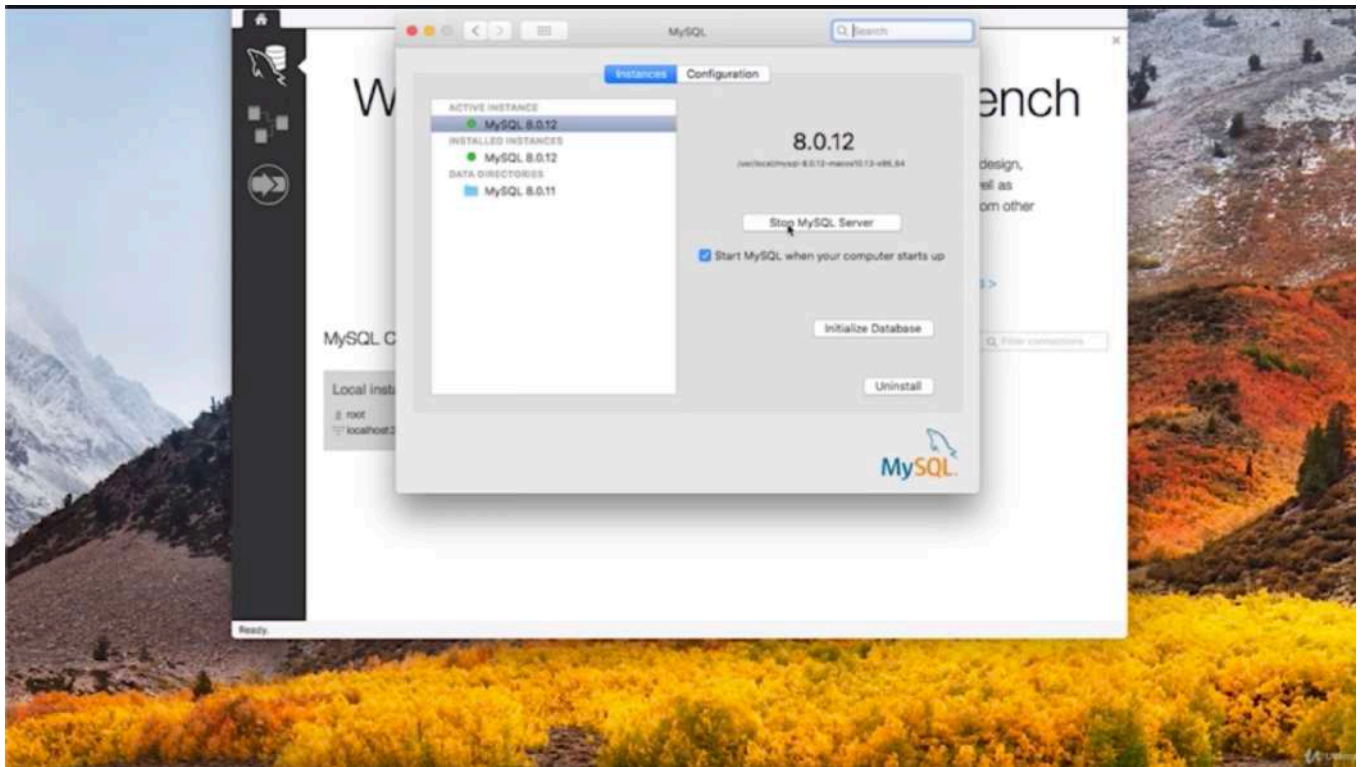
- you need to make sure that your MySQL server is running and during installation, you had a choice to check that it should always start with your system.







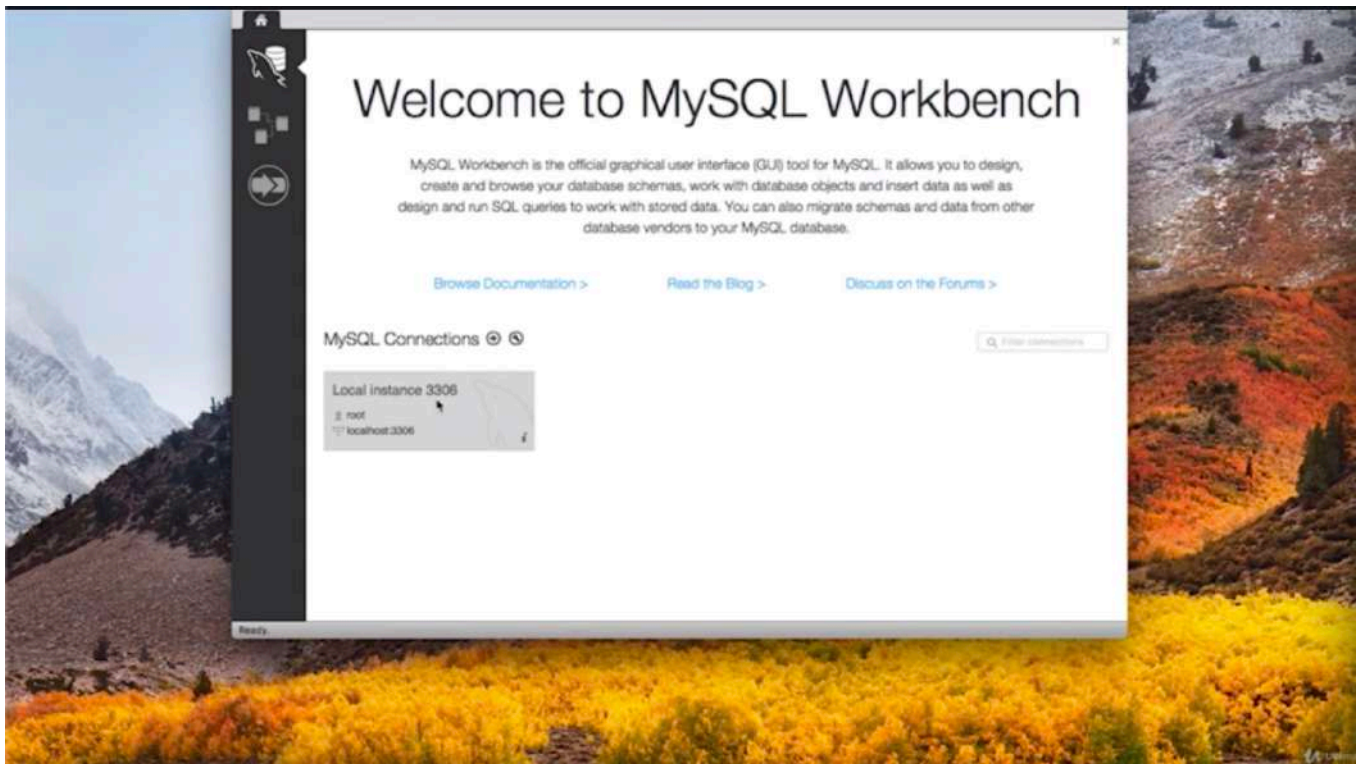


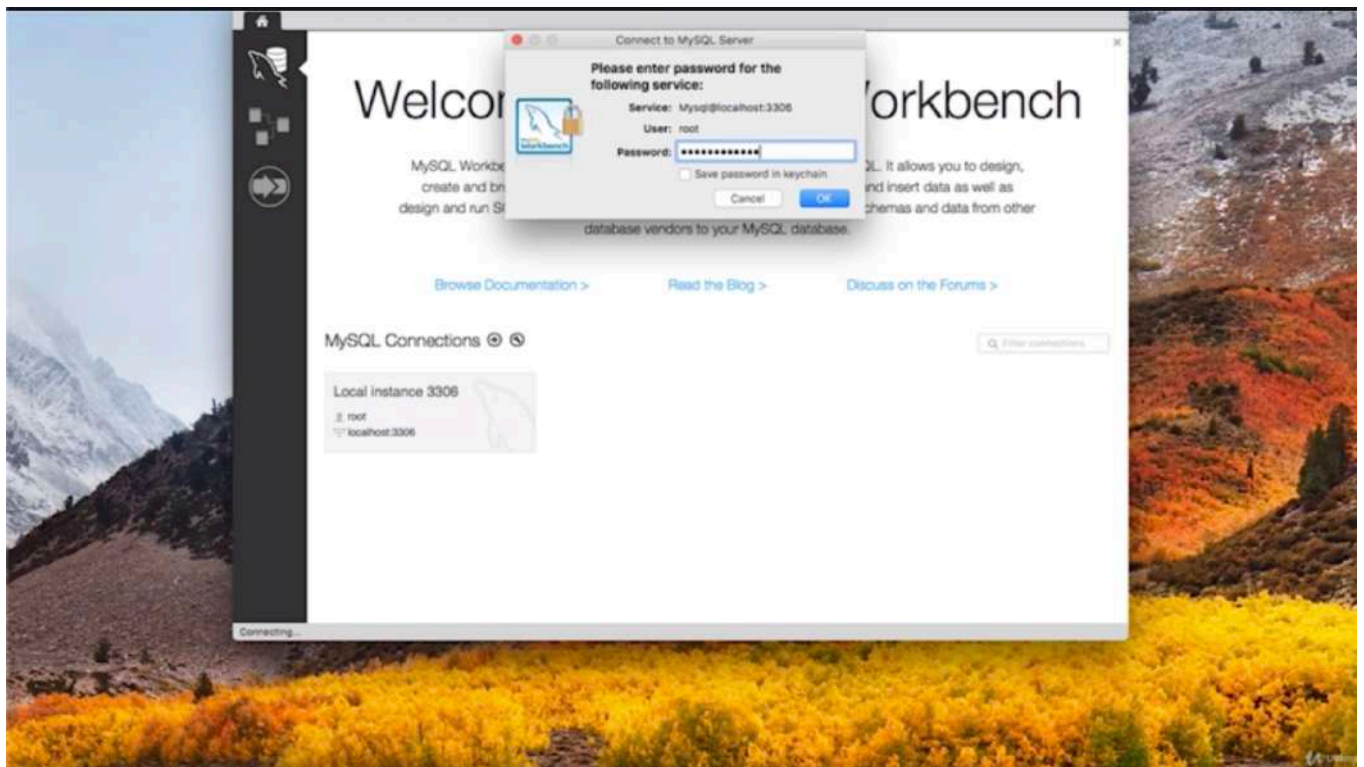


- on MacOS, you can open your system preferences and there you should have the MySQL option where you can also stop and start the server.





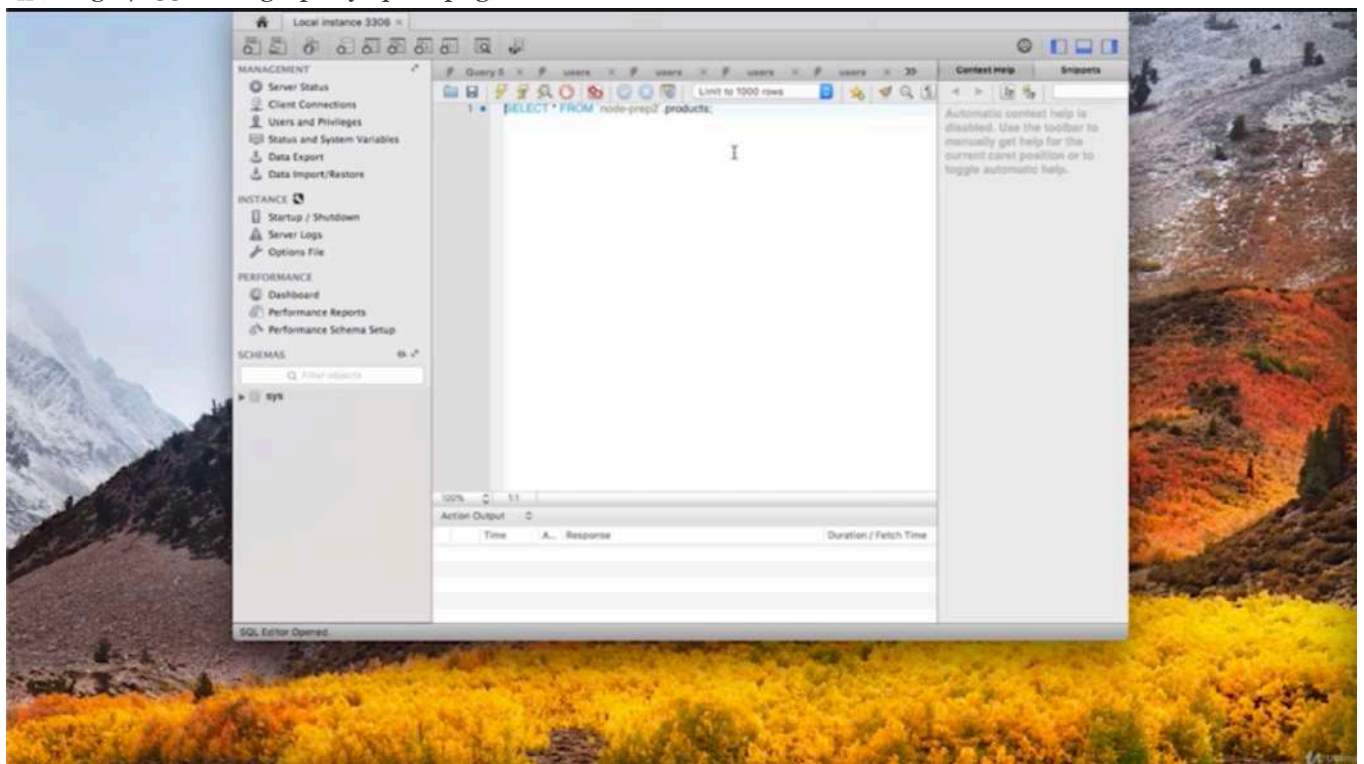


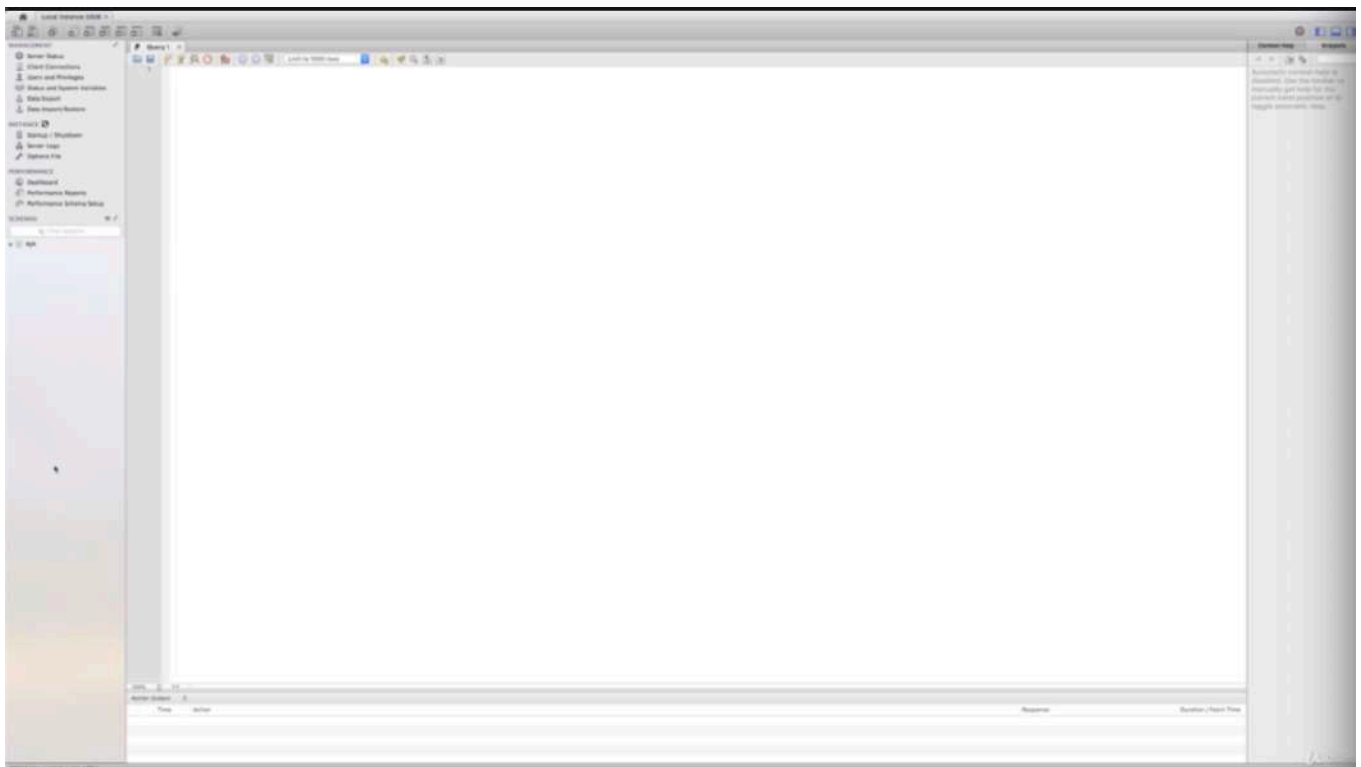


- To connect your database, click on that instance and enter that root password you assigned during the installation. this should allow you to connect to your SQL server instance like this.

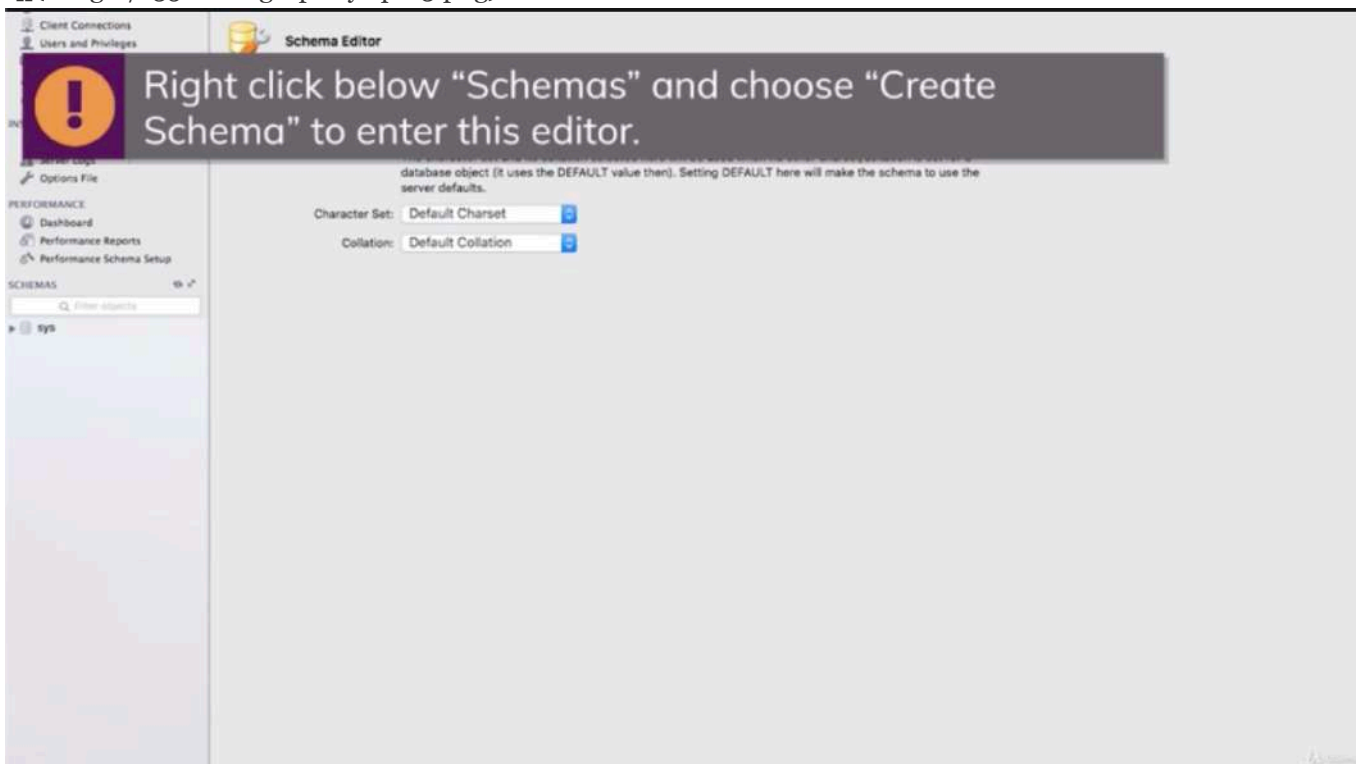




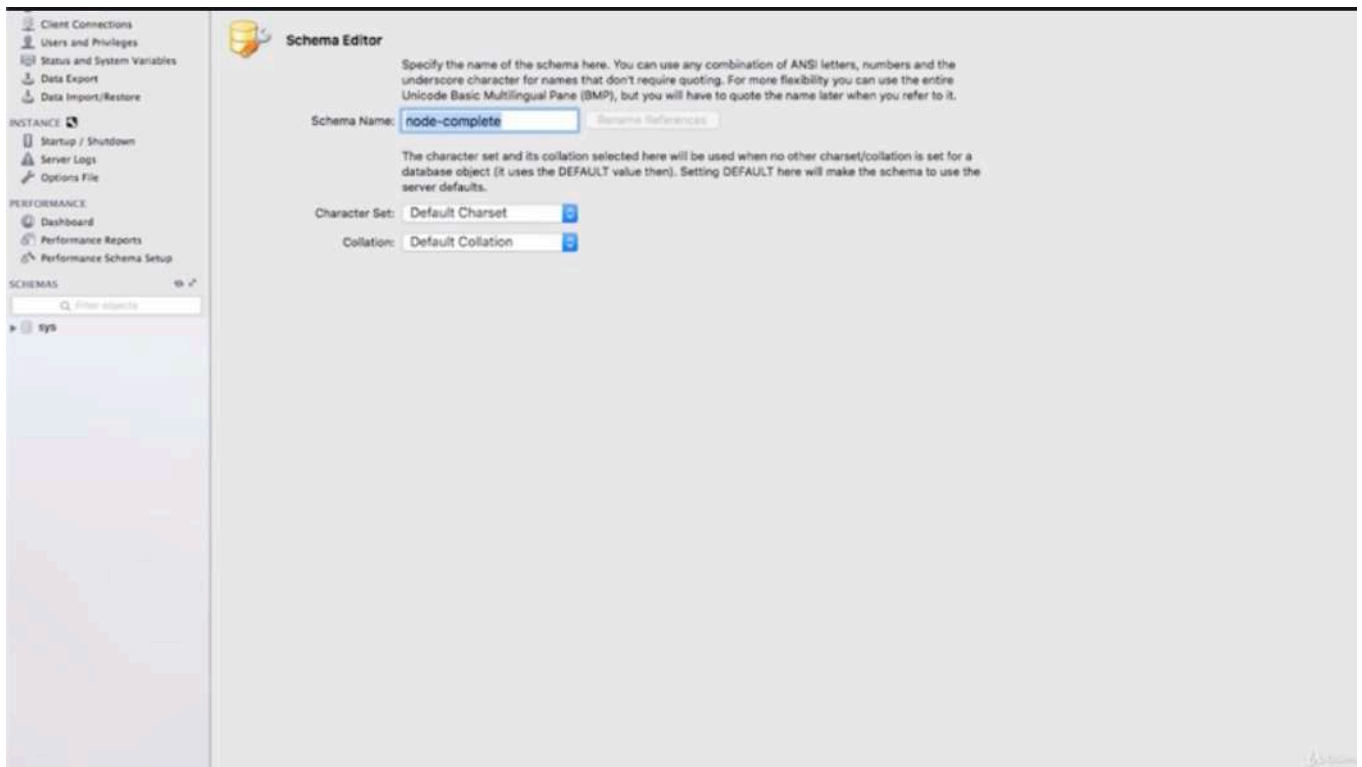




- then once you enter the password, you should be connected to your database system. and this is the empty window and we won't work too much in that. we will work with our database from inside our node application. but this will allow us to conveniently look into our database from time to time and see what is stored  
 



and one thing we can do is we can go down to schemas which can be translated with database.  
   
 



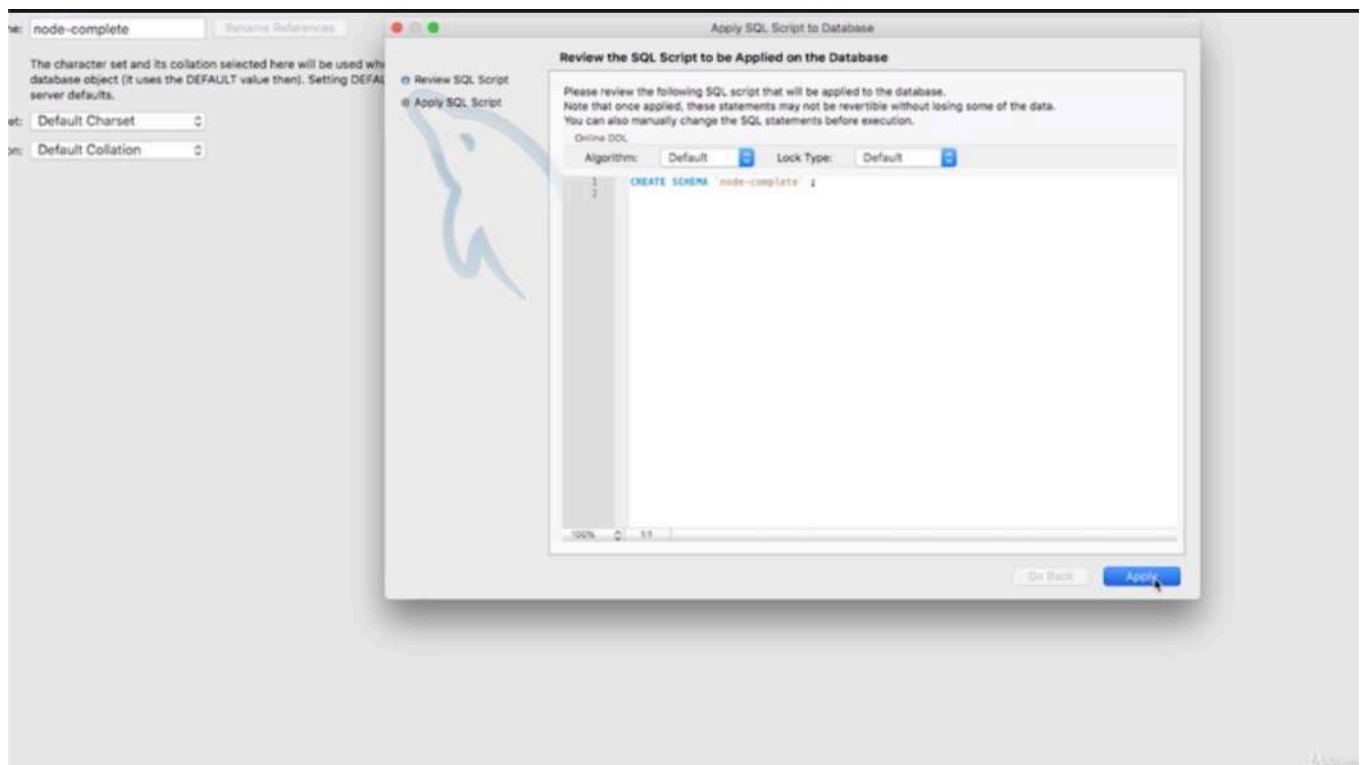
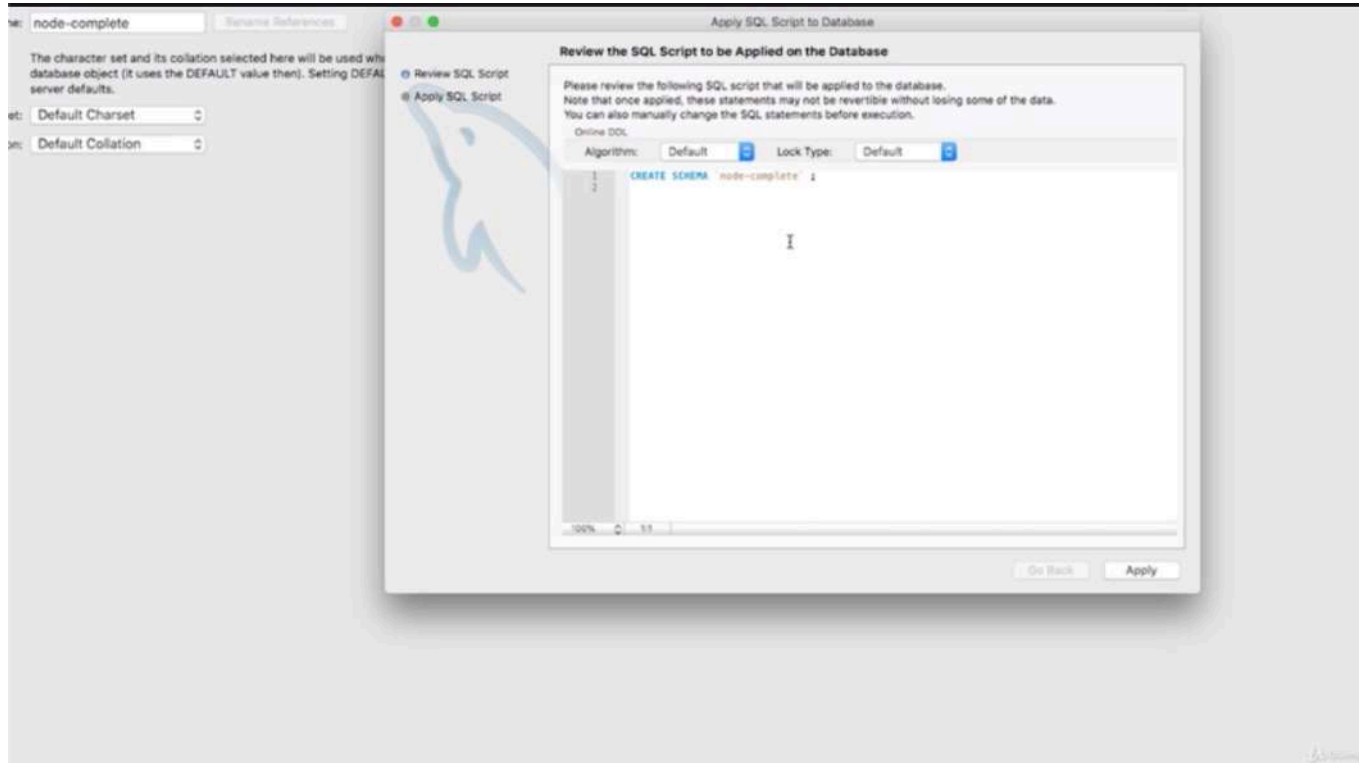
- we can define a new schema, so a new database with which we will work and i will name it 'node-complete'. you can name this whatever you want and you can leave the other setting here and then on the bottom, you can click 'apply'.





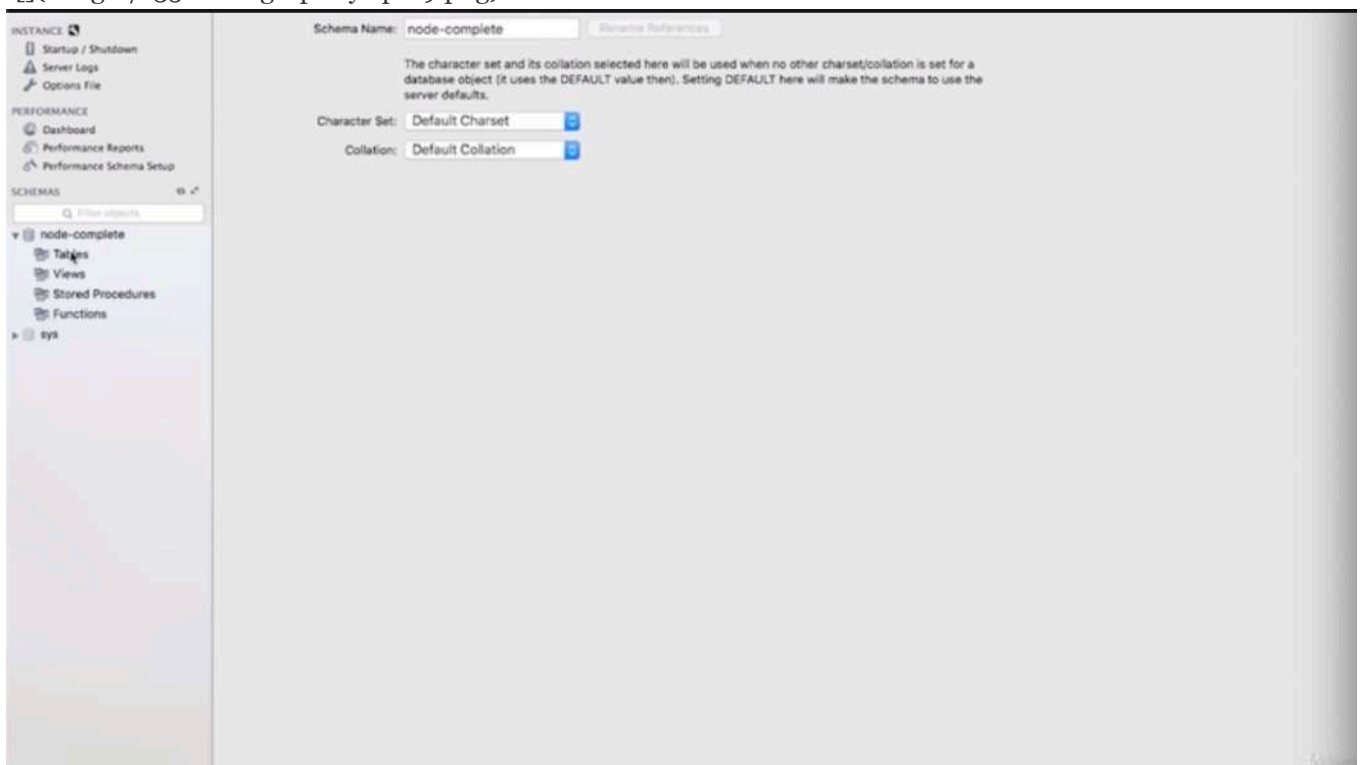




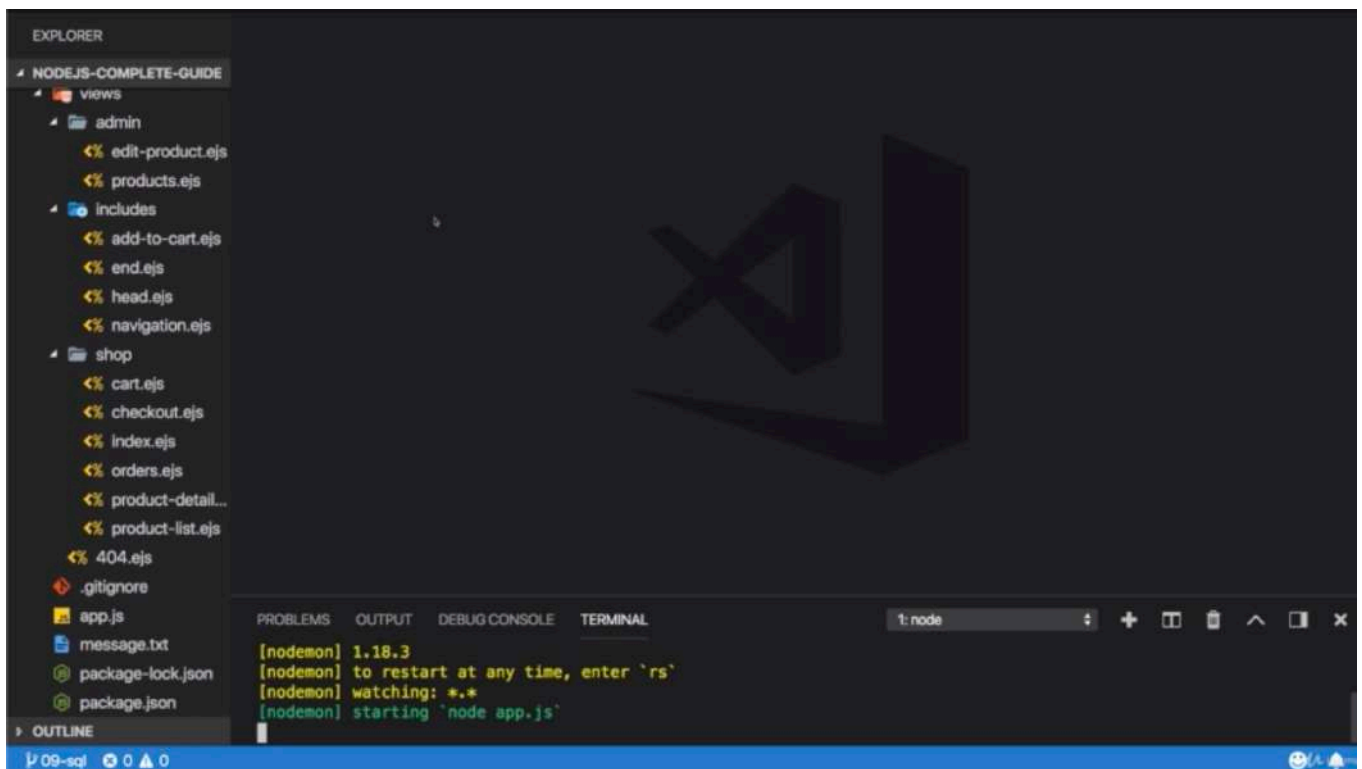




- this will create a new database which we can interact later, you can leave all the default settings and click apply  

and you should see that 'node-complete' thing which has a couple of tables or none right now. but where you can then later connect to and store your data in the tables that will be created here.  

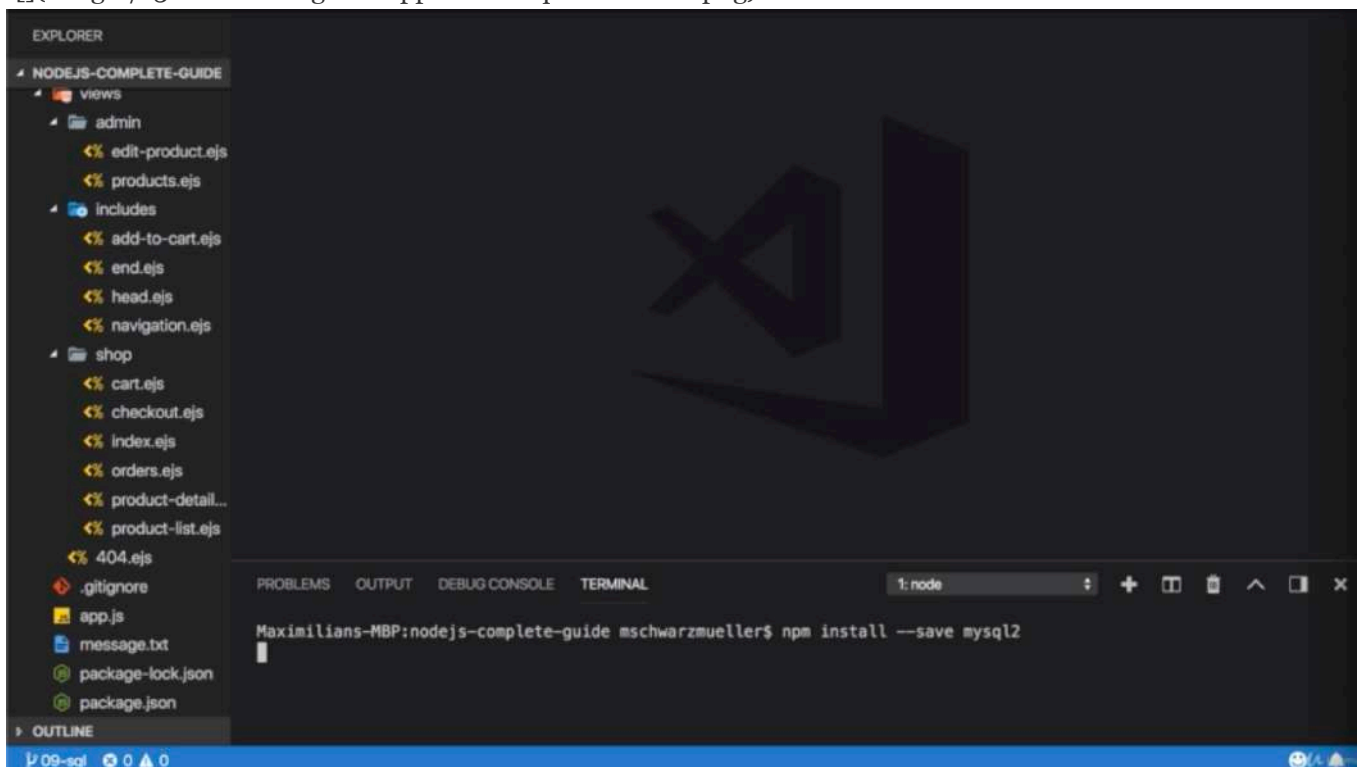
- so with that, we can continue and we can now move on with our code and start interacting with MySQL from inside our node application.

## \* Chapter 136: Connecting Our App To The SQL Database

1. update

- ./util/database.js
- app.js





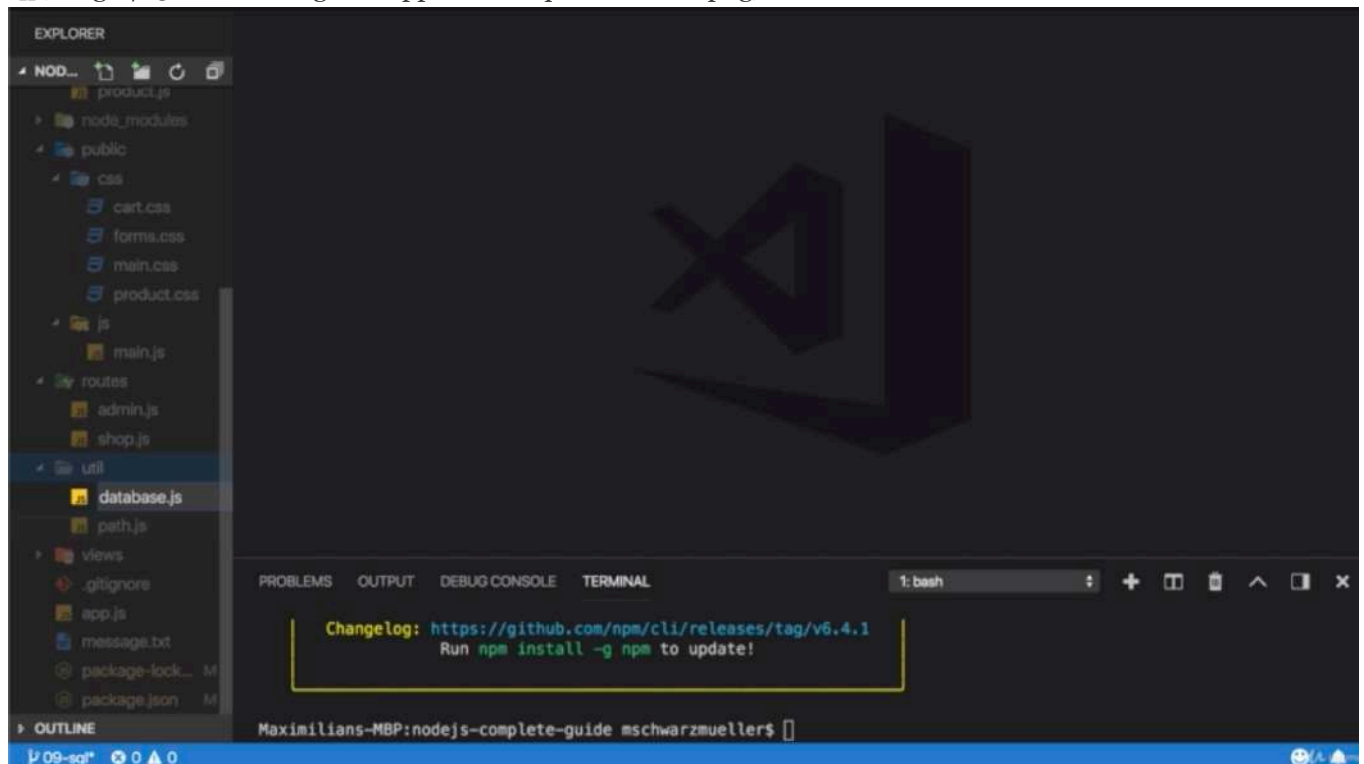
- To use SQL or to interact with SQL from inside node, we need to install a new package and we do that with 'npm



install'

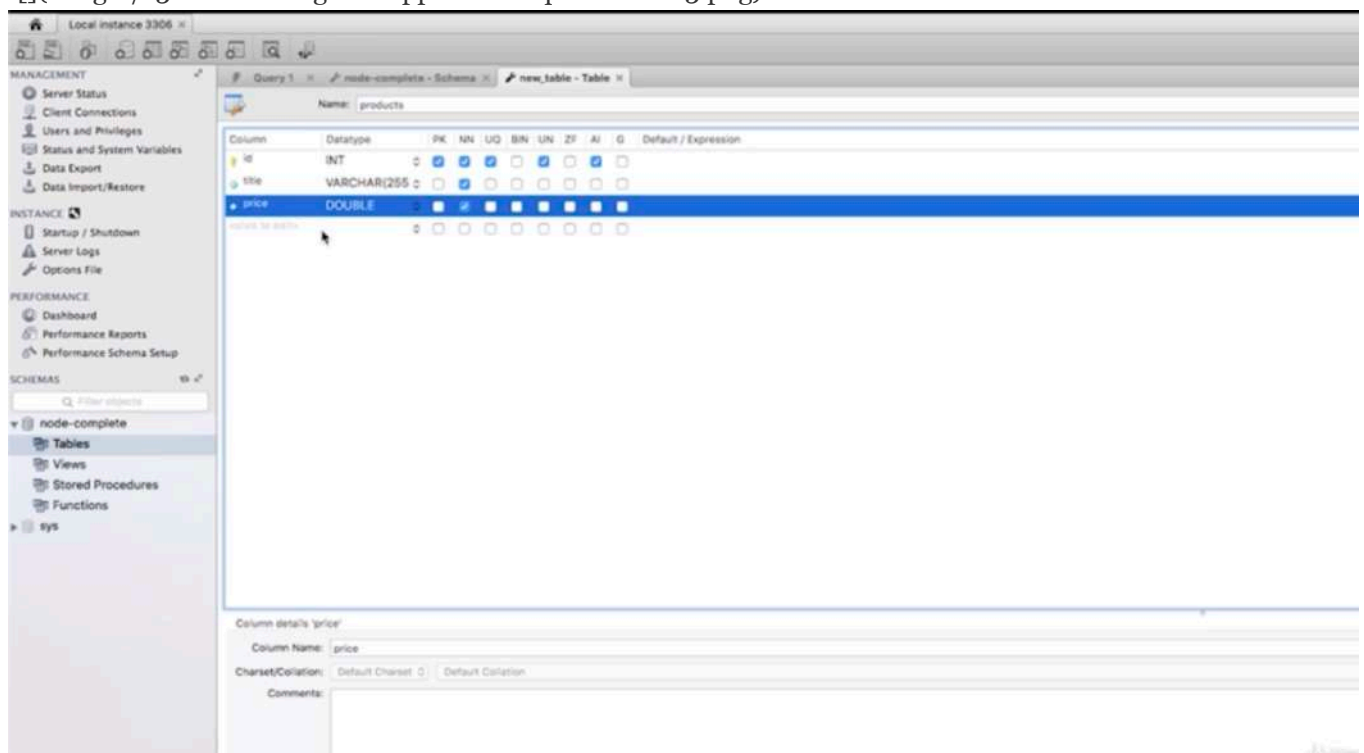
- 'npm install --save' because it will be production dependency, one which is a crucial part of our application
- the name is 'mysql2', this is a later version of well MySQL1. as you might guess and it allows us to write SQL code and execute SQL code in node and interact with a database.





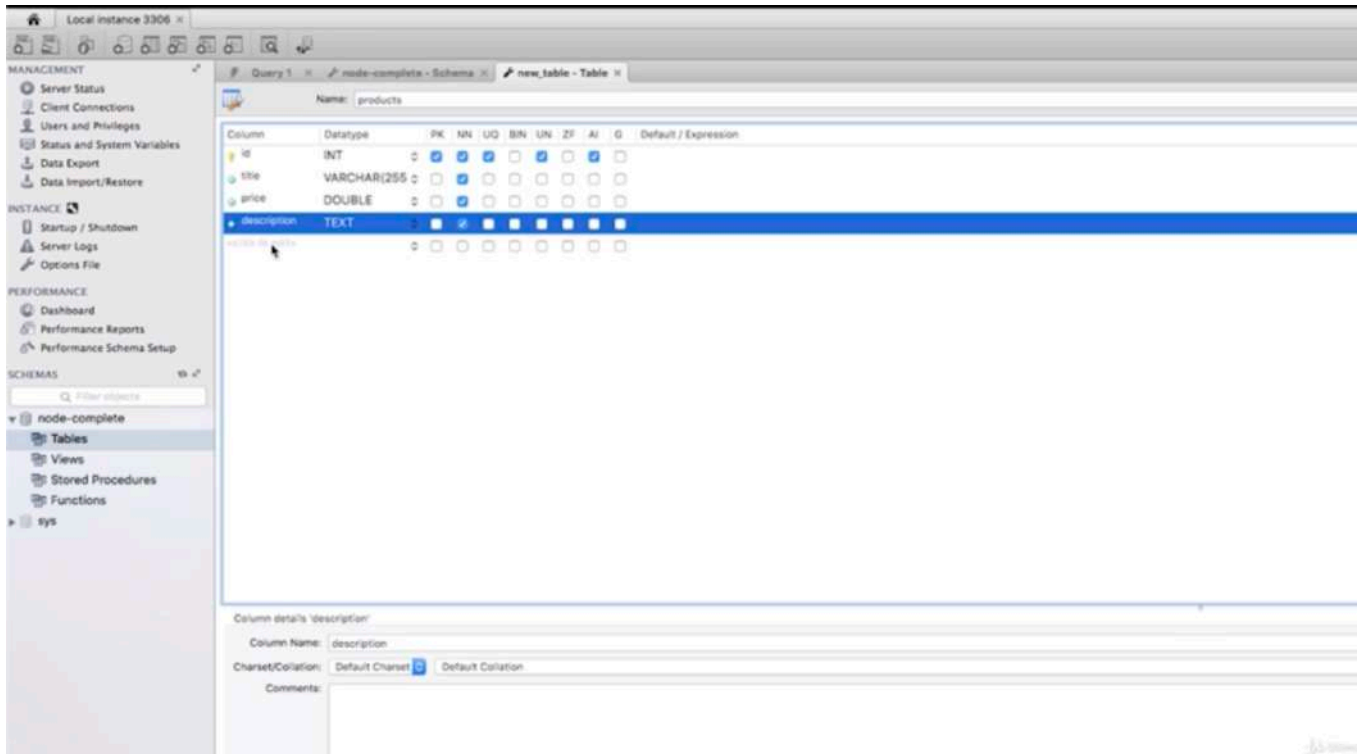
- with that installed, we made one important step forward toward using MySQL.
- The next step is that we need to connect our database from inside our application and for that, i go to the 'util' folder we created in past, and there we have that path.js file in there which we don't really use anymore but we can also create a new file in there , which name is 'database.js'.
- in there, i wanna set up the code that will allow us to connect to the SQL database and then also give us back a connection object which allows us to run queries.





- go back to our 'workbench', and in the table, we are gonna create table.





- The name should be 'products' and in there, you can add new fields.

```
1 //app.js
2
3 const path = require('path');
4
5 const express = require('express');
6 const bodyParser = require('body-parser');
7
8 const errorController = require('./controllers/error');
9 const db = require('./util/database');
10
11 const app = express();
12
13 app.set('view engine', 'ejs');
14 app.set('views', 'views');
15
16 const adminRoutes = require('./routes/admin');
17
18 const shopRoutes = require('./routes/shop');
19 /**one of them is 'execute' which allows us to execute queries
20 * we can also end it whenever our application is to shut down.
21 *
22 * we wanna connect or execute a command
23 * and we can execute a command by writing SQL syntax as string
24 * that means that you need to know SQL.
25 */
26 db.execute('SELECT * FROM products')
27
28 app.use(bodyParser.urlencoded({ extended: false }));
29 app.use(express.static(path.join(__dirname, 'public')));
30
31 app.use('/admin', adminRoutes);
```

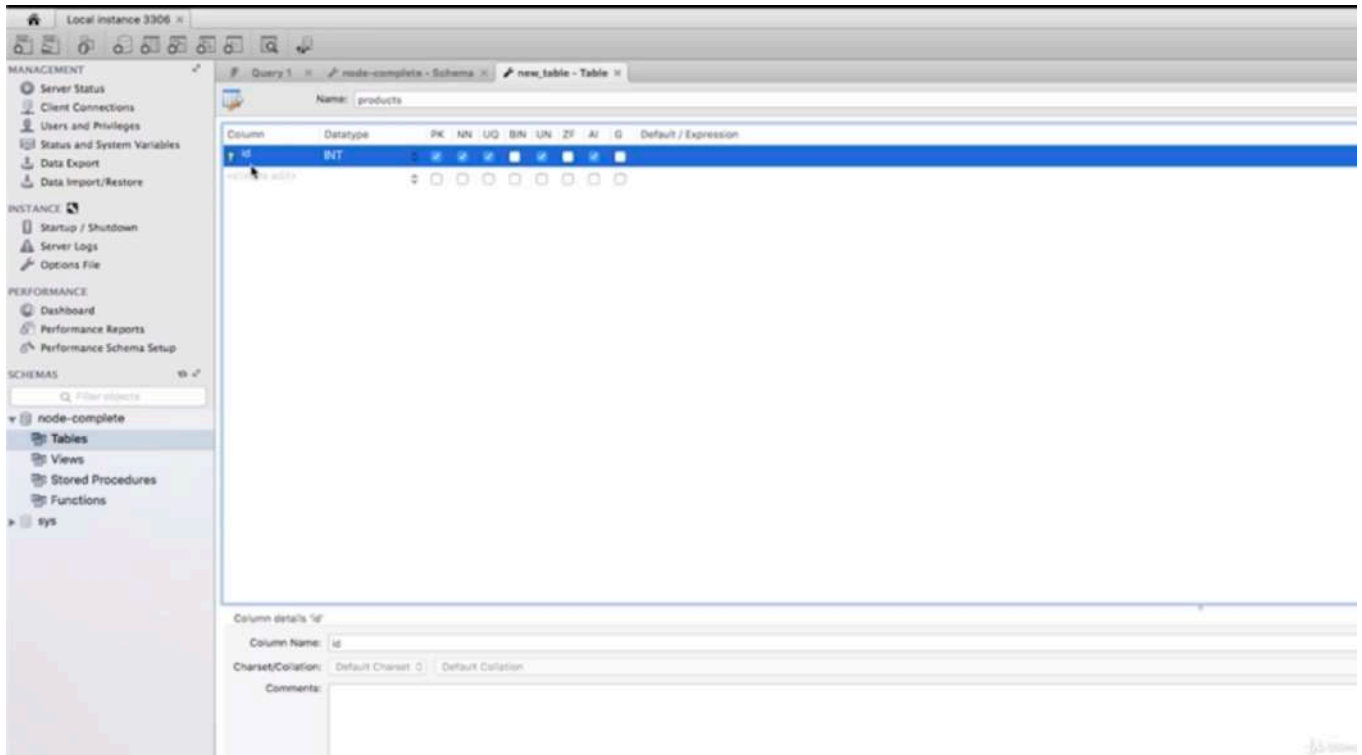
```

32 app.use(shopRoutes);
33
34 app.use(errorController.get404);
35
36 app.listen(3000);
37
1  const mysql = require('mysql2')
2
3  /**i will call 'createPool'
4   * and create connection is not what we want
5   * we don't want a single connection
6   * but a pool of connections which will allow us to always reach out to it whenever we have
   a query to run
7   * and then we get a new connection from that pool which manages multiple connection
8   * so that we can run multiple queries simultaneously
9   * because each query needs its own connection
10  * and once query is done,
11  * the connection will be handed back into the pool
12  * and it's available again for a new query
13  * and the pool can then be finished when our application shuts down
14  *
15  * and i need to pass in a javascript object
16  * with some information about our data engine
17  * we are connecting to
18  * */
19  const pool = mysql.createPool({
20      /**'localhost' because we are running it on our local machine
21       *
22       * then i need to define the username
23       * and that by default is root that was given to us during the configuration
24       *
25       * i also need to define the exact database name
26       * because this gives us access to a database server
27       * but that server typically has multiple databases
28       * and our databases are our schemas.
29       */
30      host: 'localhost',
31      user: 'root',
32      database: 'node-complete',
33      password: 'rldnjs12'
34  })
35  /**i will export it in a special way
36   * i will call 'promise()'
37   * because this will allow us to use 'promises' when working with these connections
38   * which handle asynchronous tasks, asynchronous data instead of callbacks
39   * because promises allow us to write code in more structured way,
40   * we don't have many nested callbacks.
41   * instead we can use promise chain
42   */
43
44  module.exports = pool.promise();

```

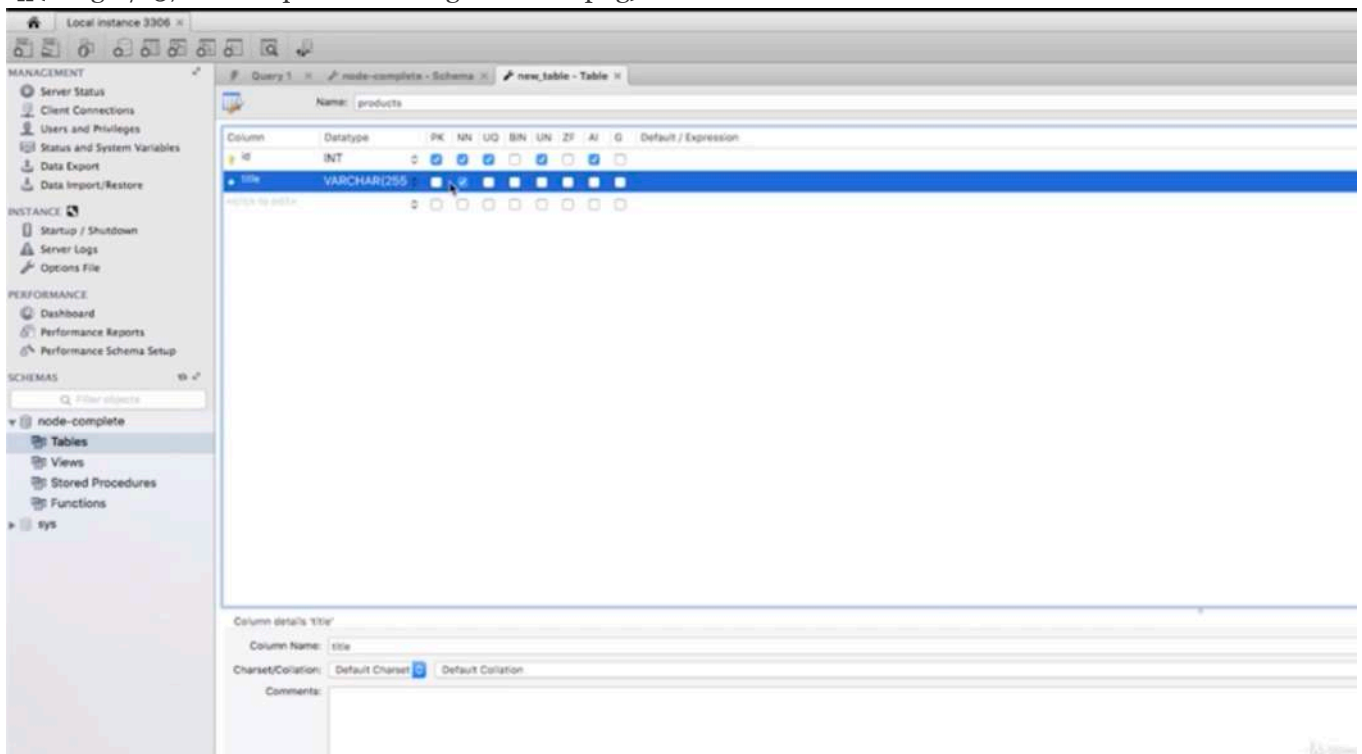
## \* Chapter 137: Basic SQL & Creating A Table





- first of all, define the name of a field, ID and the data type. and for the ID, an integer is fine.
  - we can also check that it should be the primary key(PK) by which records in this table will be identified.
- that it must not be null(NN), that it should be unique(UQ), that should definitely be the case, if it should hold binary data(BIN) which is not the case, if it's unsigned(UN) so if it holds no negative values which should also be the case because that should be an integer starting at 1 and then incrementing, if it is zero fill(ZF) and for us important, if it's auto-incrementing(AI) and that should be the case because every new record should receive that automatically and it should be a higher number than in the last record.

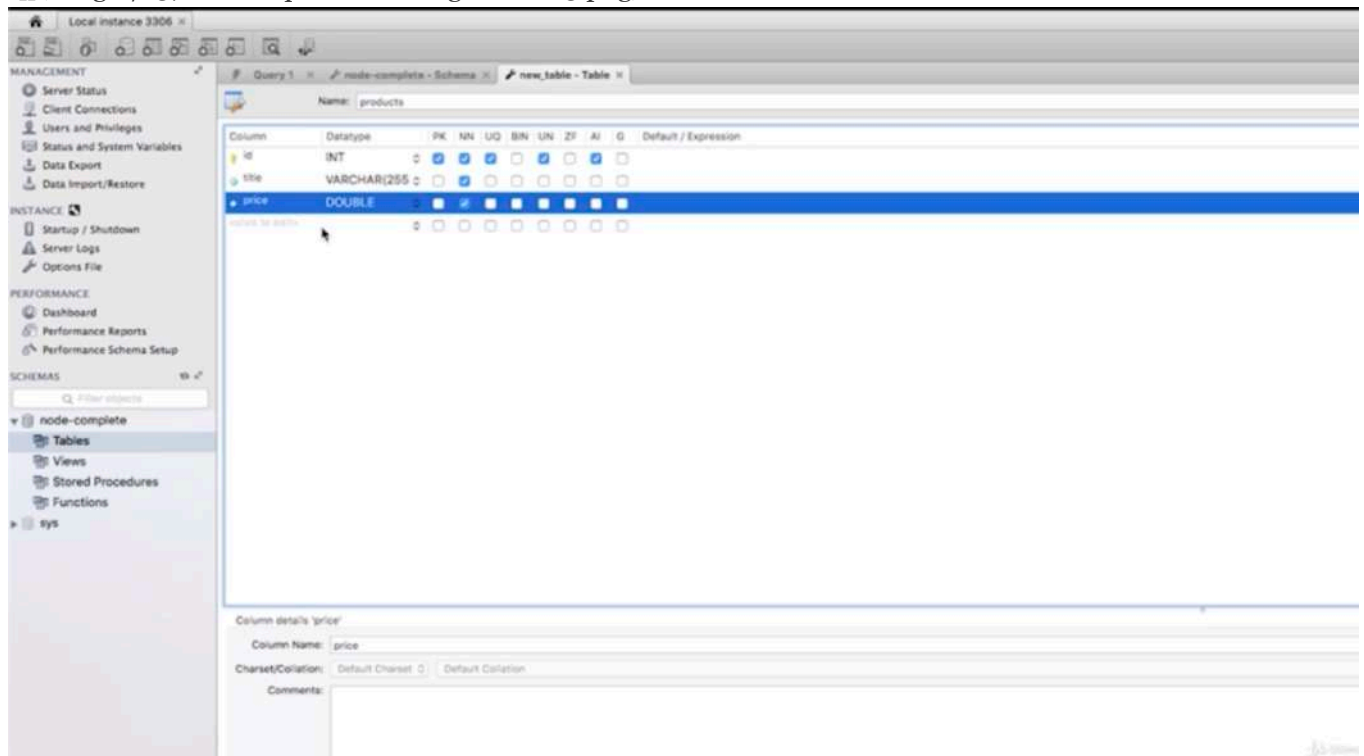




- A product typically has a title and there i will use a VARCHAR which is a string. i will define that it may be up to 255 characters long and longer titles will be cut off. so that is what we have to keep in mind. it must also not be

null. so we have to have a value in there but i don't need any other setting here.

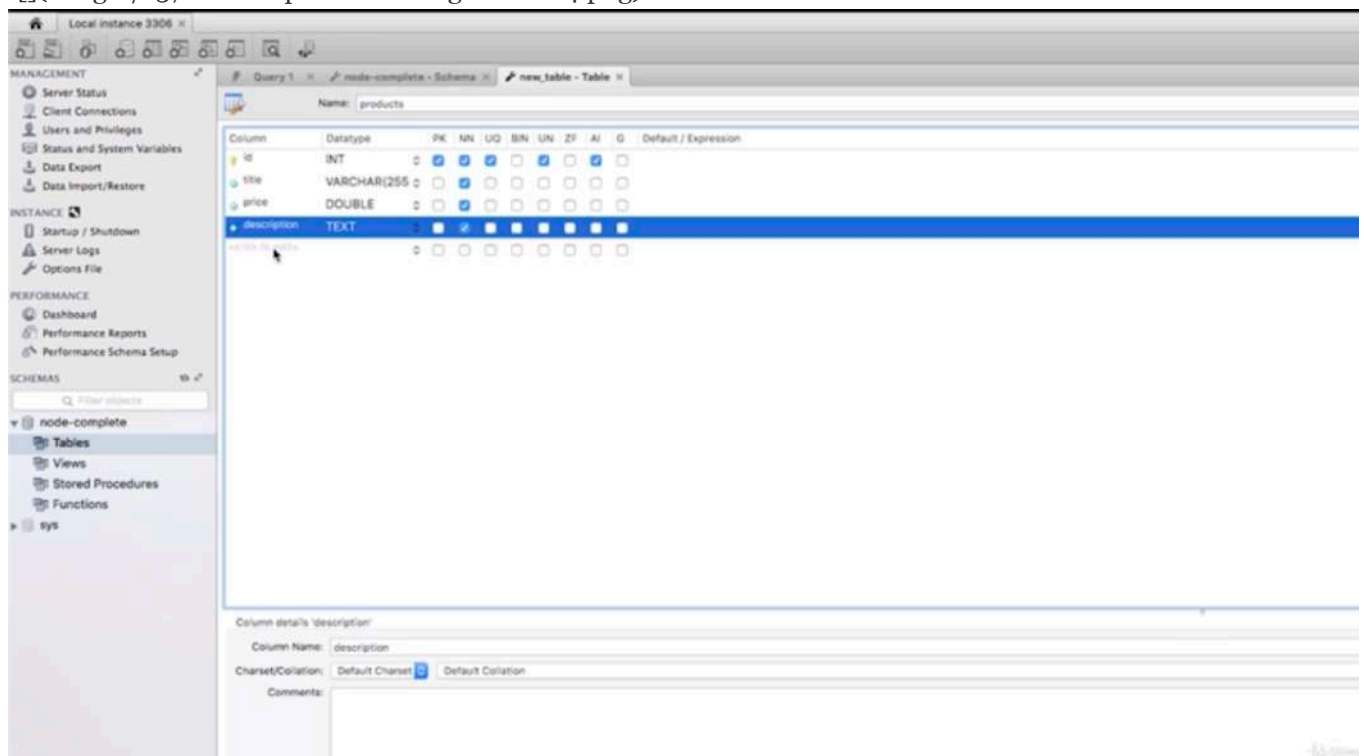




- for a product, i wanna have a price. and i wanna have DOUBLE so that we can enter decimal places.

- this must also not be null. i also wanna have a description which now will not be a VARCHAR.

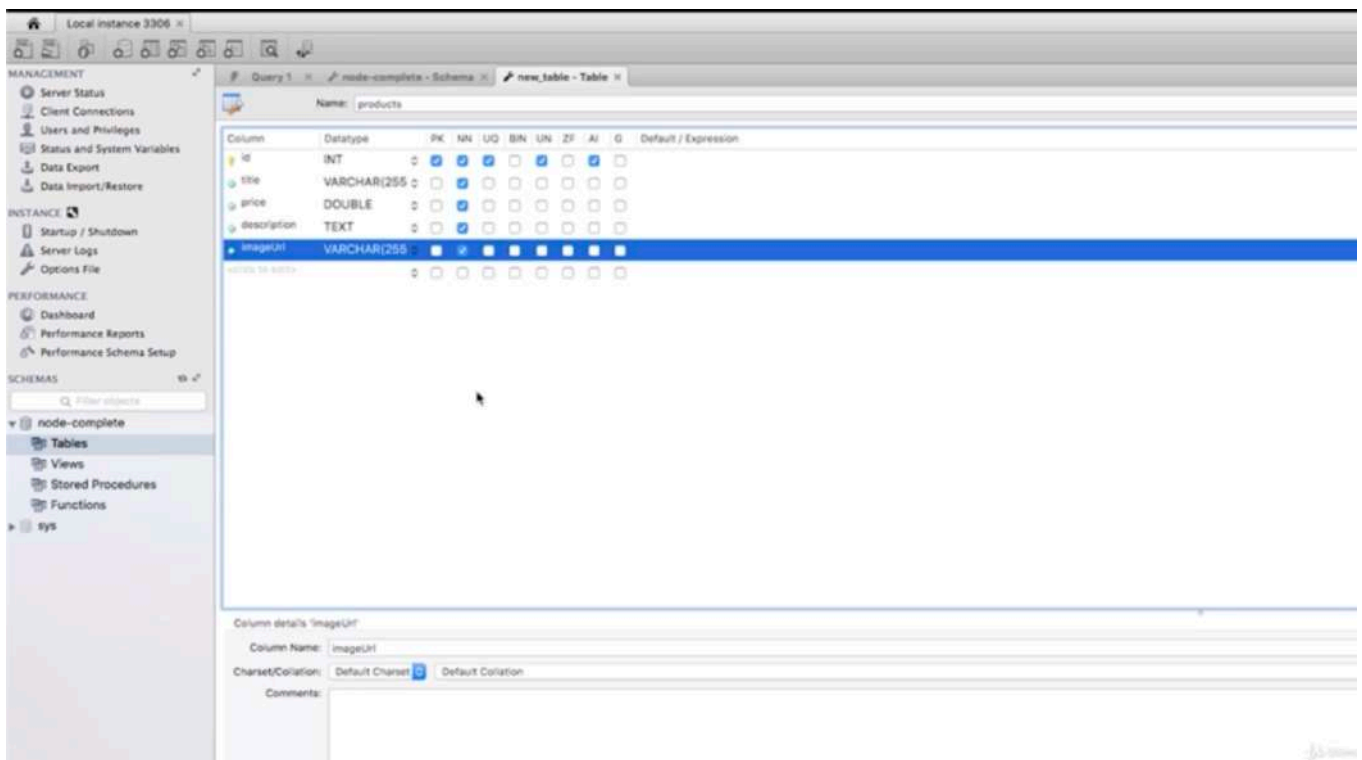




- i also wanna have a description which now will not be a VARCHAR. but will be TEXT and if you are wondering which data types are available, that is exactly what i meant. you should definitely consult a full SQL course to learn more about the available data types.

- here i got my text which is a longer text than the VARCHAR which has a limitation.





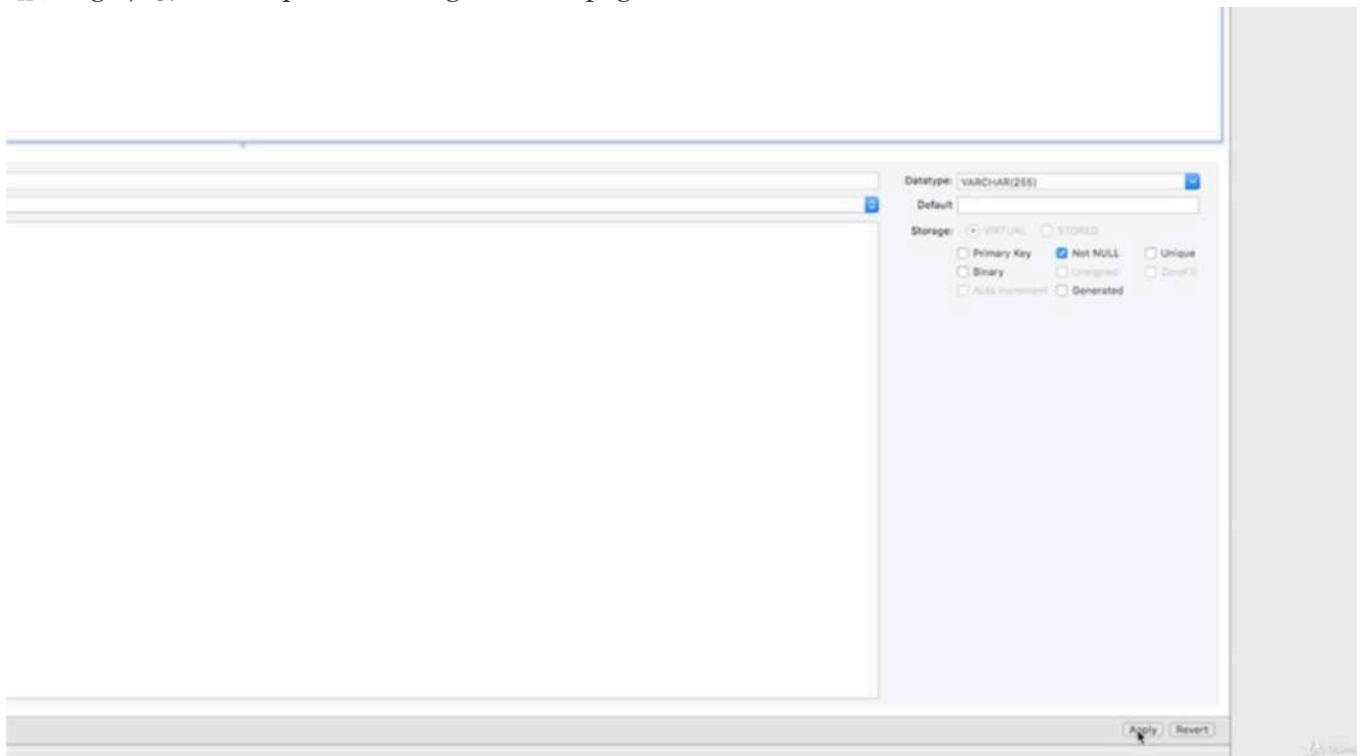
- i will have an image URL which i will set to VARCHAR 255 which means longer URLs also won't work.

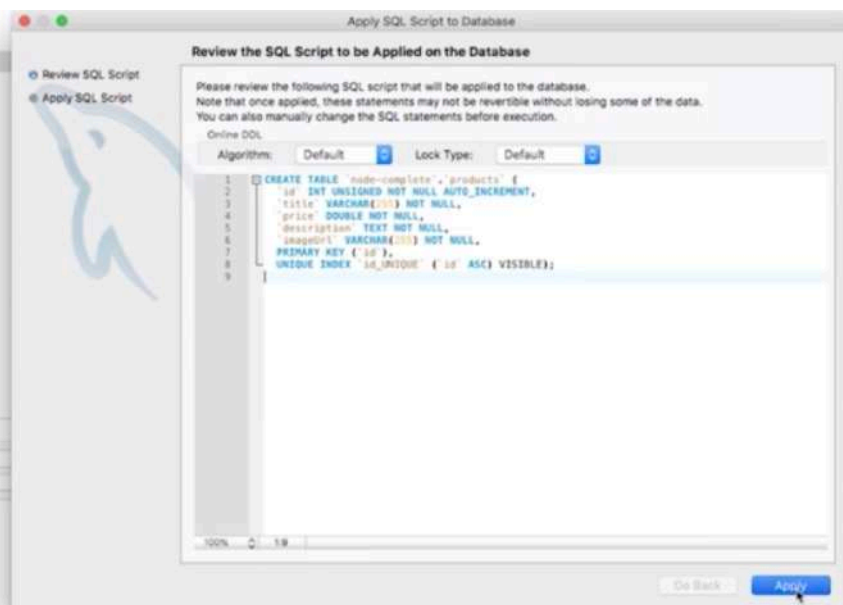
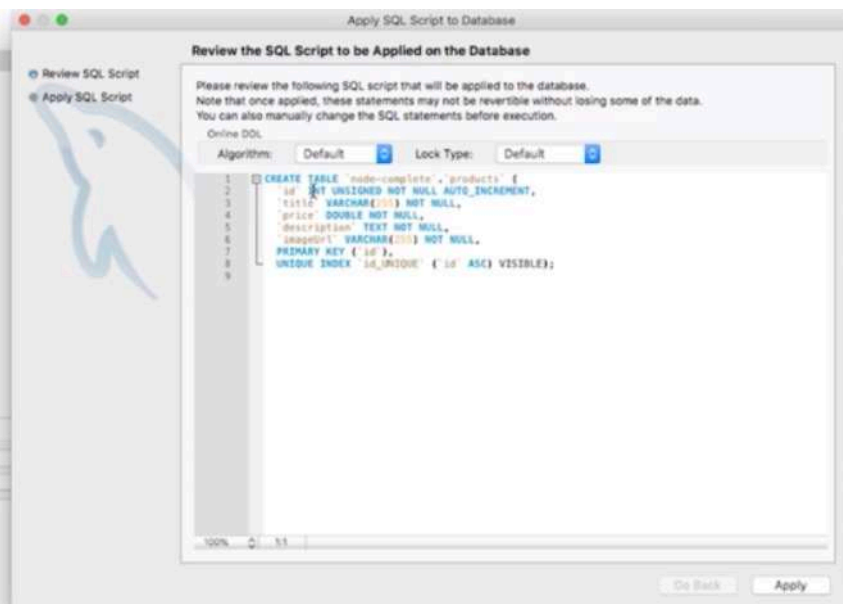
- i defined how my product should look like.







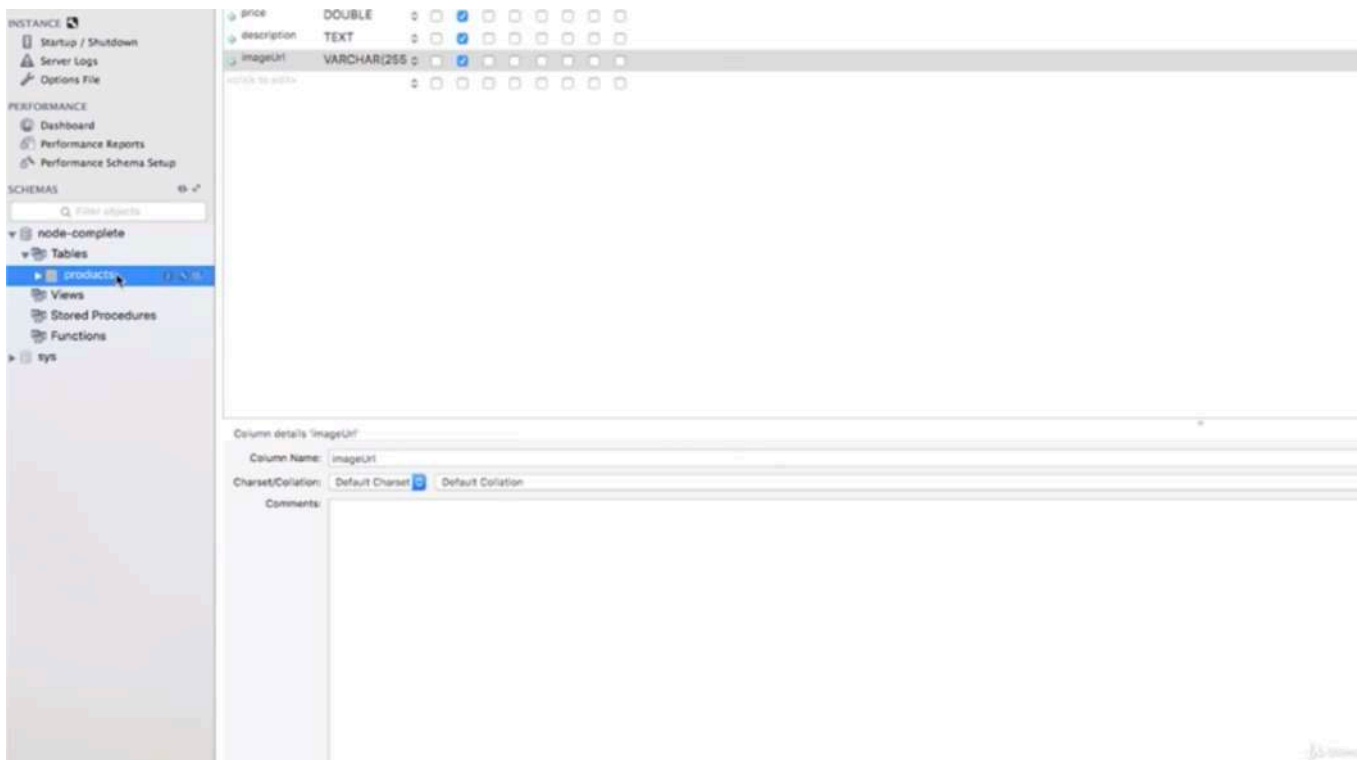




- it shows you the SQL statement. it will execute and you could execute this on your own, for example in node to always create this new table, here we will do it in the workbench.





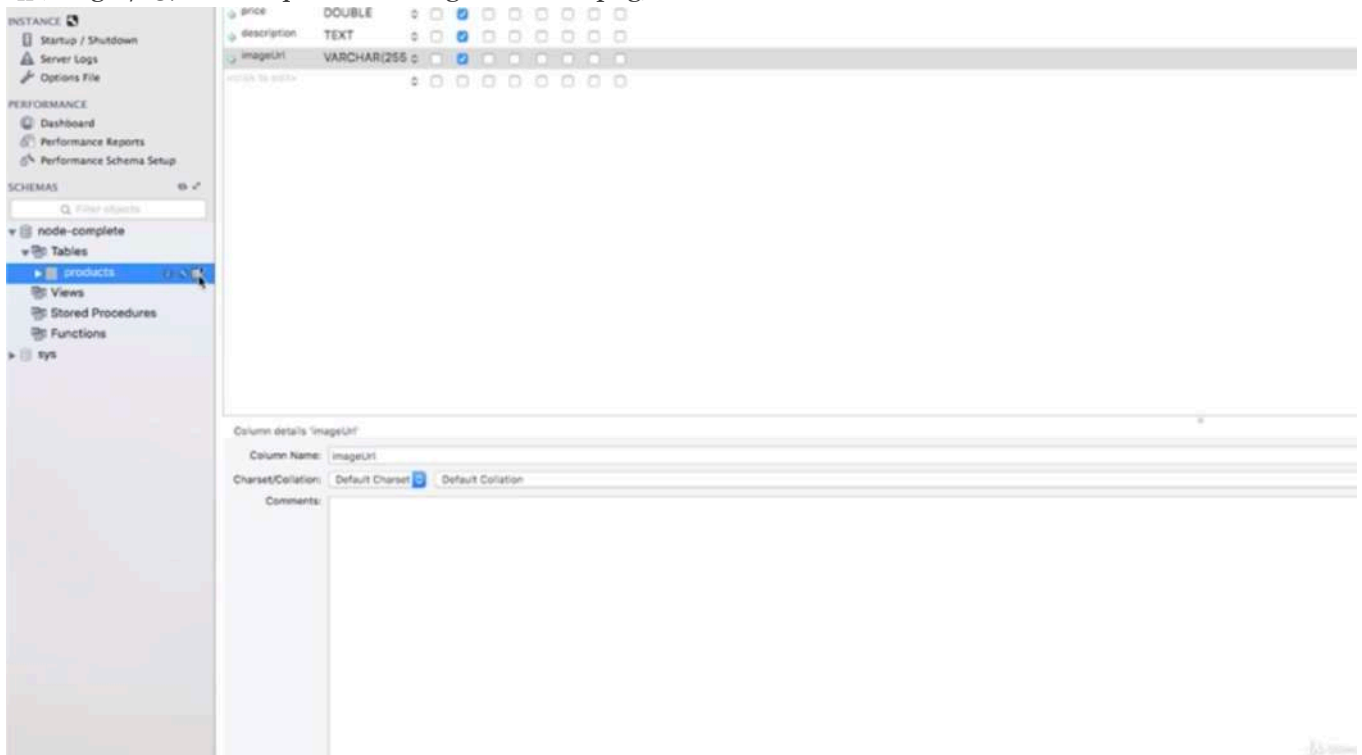


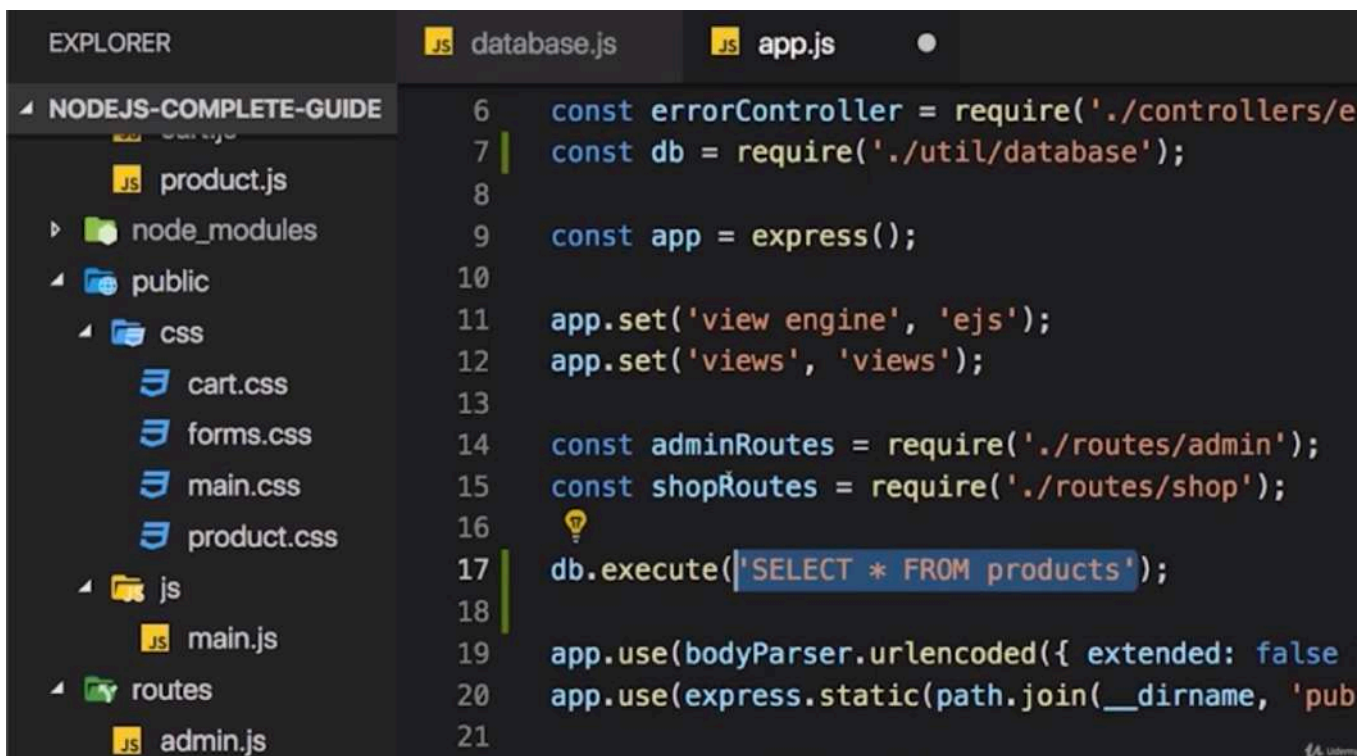
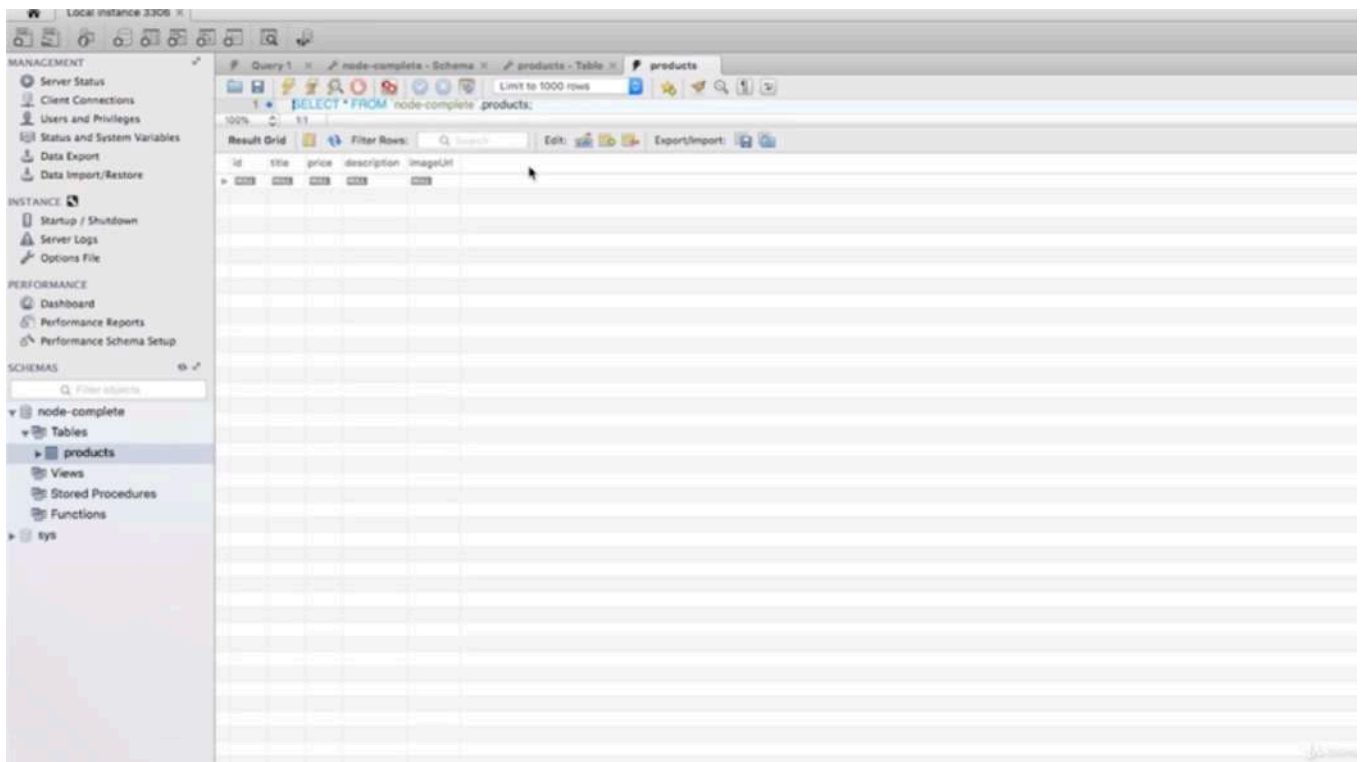
- after that, now on tables, you see the new products table







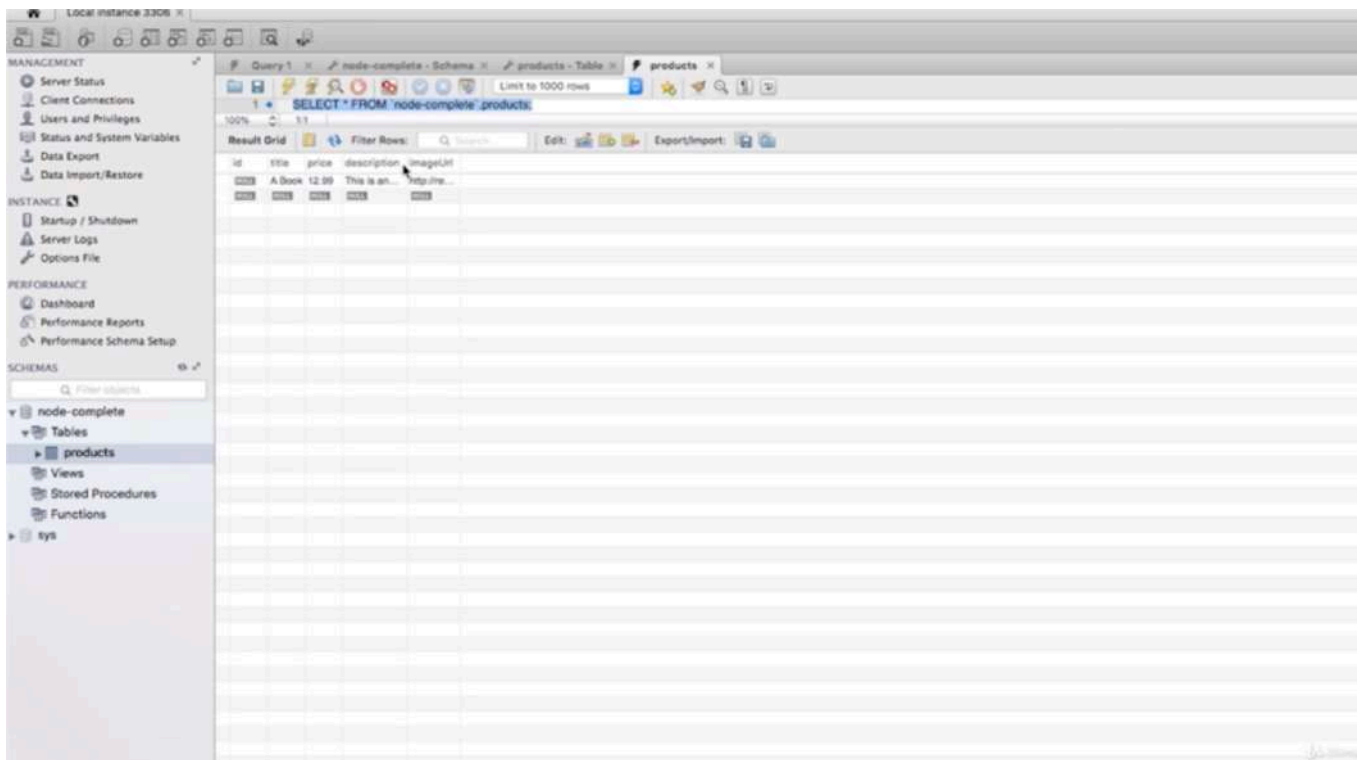




- and if you click this icon here, on the very right, you can see the entries in there.

- you also see the SQL query that was executed to look into that and that's pretty similar to the query we are executing here.



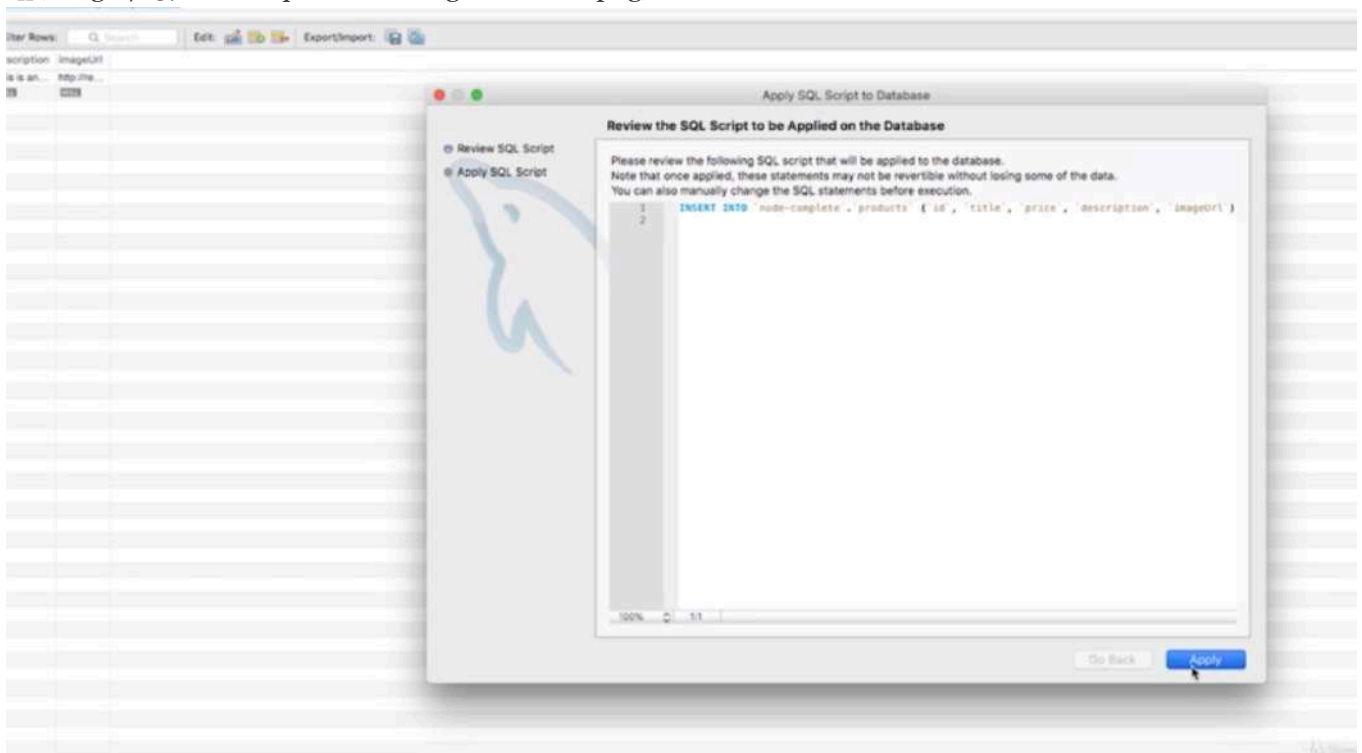


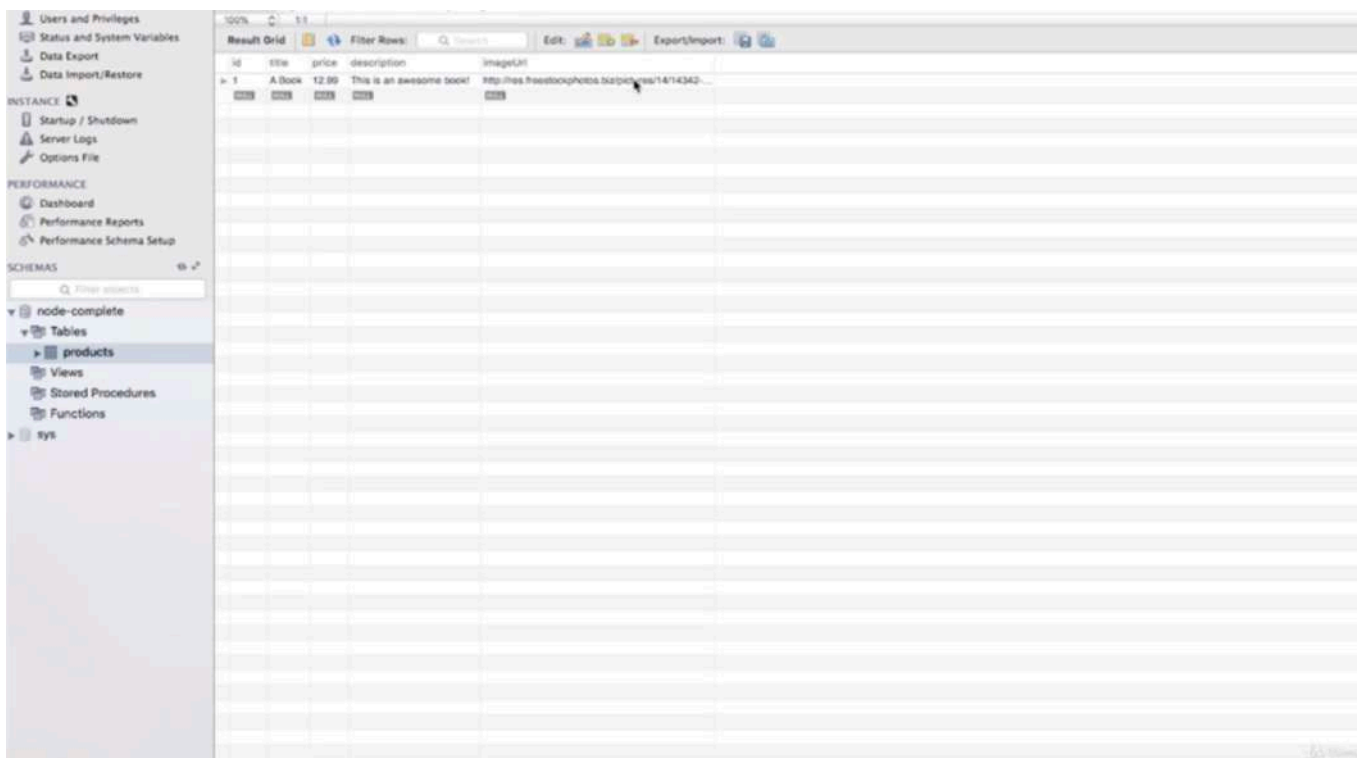
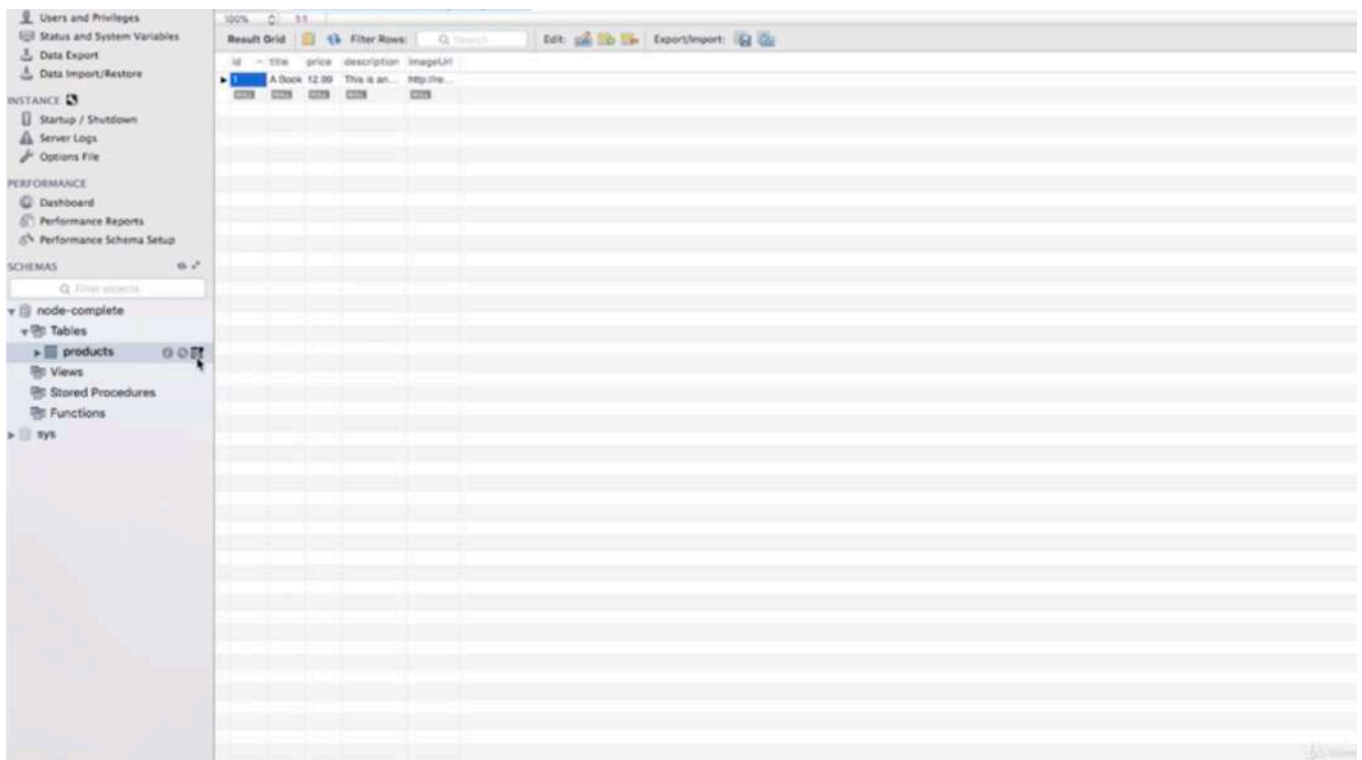
- now the table is set up, we need to enter one dummy data so that we have something to fetch.







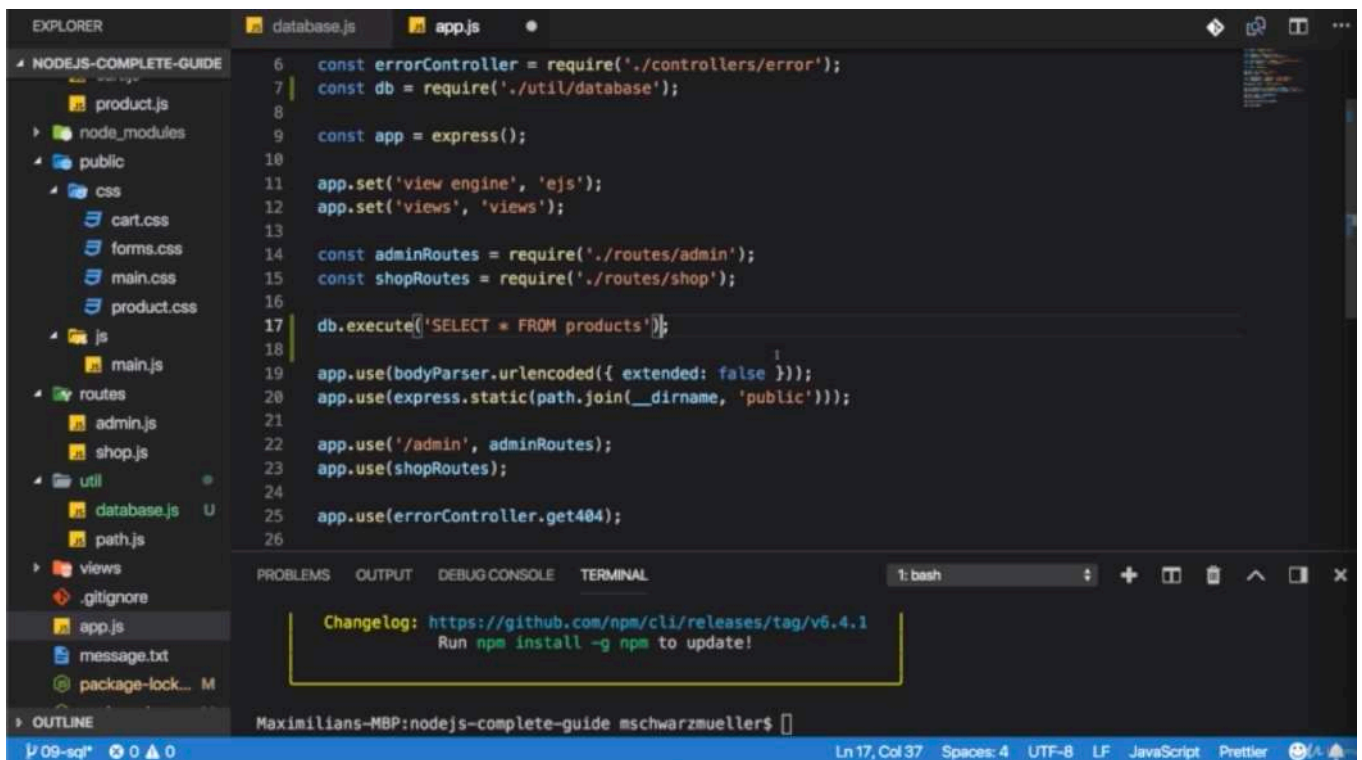




- now let's enter an ID that should be auto-generated. click 'apply' on the bottom right, apply and close and if you again click on this on the very right, you will see that now this one element was added here.

- now that we get a book in here





The screenshot shows a VS Code editor with a project named 'NODEJS-COMPLETE-GUIDE'. The file explorer on the left shows a directory structure with files like product.js, node\_modules, public, css, js, routes, admin.js, shop.js, util, database.js, path.js, views, .gitignore, app.js, message.txt, and package-lock... The main editor displays the contents of app.js, which includes database connection logic and route definitions. The code is as follows:

```
6 const errorController = require('./controllers/error');
7 const db = require('./util/database');
8
9 const app = express();
10
11 app.set('view engine', 'ejs');
12 app.set('views', 'views');
13
14 const adminRoutes = require('./routes/admin');
15 const shopRoutes = require('./routes/shop');
16
17 db.execute('SELECT * FROM products');
18
19 app.use(bodyParser.urlencoded({ extended: false }));
20 app.use(express.static(path.join(__dirname, 'public')));
21
22 app.use('/admin', adminRoutes);
23 app.use(shopRoutes);
24
25 app.use(errorController.get404);
26
```

The terminal at the bottom shows a message from npm: "Changelog: <https://github.com/npm/cli/releases/tag/v6.4.1> Run npm install -g npm to update!". The status bar at the bottom indicates the file is at line 17, column 37, with 4 spaces, UTF-8 encoding, LF line endings, and JavaScript syntax.

- let's go back to our node code.

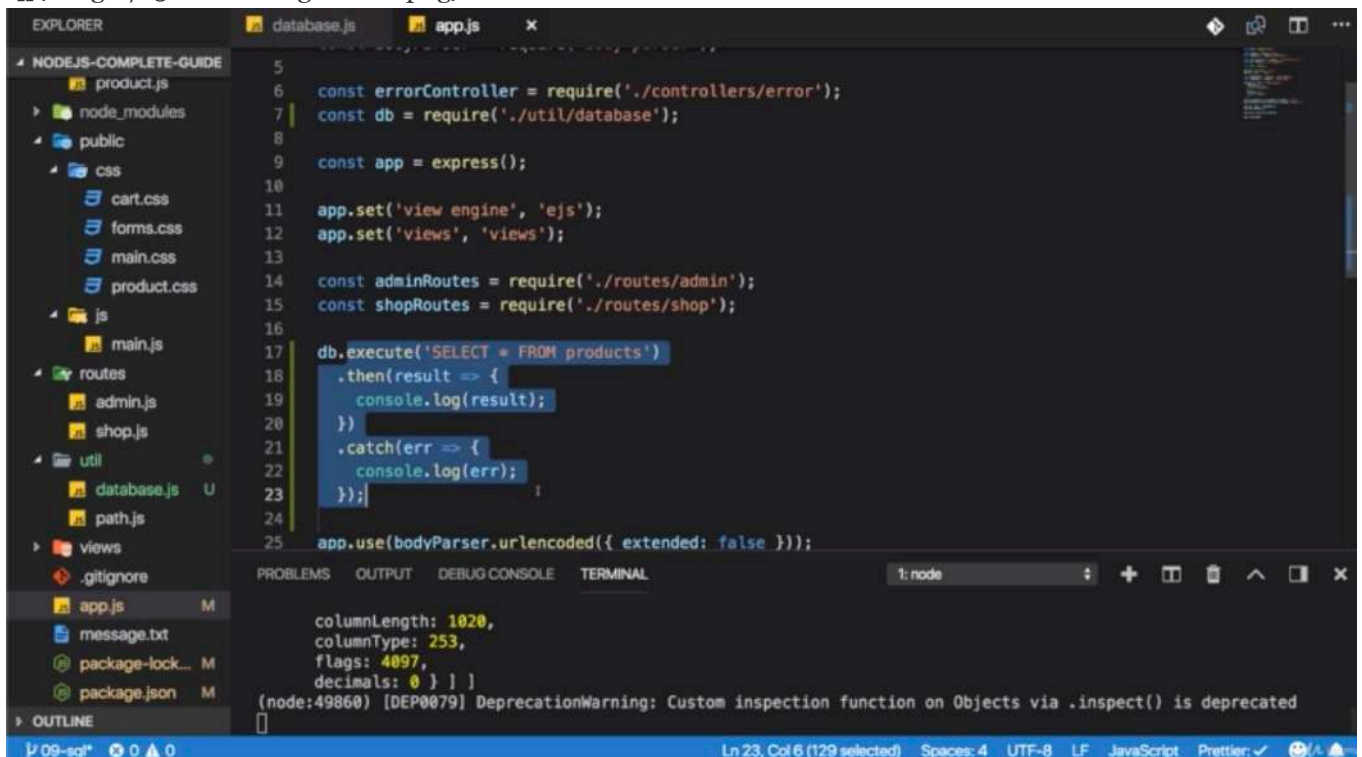
## \* Chapter 138: Retrieving Data

1. update

- app.js





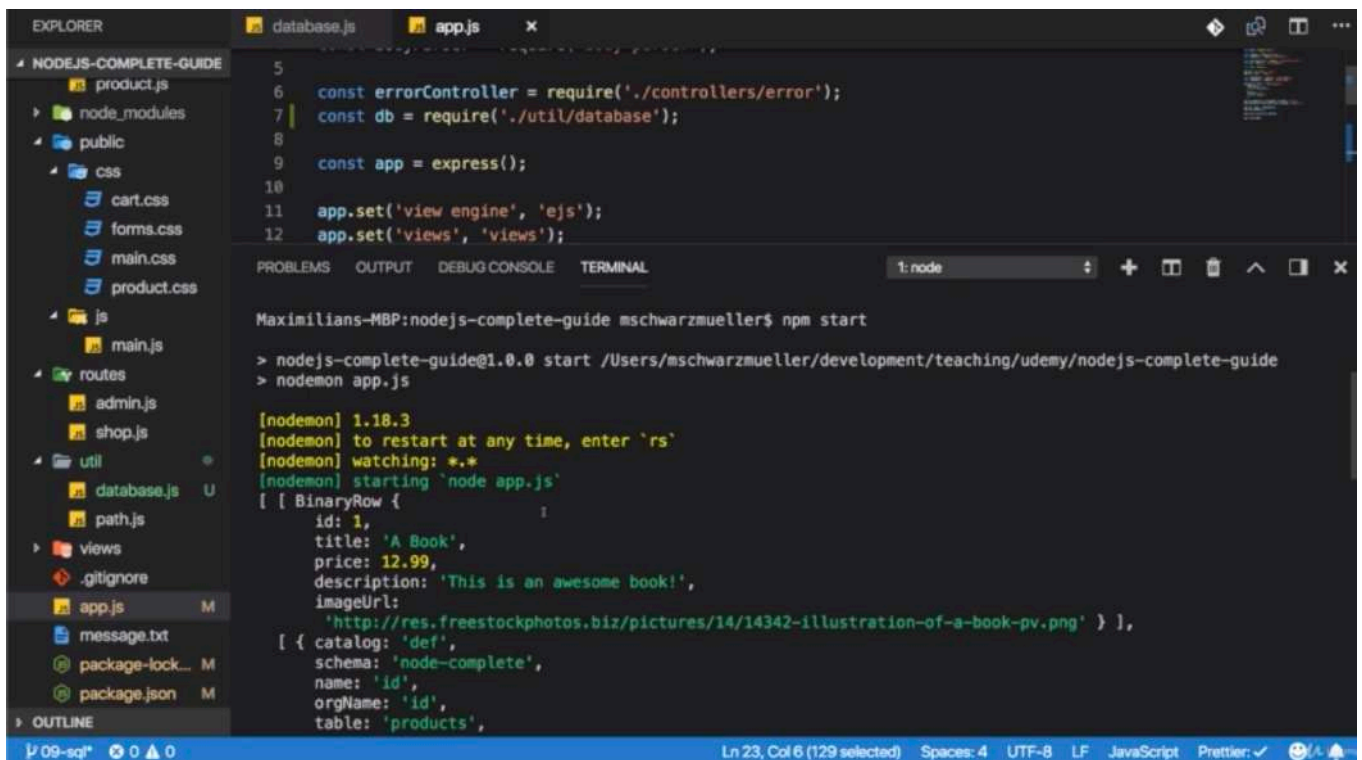


The screenshot shows the same VS Code editor with app.js. The code is updated to include a .then() callback to log the results of the database query. The code is as follows:

```
5
6 const errorController = require('./controllers/error');
7 const db = require('./util/database');
8
9 const app = express();
10
11 app.set('view engine', 'ejs');
12 app.set('views', 'views');
13
14 const adminRoutes = require('./routes/admin');
15 const shopRoutes = require('./routes/shop');
16
17 db.execute('SELECT * FROM products')
18   .then(result => {
19     console.log(result);
20   })
21   .catch(err => {
22     console.log(err);
23   });
24
25 app.use(bodyParser.urlencoded({ extended: false }));
```

The terminal at the bottom shows the output of the application: "columnLength: 1020, columnType: 253, flags: 4097, decimals: 0 ] ]". A deprecation warning is also visible: "(node:49860) [DEP0079] DeprecationWarning: Custom inspection function on Objects via .inspect() is deprecated". The status bar at the bottom indicates the file is at line 23, column 6 (129 selected), with 4 spaces, UTF-8 encoding, LF line endings, and JavaScript syntax.





The screenshot shows the VS Code editor with the `app.js` file open. The code defines an Express application that connects to a database and sets up a view engine. The terminal shows the command `npm start` being executed, which starts the application using nodemon. The output of the application is a JSON object representing a book product.

```
const errorController = require('./controllers/error');
const db = require('./util/database');

const app = express();

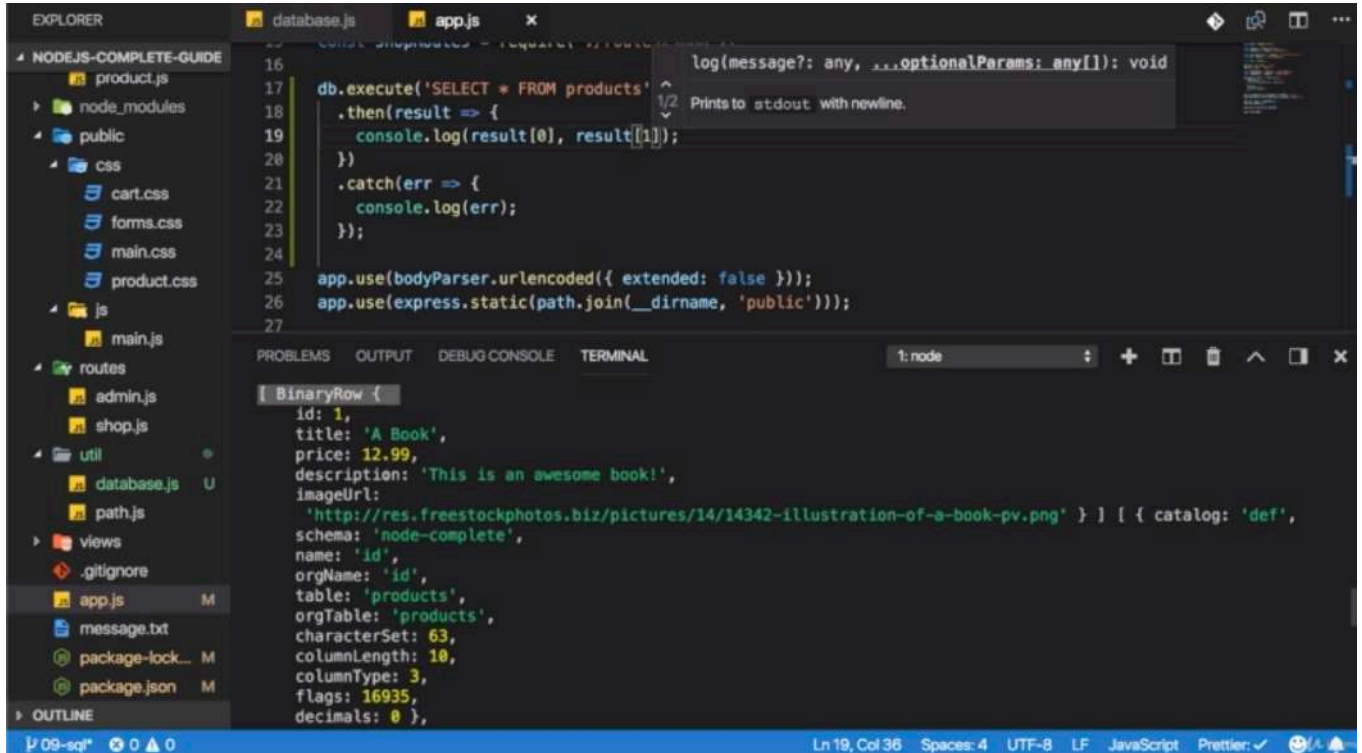
app.set('view engine', 'ejs');
app.set('views', 'views');
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ npm start
> nodejs-complete-guide@1.0.0 start /Users/mschwarzmueller/development/teaching/udemy/nodejs-complete-guide
> nodemon app.js

[nodemon] 1.18.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
[ [ BinaryRow {
  id: 1,
  title: 'A Book',
  price: 12.99,
  description: 'This is an awesome book!',
  imageUrl:
    'http://res.freestockphotos.biz/pictures/14/14342-illustration-of-a-book-pv.png' }, ],
  [ { catalog: 'def',
    schema: 'node-complete',
    name: 'id',
    orgName: 'id',
    table: 'products',
```

- this immediately executes because it's part of the `app.js` file
- if we have a look at this, we see this is the object we got back and in this object, we essentially see the data that was retrieved here.
- the data we get back has a format of an array with a nested array where the first nested array seems to depict our data, the rows it fetched and the second array seems to hold some metadata about the table we fetched it from





The screenshot shows the VS Code editor with the `app.js` file open. A `db.execute` call is added to the code, which logs the result of a SQL query. The terminal shows the output of the application, which is a JSON object representing a book product. A tooltip for the `log` function is visible over the `console.log` statement.

```
db.execute('SELECT * FROM products')
  .then(result => {
    console.log(result[0], result[1]);
  })
  .catch(err => {
    console.log(err);
  });

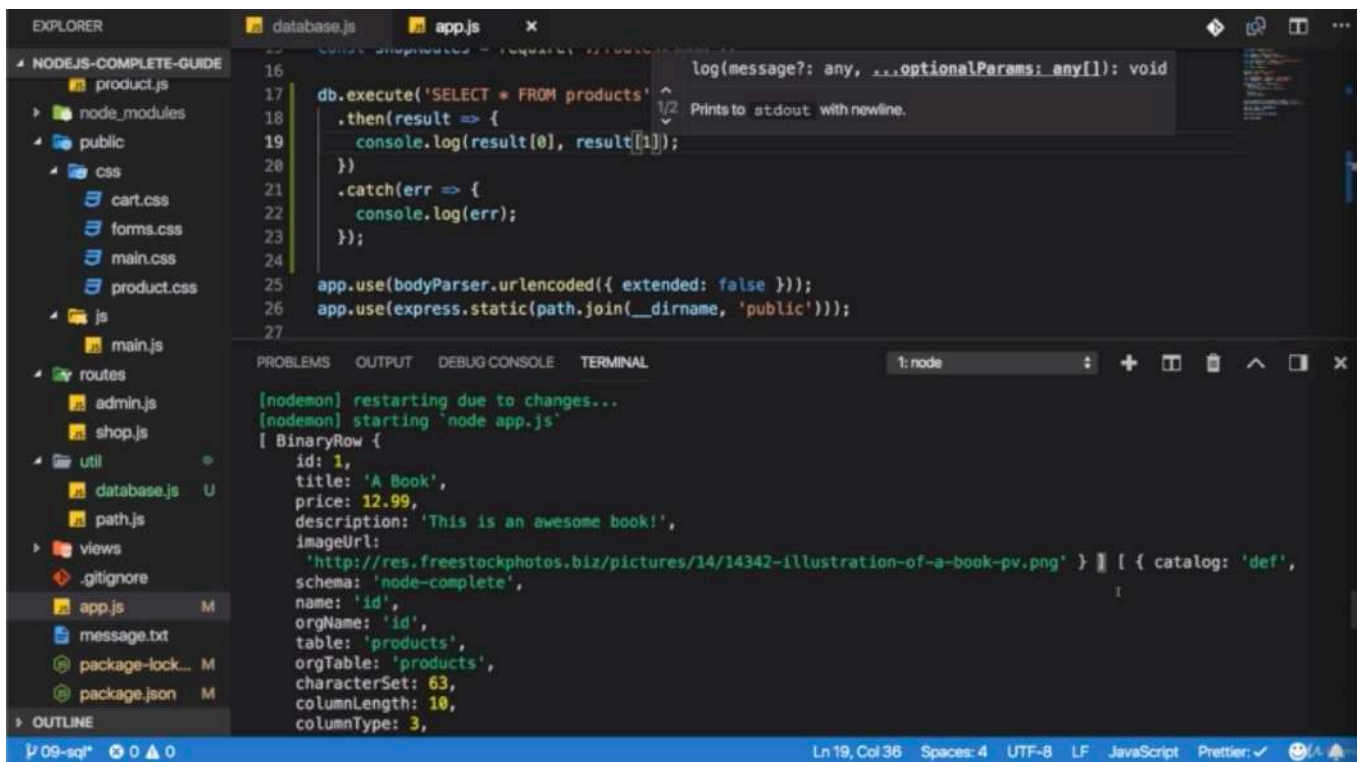
app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, 'public')));
```

```
[ BinaryRow {
  id: 1,
  title: 'A Book',
  price: 12.99,
  description: 'This is an awesome book!',
  imageUrl:
    'http://res.freestockphotos.biz/pictures/14/14342-illustration-of-a-book-pv.png' }, ],
  [ { catalog: 'def',
    schema: 'node-complete',
    name: 'id',
    orgName: 'id',
    table: 'products',
    orgTable: 'products',
    characterSet: 63,
    columnLength: 10,
    columnType: 3,
    flags: 16935,
    decimals: 0 },
```

- so `result` is an array with 2 nested elements. so we can log out `result[0]` and `result[1]`. and if we save this and therefore our server restarts, we have almost the same output, but now it's not a nested array but we have the first object we retrieved, the row we got and then this ends



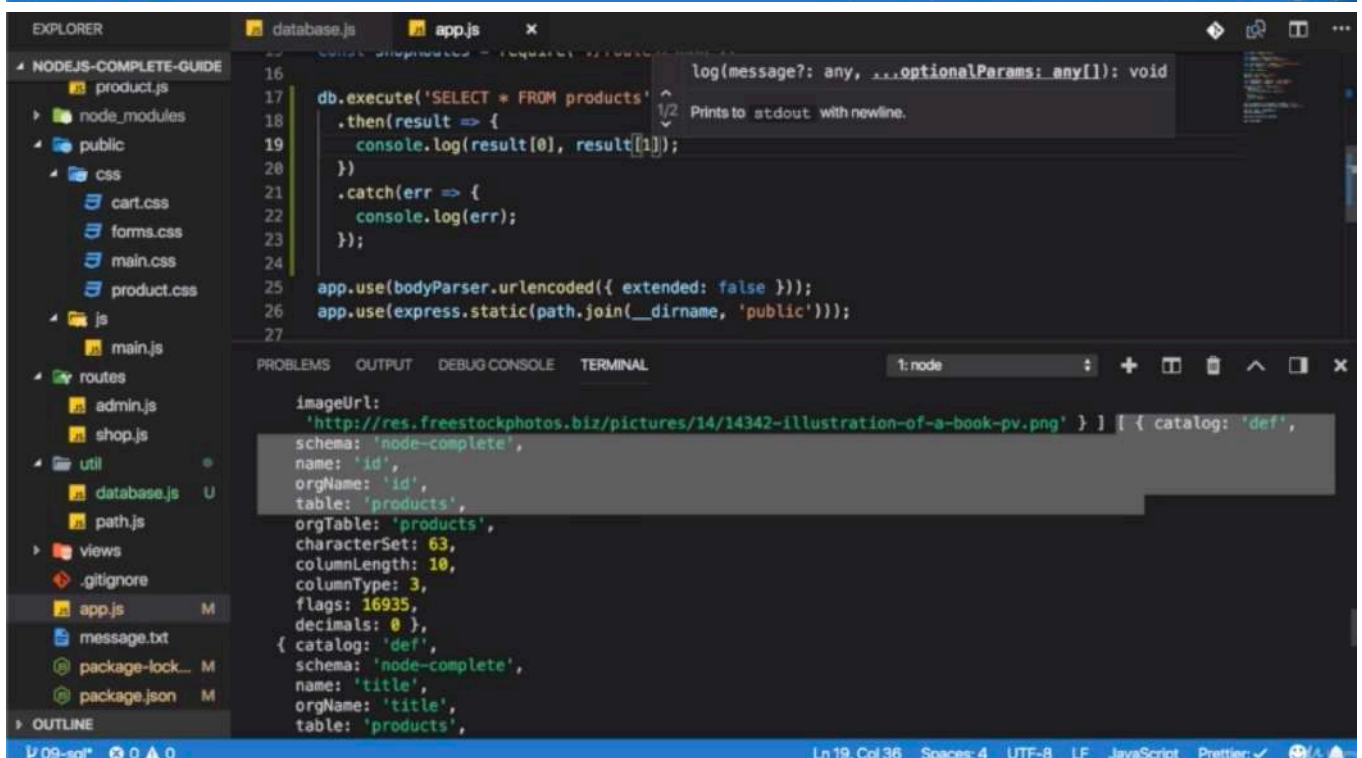




This screenshot shows the VS Code editor with a project named 'nodejs-complete-guide'. The file explorer on the left shows a directory structure including 'public', 'css', 'js', 'routes', and 'util'. The 'app.js' file is open in the editor, showing a database query and a log statement. The terminal at the bottom shows the output of the application, which is a JSON object representing a product from the database. The log statement in the code is `console.log(result[0], result[1]);`, and the terminal output shows the product details and its metadata.

```
16 const db = require('./util/database');
17
18 db.execute('SELECT * FROM products')
19   .then(result => {
20     console.log(result[0], result[1]);
21   })
22   .catch(err => {
23     console.log(err);
24   });
25
26 app.use(bodyParser.urlencoded({ extended: false }));
27 app.use(express.static(path.join(__dirname, 'public')));
```

```
[nodeemon] restarting due to changes...
[nodeemon] starting 'node app.js'
[ BinaryRow {
  id: 1,
  title: 'A Book',
  price: 12.99,
  description: 'This is an awesome book!',
  imageUrl:
    'http://res.freestockphotos.biz/pictures/14/14342-illustration-of-a-book-pv.png' } ] [ { catalog: 'def',
  schema: 'node-complete',
  name: 'id',
  orgName: 'id',
  table: 'products',
  orgTable: 'products',
  characterSet: 63,
  columnLength: 10,
  columnType: 3,
```



This screenshot shows the same VS Code editor as the first one, but the terminal now displays the second log output. The log statement in the code is `console.log(result[0], result[1]);`, and the terminal output shows the product details and its metadata. The log statement in the code is `console.log(result[0], result[1]);`, and the terminal output shows the product details and its metadata.

```
imageUrl:
  'http://res.freestockphotos.biz/pictures/14/14342-illustration-of-a-book-pv.png' } ] [ { catalog: 'def',
  schema: 'node-complete',
  name: 'id',
  orgName: 'id',
  table: 'products',
  orgTable: 'products',
  characterSet: 63,
  columnLength: 10,
  columnType: 3,
  flags: 16935,
  decimals: 0 },
{ catalog: 'def',
  schema: 'node-complete',
  name: 'title',
  orgName: 'title',
  table: 'products',
```

here is the closing square bracket and then here we get the second log, so this is the result one with the metadata.

```
1 //app.js
2
3 const path = require('path');
4
5 const express = require('express');
6 const bodyParser = require('body-parser');
7
8 const errorController = require('./controllers/error');
9 const db = require('./util/database');
10
11 const app = express();
12
13 app.set('view engine', 'ejs');
```



```

14 app.set('views', 'views');
15
16 const adminRoutes = require('./routes/admin');
17
18 const shopRoutes = require('./routes/shop');
19
20 /**we can chain 'then()'
21  * and this is something provided by the fact
22  * that we are using 'promise()' when exporting the pool in ./util/database.js
23  * we get back promises when executing queries like this with execute
24  * and promises have 2 functions 'then()' and 'catch()'
25  *
26  * these are functions we can chain onto the result of the execute call.
27  * they will execute on whatever this gives us back
28  * and this whatever is some so-called promise
29  *
30  * promise is a basic javascript object not specific to node
31  * it's available in javascript in the browser which allows us to work with asynchronous
  code.
32  * instead of using callbacks which we could also use with the MySQL package,
33  * promises allow us to write more structured code
34  * because instead of having nested anonymous function like having a second argument,
35  * we have 'then()' block which will then get the anonymous function to execute.
36  *     then(() => {})
37  */
38
39 /**we also have 'catch()'
40  * and this also has a function which executes in case of an error.
41  */
42 db.execute('SELECT * FROM products')
43   .then(result => {
44     console.log(result[0], result[1])
45   })
46   .catch(err => {
47     console.log(err)
48   })
49
50 app.use(bodyParser.urlencoded({ extended: false }));
51 app.use(express.static(path.join(__dirname, 'public')));
52
53 app.use('/admin', adminRoutes);
54 app.use(shopRoutes);
55
56 app.use(errorController.get404);
57
58 app.listen(3000);
59

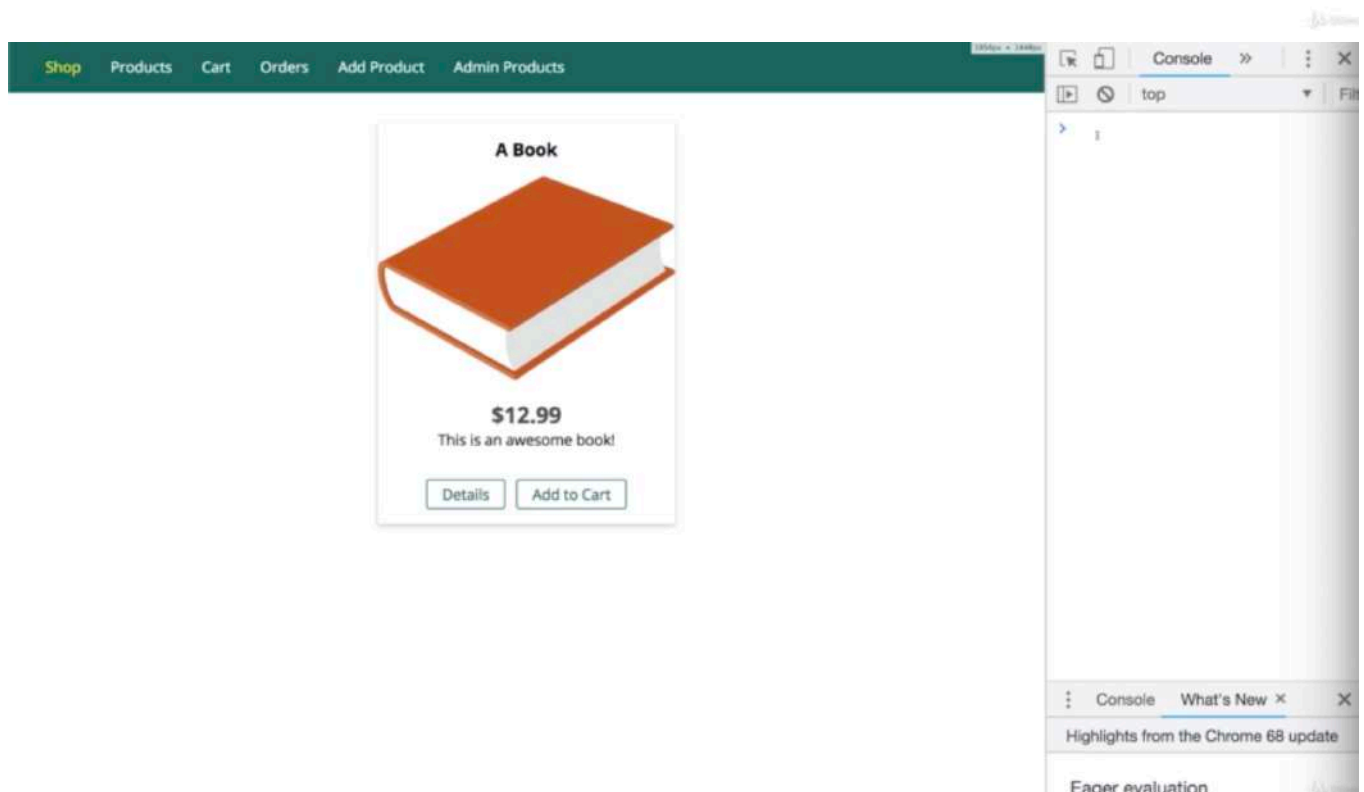
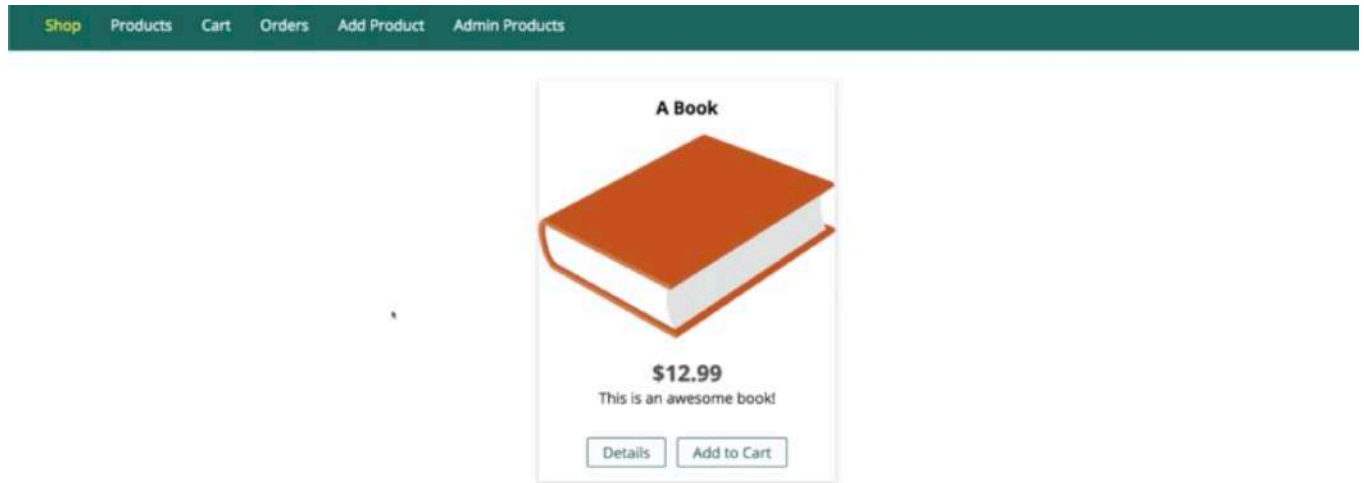
```

## \* Chapter 139: Fetching Products

1. update
  - ./models/product.js
  - ./controllers/shop.js







- if we go to localhost:3000, you can see the book here and also have no errors on our console. now we see that book because our data is retrieved from the database.









1 • **SELECT** \* FROM node-complete.products;

100% 1:1

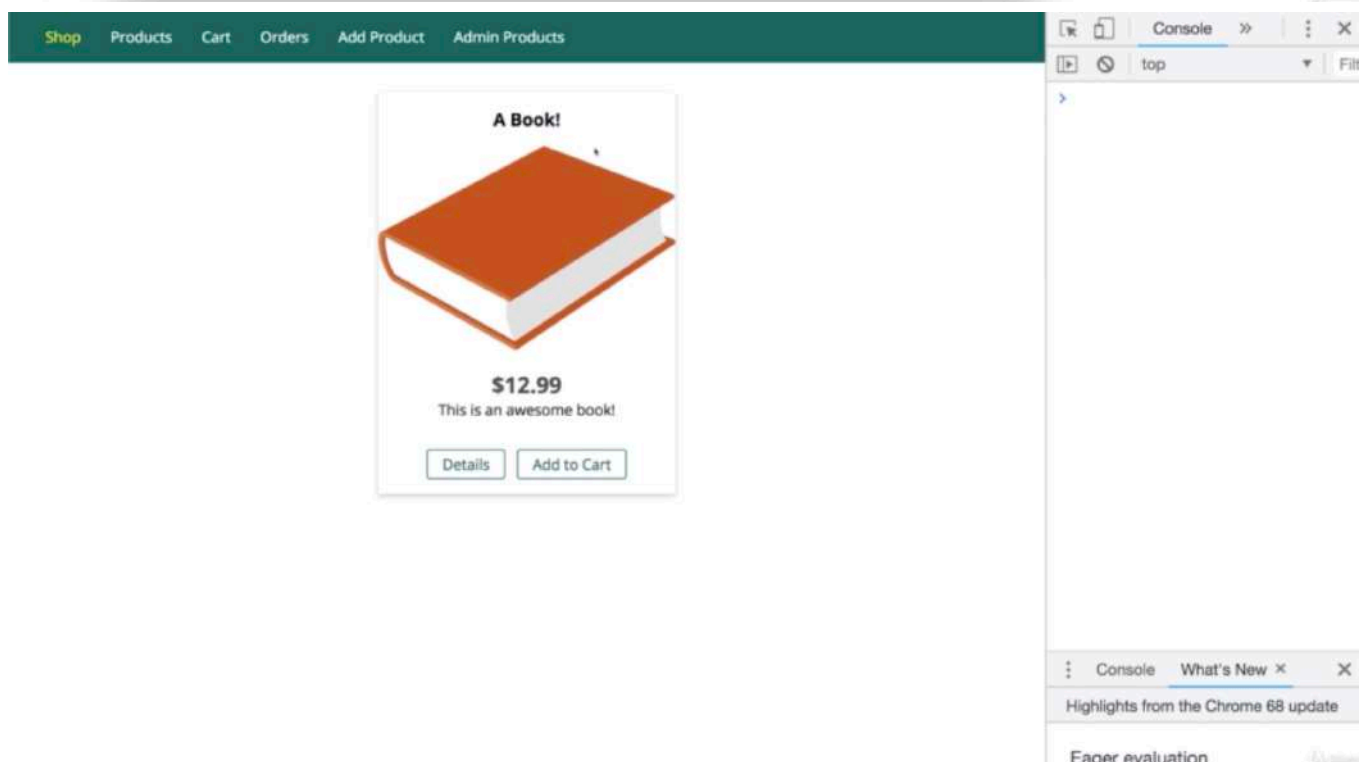
Result Grid Filter Rows: Search Edit: Export/Import:

id	title	price	description	imageUrl
1	A Book!	12.99	This is an awesome book!	http://res.freestockphotos.biz/pictures/14/14342-...

# The Role of the Teacher

1. The teacher is a facilitator of learning.
2. The teacher is a guide on the side.
3. The teacher is a participant-observer.
4. The teacher is a resource person.
5. The teacher is a learner.
6. The teacher is a co-learner.
7. The teacher is a co-facilitator.
8. The teacher is a co-learner.
9. The teacher is a co-facilitator.
10. The teacher is a co-learner.

Apply Revert



- if we were to go to the database and we add an exclamation mark(!) in the title, and click 'apply' and if i reload that, and i can see the change.

```

1 //./controllers/shop.js
2
3 const Product = require('../models/product');
4 const Cart = require('../models/cart');
5
6 exports.getProducts = (req, res, next) => {
7   Product.fetchAll(products => {
8     res.render('shop/product-list', {
9       prods: products,
10      pageTitle: 'All Products',
11      path: '/products'
12    });
13  });

```

```

13   });
14 };
15
16 exports.getProduct = (req, res, next) => {
17   const prodId = req.params.productId;
18   Product.findById(prodId, product => {
19     res.render('shop/product-detail', {
20       product: product,
21       pageTitle: product.title,
22       path: '/products'
23     });
24   });
25 };
26
27 exports.getIndex = (req, res, next) => {
28   /**we call 'fetchAll'
29    * but we still pass in a function that previously was the callback.
30    * but now we got no callback anymore, so let's take out that render code,
31    *
32    * 'fetchAll' will return a promise.
33    * so we can add 'then()' and 'catch()'
34    *
35   */
36   Product.fetchAll()
37   /**'rows' will be the first element of the nested array
38    * which would be our argument data
39    * and 'fieldData' will be the second element
40    *
41    * we can use these 2 variables which holds these 2 nested arrays
42   */
43   .then(([rows, fieldData]) => {
44     res.render('shop/index', {
45       /**'rows' should be my products
46        * because my rows are the entries in the products table
47        * and therefore these should be my products.
48       */
49       prods: rows,
50       pageTitle: 'Shop',
51       path: '/'
52     });
53   })
54   .catch(err => console.log(err))
55 };
56
57 exports.getCart = (req, res, next) => {
58   Cart.getCart(cart => {
59     Product.fetchAll(products => {
60       const cartProducts = [];
61       for (product of products) {
62         const cartProductData = cart.products.find(
63           prod => prod.id === product.id
64         );
65         if (cartProductData) {
66           cartProducts.push({ productData: product, qty: cartProductData.qty });
67         }
68       }

```

```

69     res.render('shop/cart', {
70         path: '/cart',
71         pageTitle: 'Your Cart',
72         products: cartProducts
73     });
74 });
75 });
76 };
77
78 exports.postCart = (req, res, next) => {
79     const prodId = req.body.productId;
80     Product.findById(prodId, product => {
81         Cart.addProduct(prodId, product.price);
82     });
83     res.redirect('/cart');
84 };
85
86 exports.postCartDeleteProduct = (req, res, next) => {
87     const prodId = req.body.productId;
88     Product.findById(prodId, product => {
89         Cart.deleteProduct(prodId, product.price);
90         res.redirect('/cart');
91     });
92 };
93
94 exports.getOrders = (req, res, next) => {
95     res.render('shop/orders', {
96         path: '/orders',
97         pageTitle: 'Your Orders'
98     });
99 };
100
101 exports.getCheckout = (req, res, next) => {
102     res.render('shop/checkout', {
103         path: '/checkout',
104         pageTitle: 'Checkout'
105     });
106 };
107

```

```

1  //./models/product.js
2
3  const db = require('../util/database')
4
5  const Cart = require('./cart');
6
7
8  module.exports = class Product {
9      constructor(id, title, imageUrl, description, price) {
10         this.id = id;
11         this.title = title;
12         this.imageUrl = imageUrl;
13         this.description = description;
14         this.price = price;
15     }
16
17     save() {

```

```

18
19 }
20
21 static deleteById(id) {
22
23 }
24
25 static fetchAll() {
26     /** '*' star stands for 'everything' in SQL
27     * you could write 'SELECT' and 'FROM' in lowercase too like 'select' and 'from'
28     * but i like to keep these keyword uppercase to indicate what is core SQL syntax
29     * and what are our dynamic values.
30     */
31     /** i'm interested in the returned value in the place where i'm calling fetchAll
32     * so i will simply return the entire promise that execute returns
33     * so that we can use it somewhere else.
34     *
35     */
36     return db.execute('SELECT * FROM products')
37 }
38
39 static findById(id, ) {
40
41 };
42 }

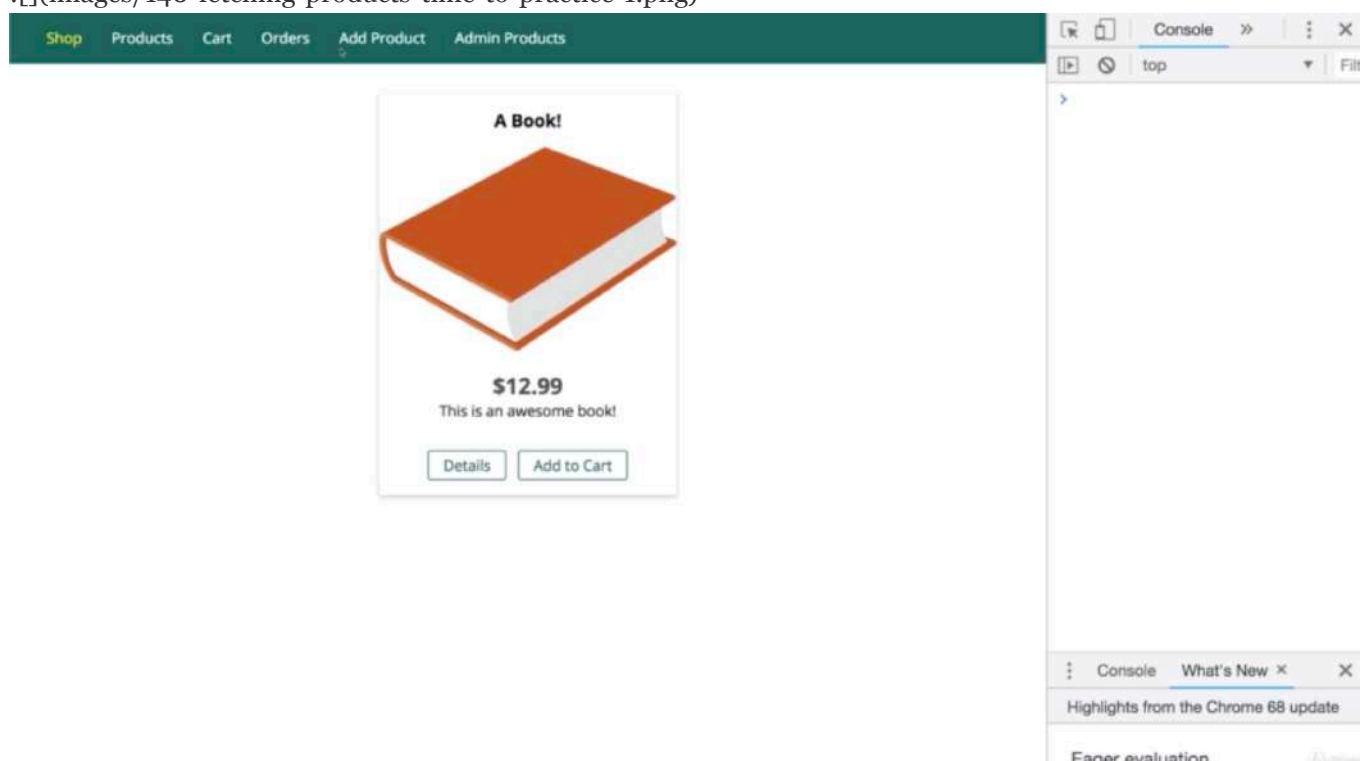
```

## \* Chapter 140: Fetching Products - Time To Practice

1. update

- ./controllers/shop.js







- products page also work well again.

```
1  //./controllers/shop.js
2
3  const Product = require('../models/product');
4  const Cart = require('../models/cart');
5
6  exports.getProducts = (req, res, next) => {
7    Product.fetchAll()
8    /**you can get rid of 'fieldData'
9     * you don't need to extract that
10    * because we are not using it.
11    * i just wanna show you
12    * how you can extract the different elements of an array in the argument list already
13    */
14    .then([rows, fieldData]) => {
15      res.render('shop/product-list', {
16        prods: rows,
17        pageTitle: 'All Products',
18        path: '/products'
19      })
20    }
21  };
22
23  exports.getProduct = (req, res, next) => {
24    const prodId = req.params.productId;
25    Product.findById(prodId, product => {
26      res.render('shop/product-detail', {
27        product: product,
28        pageTitle: product.title,
29        path: '/products'
30      });
31    });
32  };
33
34  exports.getIndex = (req, res, next) => {
35    Product.fetchAll()
36    .then([rows, fieldData]) => {
37      res.render('shop/index', {
38        prods: rows,
39        pageTitle: 'Shop',
40        path: '/'
41      })
42    }
43    .catch(err => console.log(err))
44  };
45
46  exports.getCart = (req, res, next) => {
47    Cart.getCart(cart => {
48      Product.fetchAll(products => {
49        const cartProducts = [];
50        for (product of products) {
51          const cartProductData = cart.products.find(
52            prod => prod.id === product.id
53          );
54          if (cartProductData) {
```

```

55         cartProducts.push({ productData: product, qty: cartProductData.qty });
56     }
57 }
58 res.render('shop/cart', {
59     path: '/cart',
60     pageTitle: 'Your Cart',
61     products: cartProducts
62 });
63 });
64 });
65 };
66
67 exports.postCart = (req, res, next) => {
68     const prodId = req.body.productId;
69     Product.findById(prodId, product => {
70         Cart.addProduct(prodId, product.price);
71     });
72     res.redirect('/cart');
73 };
74
75 exports.postCartDeleteProduct = (req, res, next) => {
76     const prodId = req.body.productId;
77     Product.findById(prodId, product => {
78         Cart.deleteProduct(prodId, product.price);
79         res.redirect('/cart');
80     });
81 };
82
83 exports.getOrders = (req, res, next) => {
84     res.render('shop/orders', {
85         path: '/orders',
86         pageTitle: 'Your Orders'
87     });
88 };
89
90 exports.getCheckout = (req, res, next) => {
91     res.render('shop/checkout', {
92         path: '/checkout',
93         pageTitle: 'Checkout'
94     });
95 };
96

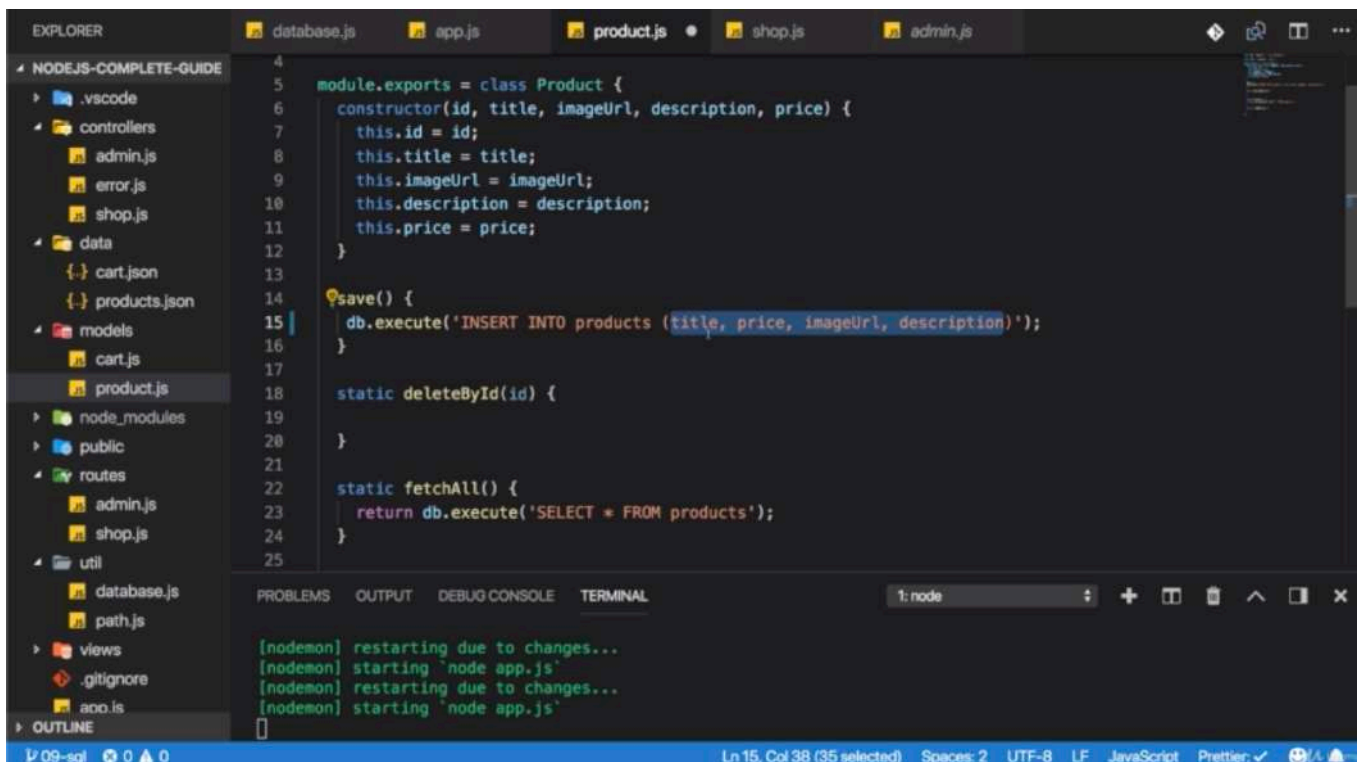
```

## \* Chapter 141: Inserting Data Into The Database

1. update
  - ./models/product.js
  - ./controllers/admin.js





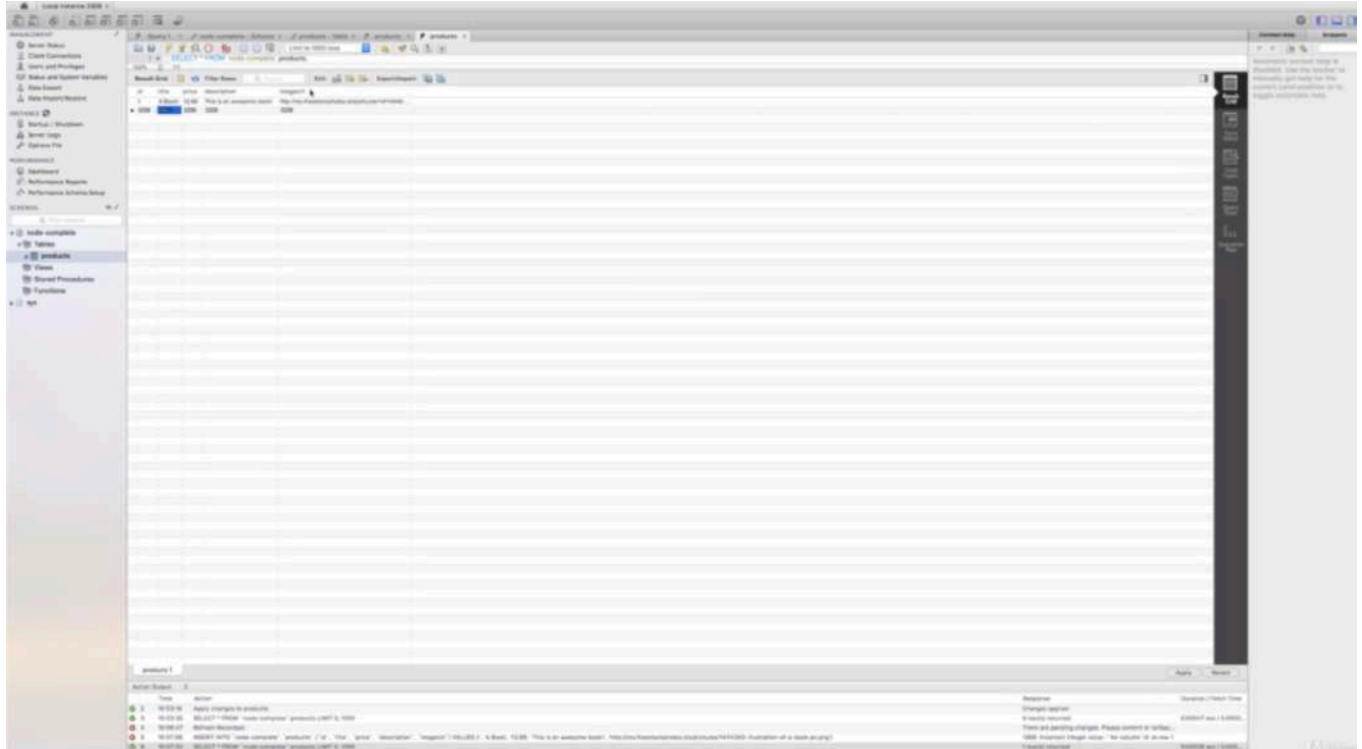


The screenshot shows the VS Code editor with the Explorer sidebar on the left displaying a project structure. The main editor area shows the `product.js` file with the following code:

```
4 module.exports = class Product {
5   constructor(id, title, imageUrl, description, price) {
6     this.id = id;
7     this.title = title;
8     this.imageUrl = imageUrl;
9     this.description = description;
10    this.price = price;
11  }
12
13  save() {
14    db.execute('INSERT INTO products (title, price, imageUrl, description)');
15  }
16
17  static deleteById(id) {
18
19  }
20
21  static fetchAll() {
22    return db.execute('SELECT * FROM products');
23  }
24 }
25
```

The terminal window at the bottom shows the following output:

```
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
```



- we have the title, the price, the imageUrl and the description and the important, you need to make sure that the field you define here match the field names you defined in your table, in the database.
  - you don't need to specify the ID because that should be generated automatically by the database engine.
- 
- 

The image displays two screenshots of a web application interface, likely a shopping cart or product management system, along with a Chrome DevTools console window.

**Top Screenshot (Add Product Form):**

- Navigation Bar:** Shop, Products, Cart, Orders, **Add Product** (highlighted), Admin Products.
- Form Fields:**
  - Title: A second Product
  - Image URL: fadsfasdf
  - Price: 9.99
  - Description: Some description!
- Action:** Add Product button.

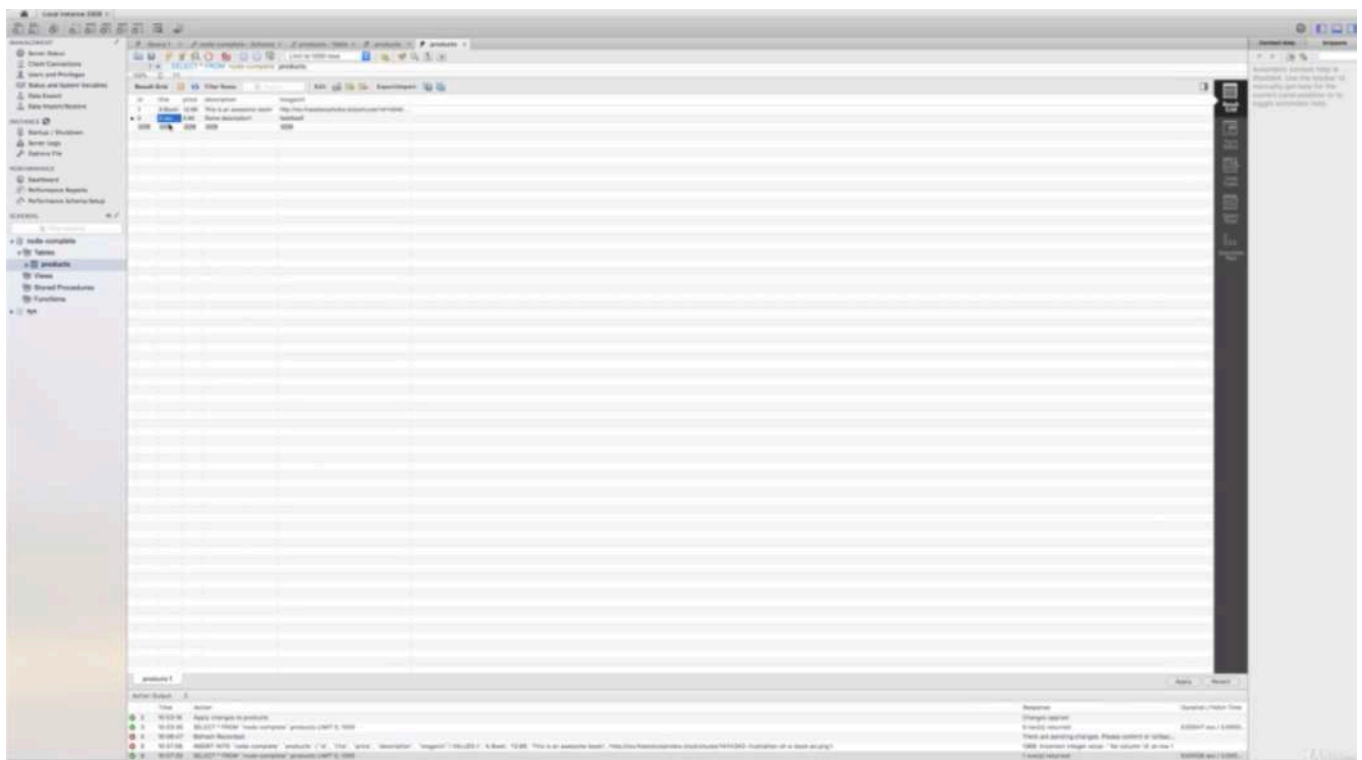
**Bottom Screenshot (Product Listings):**

- Navigation Bar:** Shop, Products, Cart, Orders, Add Product, Admin Products.
- Product 1:**
  - Title: A Book!
  - Image: A 3D rendering of an orange book.
  - Price: \$12.99
  - Description: This is an awesome book!
  - Buttons: Details, Add to Cart
- Product 2:**
  - Title: A second Product
  - Image: A small placeholder icon.
  - Price: \$9.99
  - Description: Some description!
  - Buttons: Details, Add to Cart

**Chrome DevTools Console:**

- Top Console:** Shows a 404 error: `GET http://localhost:99 alhost:3000/fadsfasdf 404 (Not Found)`.
- Bottom Console:** Shows a 404 error: `GET http://localhost:99 alhost:3000/fadsfasdf 404 (Not Found)`.

- if we click 'Add Product' button, we are redirected and this is looking good.  

- and if we have a look at our database and click that refresh button here, we see our entry, so our new entry with an auto-generated ID.

```

1  //./models/product.js
2
3  const db = require('../util/database');
4
5  const Cart = require('./cart');
6
7  module.exports = class Product {
8    constructor(id, title, imageUrl, description, price) {
9      this.id = id;
10     this.title = title;
11     this.imageUrl = imageUrl;
12     this.description = description;
13     this.price = price;
14   }
15
16   save() {
17     /** with SQL, we saw select for getting data,
18     * for inserting data, there is the 'INSERT INTO' command
19     * and we then define the table where we wanna insert something
20     * and i will use the 'products' table here, followed by brackets
21     * where we list the different fields we wanna insert value into
22     * so we have the title, price, imageUrl, description
23     * and important, you need to make sure
24     * that the fields you define here match the field names you defined in your table, in
the database.
25     * and you don't need to specify the ID
26     * because that should be generated automatically by the database engine.
27     *
28     * we need the 'VALUE' keyword followed by brackets with the VALUES
29     * to safely insert values and not face the issue of SQL injection
30     * which is an attack pattern
31     * where users can insert special data into your input fields in your web page that runs

```

```

as SQL queries,
32     * we should use an approach where we just use question marks,
33     * one for each of the fields we insert data into
34     * separated with commas,
35     * and then there is a second argument we pass to execute with the VALUES that will be
    injected
36     * instead of these question marks
37     * so the order of the elements we add to this array is the order of arguments.
38     *
39     * and again i will simply return the promise that execute yields
40     * that allows us to go back to the admin.js file to the controller
41     */
42     return db.execute(
43         'INSERT INTO products (title, price, imageUrl, description) VALUES (?, ?, ?, ?)',
44         [this.title, this.price, this.imageUrl, this.description]
45     );
46 }
47
48 static deleteById(id) {}
49
50 static fetchAll() {
51     return db.execute('SELECT * FROM products');
52 }
53
54 static findById(id) {}
55 };
56

```

```

1 // ./controllers/admin.js
2
3 const Product = require('../models/product');
4
5 exports.getAddProduct = (req, res, next) => {
6     res.render('admin/edit-product', {
7         pageTitle: 'Add Product',
8         path: '/admin/add-product',
9         editing: false
10    });
11 };
12
13 exports.postAddProduct = (req, res, next) => {
14     const title = req.body.title;
15     const imageUrl = req.body.imageUrl;
16     const price = req.body.price;
17     const description = req.body.description;
18     const product = new Product(null, title, imageUrl, description, price);
19     product
20         .save()
21         .then(() => {
22             res.redirect('/');
23         })
24         .catch(err => console.log(err));
25 };
26
27 exports.getEditProduct = (req, res, next) => {
28     const editMode = req.query.edit;
29     if (!editMode) {

```

```

30     return res.redirect('/');
31 }
32 const prodId = req.params.productId;
33 Product.findById(prodId, product => {
34     if (!product) {
35         return res.redirect('/');
36     }
37     res.render('admin/edit-product', {
38         pageTitle: 'Edit Product',
39         path: '/admin/edit-product',
40         editing: editMode,
41         product: product
42     });
43 });
44 };
45
46 exports.postEditProduct = (req, res, next) => {
47     const prodId = req.body.productId;
48     const updatedTitle = req.body.title;
49     const updatedPrice = req.body.price;
50     const updatedImageUrl = req.body.imageUrl;
51     const updatedDesc = req.body.description;
52     const updatedProduct = new Product(
53         prodId,
54         updatedTitle,
55         updatedImageUrl,
56         updatedDesc,
57         updatedPrice
58     );
59     updatedProduct.save();
60     res.redirect('/admin/products');
61 };
62
63 exports.getProducts = (req, res, next) => {
64     Product.fetchAll(products => {
65         res.render('admin/products', {
66             prods: products,
67             pageTitle: 'Admin Products',
68             path: '/admin/products'
69         });
70     });
71 };
72
73 exports.postDeleteProduct = (req, res, next) => {
74     const prodId = req.body.productId;
75     Product.deleteById(prodId);
76     res.redirect('/admin/products');
77 };
78

```

## \* Chapter 142: Fetching A Single Product With The “Where” Condition

1. update
- ./models/product.js



- ./controllers/shop.js






The screenshot shows a web application with a dark green navigation bar containing links: Shop, Products, Cart, Orders, Add Product, and Admin Products. The main content area displays two product cards. The first card, titled 'A Book!', features an orange book image, a price of \$12.99, and the description 'This is an awesome book!'. The second card, titled 'A second Product', has a placeholder image, a price of \$9.99, and the description 'Some description!'. Both cards include 'Details' and 'Add to Cart' buttons. The browser's developer console on the right shows a 404 error for the second product's image URL.

Shop Products Cart Orders Add Product Admin Products

**A Book!**




**\$12.99**

This is an awesome book!

Details Add to Cart

**A second Product**



**\$9.99**

Some description!

Details Add to Cart

GET http://localhost:99 alhost:3000/fadsfasdf 404 (Not Found)


Console What's New x

Highlights from the Chrome 68 update

Eager evaluation

Shop Products Cart Orders Add Product Admin Products

**A second Product**



**9.99**

Some description!

Add to Cart

GET http://localhost:30 2:70 00/products/fadsfasdf 500 (Internal Server Error)

Console What's New x

Highlights from the Chrome 68 update

Eager evaluation

```
1 //./models/product.js
2
3 const db = require('../util/database');
4
5 const Cart = require('./cart');
6
7 module.exports = class Product {
8   constructor(id, title, imageUrl, description, price) {
9     this.id = id;
10    this.title = title;
```

```

11     this.imageUrl = imageUrl;
12     this.description = description;
13     this.price = price;
14 }
15
16 save() {
17     return db.execute(
18         'INSERT INTO products (title, price, imageUrl, description) VALUES (?, ?, ?, ?)',
19         [this.title, this.price, this.imageUrl, this.description]
20     );
21 }
22
23 static deleteById(id) {}
24
25 static fetchAll() {
26     return db.execute('SELECT * FROM products');
27 }
28
29 static findById(id) {
30     /** everything '*' means not all rows but simply all fields
31     * but now we can restrict the number of rows with a 'WHERE' condition
32     * and 'WHERE' is another SQL keyword.
33     * so we can execute 'WHERE products.id = '
34     * one equals sign only, not multiple ones as in javascript
35     * 'WHERE products.id' is equal to question mark
36     * simply to let my MySQL inject the value again,
37     * the ID we are getting as an argument
38     *
39     * let's return this promise here.
40     * and this is our statement for fetching a single product with all the columns though.
41     * so with all the data.
42     * now we can go back to ./controllers/shop.js
43     */
44
45     return db.execute('SELECT * FROM products WHERE products.id = ?', [id]);
46 }
47 };
48

```

```

1 //./controllers/shop.js
2
3 const Product = require('../models/product');
4 const Cart = require('../models/cart');
5
6 exports.getProducts = (req, res, next) => {
7     Product.fetchAll()
8         .then(([rows, fieldData]) => {
9             res.render('shop/product-list', {
10                 prods: rows,
11                 pageTitle: 'All Products',
12                 path: '/products'
13             });
14         })
15         .catch(err => console.log(err));
16 };
17
18 exports.getProduct = (req, res, next) => {

```

```

19  const prodId = req.params.productId;
20  /**we got that nested array
21   * where we know that the first element will be all the rows we got
22   * and that will just be our product
23   * or it should just be the product
24   *
25   * make sure to wrap that special syntax with the square brackets in the parentheses
26   */
27  Product.findById(prodId)
28    .then([product] => {
29      res.render('shop/product-detail', {
30        product: product[0],
31        pageTitle: product.title,
32        path: '/products'
33      });
34    })
35    .catch(err => console.log(err));
36  };
37
38  exports.getIndex = (req, res, next) => {
39    Product.fetchAll()
40      .then([rows, fieldData] => {
41        res.render('shop/index', {
42          prods: rows,
43          pageTitle: 'Shop',
44          path: '/'
45        });
46      })
47      .catch(err => console.log(err));
48  };
49
50  exports.getCart = (req, res, next) => {
51    Cart.getCart(cart => {
52      Product.fetchAll(products => {
53        const cartProducts = [];
54        for (product of products) {
55          const cartProductData = cart.products.find(
56            prod => prod.id === product.id
57          );
58          if (cartProductData) {
59            cartProducts.push({ productData: product, qty: cartProductData.qty });
60          }
61        }
62        res.render('shop/cart', {
63          path: '/cart',
64          pageTitle: 'Your Cart',
65          products: cartProducts
66        });
67      });
68    });
69  };
70
71  exports.postCart = (req, res, next) => {
72    const prodId = req.body.productId;
73    Product.findById(prodId, product => {
74      Cart.addProduct(prodId, product.price);

```

```
75   });
76   res.redirect('/cart');
77 };
78
79 exports.postCartDeleteProduct = (req, res, next) => {
80   const prodId = req.body.productId;
81   Product.findById(prodId, product => {
82     Cart.deleteProduct(prodId, product.price);
83     res.redirect('/cart');
84   });
85 };
86
87 exports.getOrders = (req, res, next) => {
88   res.render('shop/orders', {
89     path: '/orders',
90     pageTitle: 'Your Orders'
91   });
92 };
93
94 exports.getCheckout = (req, res, next) => {
95   res.render('shop/checkout', {
96     path: '/checkout',
97     pageTitle: 'Checkout'
98   });
99 };
100
```