DataBases Notes

Table of Contents :

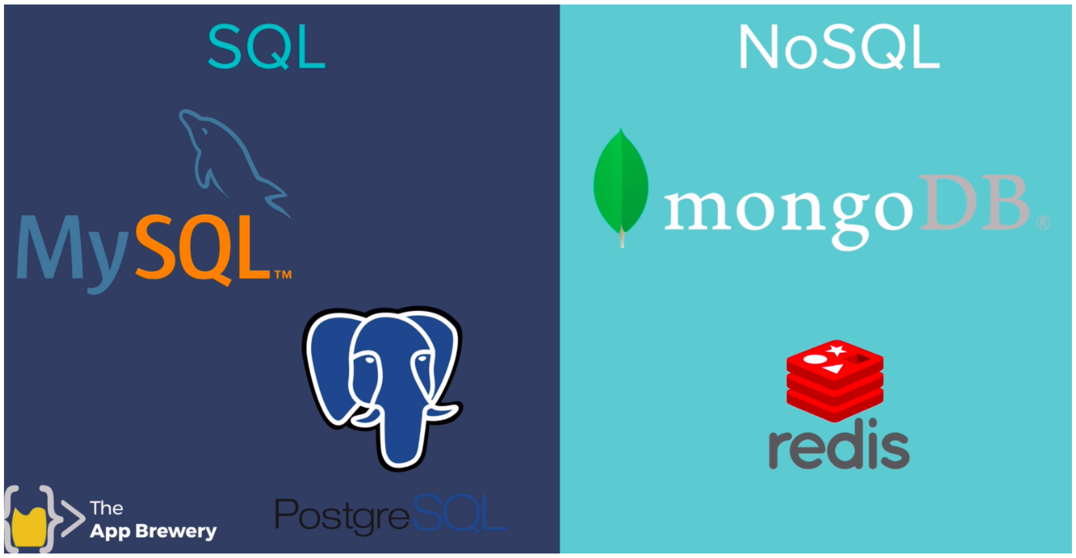
1. **Introduction to Database**
2. **Types of Database**
    a. SQL
    b. NoSQL
3. **SQL**
    a. Introduction
    b. CRUD Operations
    c. Relationships, Inner Join, Foreign Key
4. **MongoDB**
    a. Introduction
    b. CRUD Operations
    c. Relationships
    d. MongoDB with Node.js (Native Driver)
5. **Mongoose**
    a. Introduction
    b. CRUD Operations
6. **Mongoose vs MongoDB Node.js Driver: A Comparison**

DataBase : any collection of data, or information, that is specially organized for rapid search and retrieval by a computer.

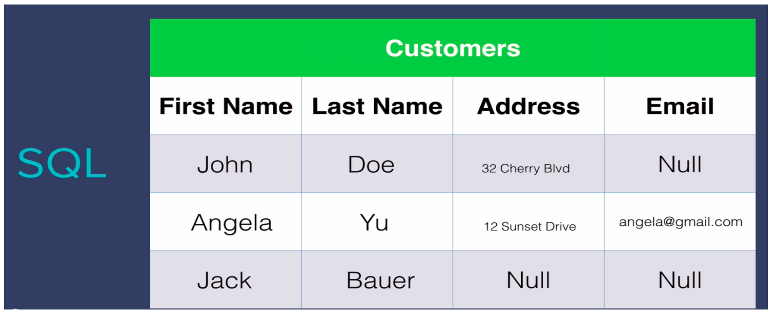
Two types:
SQL (Structured Query Language) and **NOSQL** (Not Only Structured Query Language).

The most common Database types used in the industry are :

- SQL : MySQL, Postgres
- NoSQL : MongoDB, Redis



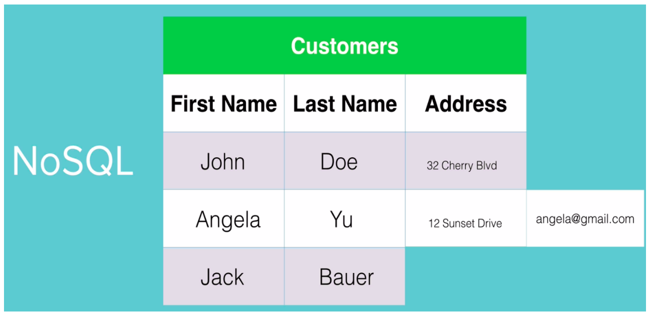
- SQL databases have fixed or static or predefined schema.
- SQL databases display data in the form of tables so it is known as table-based databases.



Customers

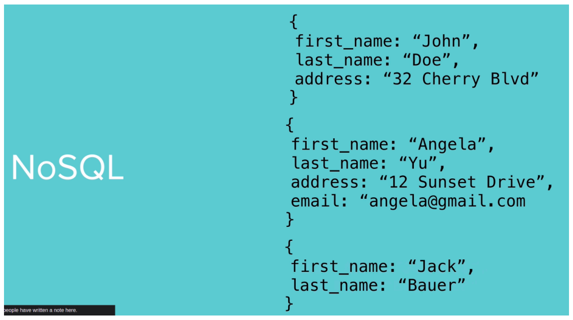
First Name	Last Name	Address	Email
John	Doe	32 Cherry Blvd	Null
Angela	Yu	12 Sunset Drive	angela@gmail.com
Jack	Bauer	Null	Null

- NoSQL databases have dynamic schemas.
- NoSQL databases display data as collections of key-value pairs, documents, graph databases or wide-column stores.
- In NoSQL databases, collections of documents are used to query the data. It is also called unstructured query language.



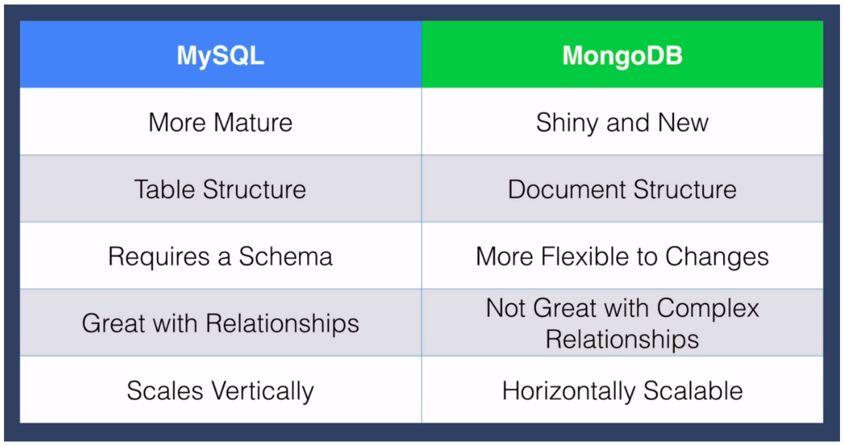
Customers

First Name	Last Name	Address	
John	Doe	32 Cherry Blvd	
Angela	Yu	12 Sunset Drive	angela@gmail.com
Jack	Bauer		



```
{
 first_name: “John”,
 last_name: “Doe”,
 address: “32 Cherry Blvd”
}
{
 first_name: “Angela”,
 last_name: “Yu”,
 address: “12 Sunset Drive”,
 email: “angela@gmail.com
}
{
 first_name: “Jack”,
 last_name: “Bauer”
}
```

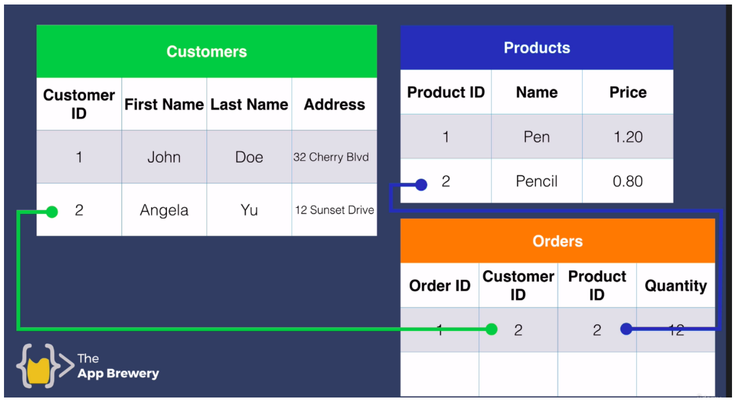
Difference Between SQL and NoSQL Database :



MySQL	MongoDB
More Mature	Shiny and New
Table Structure	Document Structure
Requires a Schema	More Flexible to Changes
Great with Relationships	Not Great with Complex Relationships
Scales Vertically	Horizontally Scalable

Relationship in data (Connecting one or more tables) :

The below screenshot shows the relationship between customer, products and orders table.



Customers

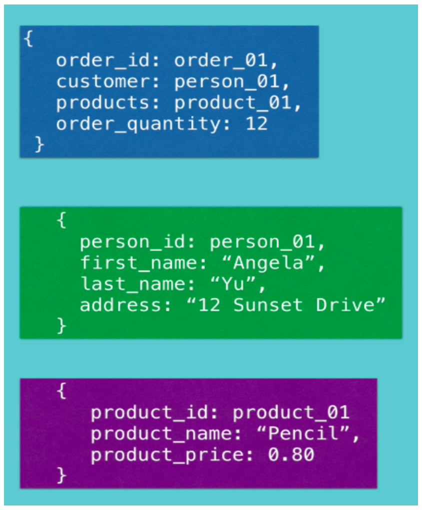
Customer ID	First Name	Last Name	Address
1	John	Doe	32 Cherry Blvd
2	Angela	Yu	12 Sunset Drive

Products

Product ID	Name	Price
1	Pen	1.20
2	Pencil	0.80

Orders

Order ID	Customer ID	Product ID	Quantity
1	2	2	12



```
{
   order_id: order_01,
   customer: person_01,
   products: product_01,
   order_quantity: 12
 }

  {
    person_id: person_01,
    first_name: “Angela”,
    last_name: “Yu”,
    address: “12 Sunset Drive”
  }

  {
     product_id: product_01
     product_name: “Pencil”,
     product_price: 0.80
  }
```

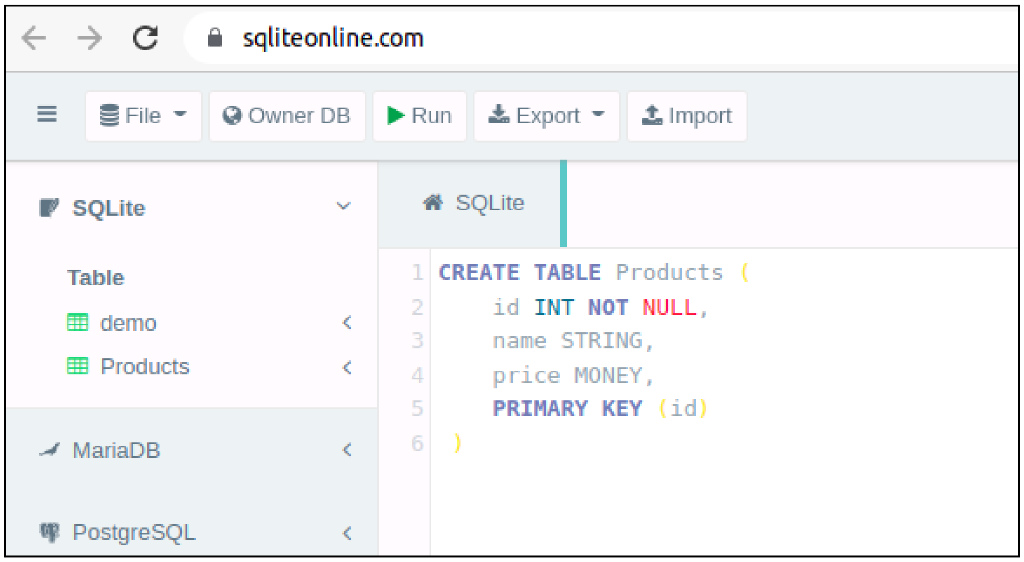
SQL : RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)

For every type of database, the main thing to do is the following (CRUD Operations) :

Create
Read
Update
Destroy

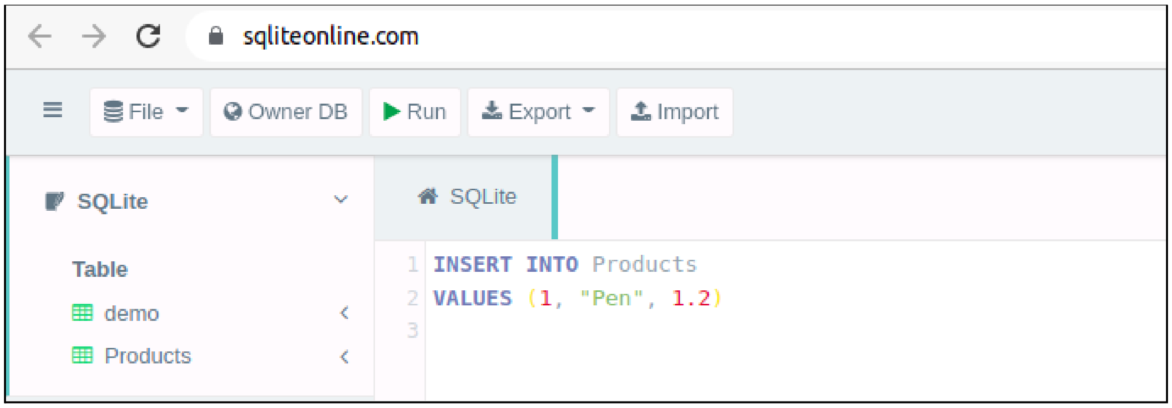
1. Create Operation :

Created table products with the following column and the constraints.



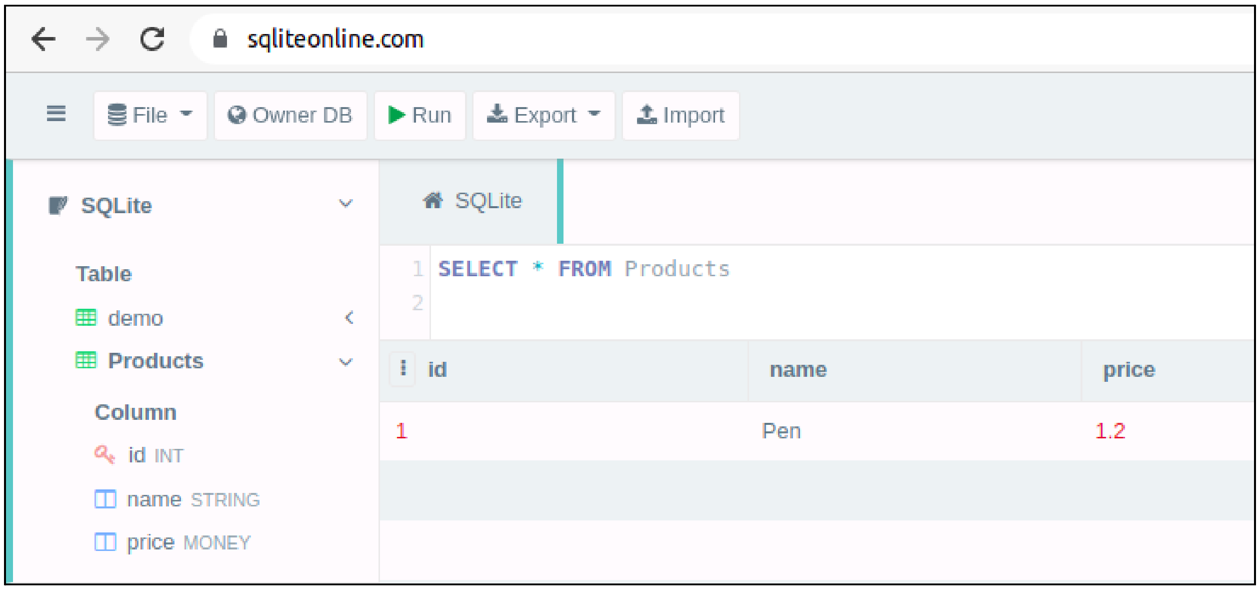
```
CREATE TABLE Products (
    id INT NOT NULL,
    name STRING,
    price MONEY,
    PRIMARY KEY (id)
 )
```

Inserted data into Products table.



```
INSERT INTO Products
VALUES (1, "Pen", 1.2)
```

Check the inserted data using ***SELECT*** * Command

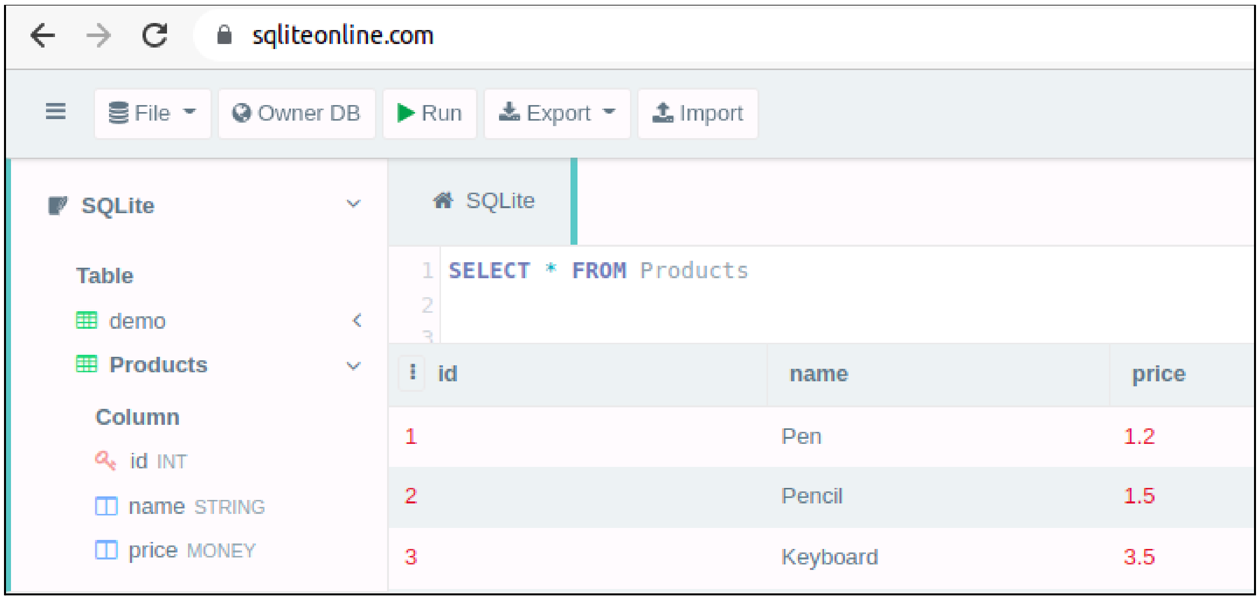


```
SELECT * FROM Products
```

id	name	price
1	Pen	1.2

2. READ Operation :

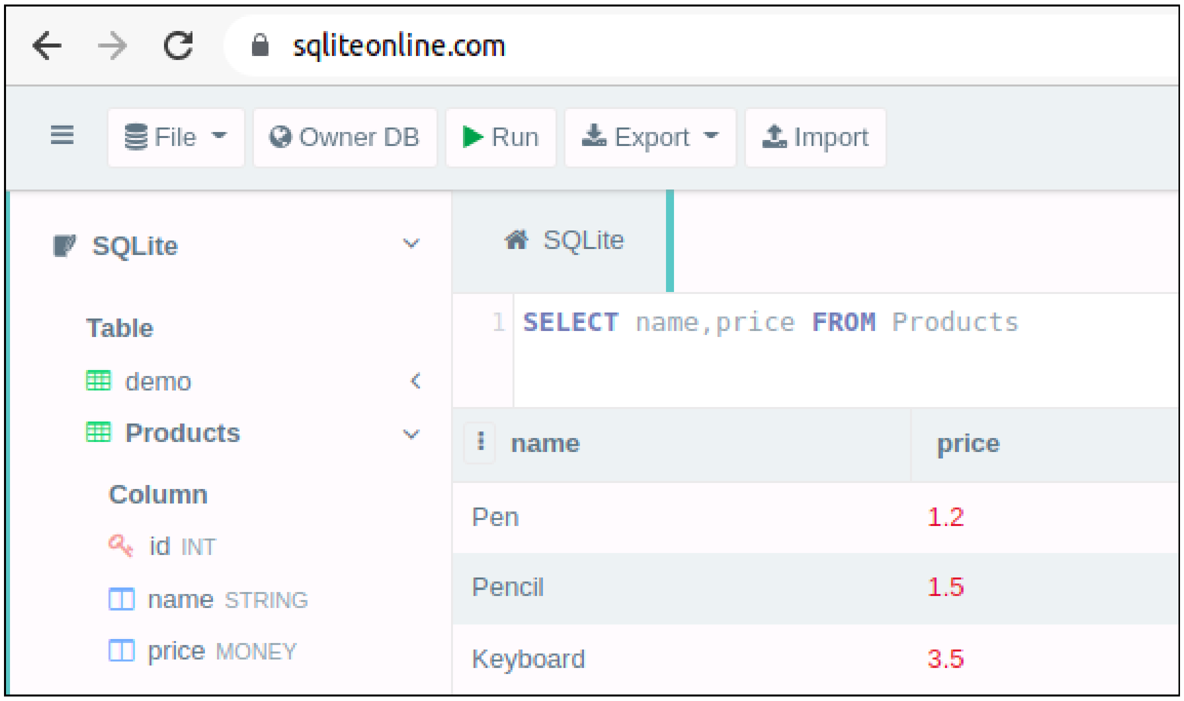
Using SELECT command to read the data:



```
SELECT * FROM Products
```

id	name	price
1	Pen	1.2
2	Pencil	1.5
3	Keyboard	3.5

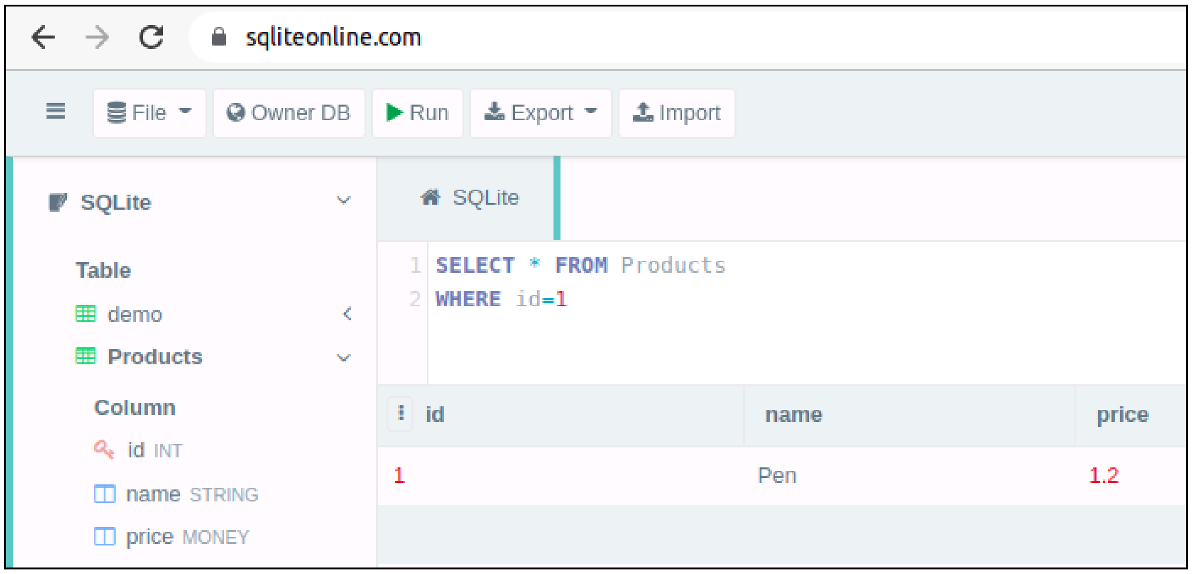
Reading name and price Only : ***SELECT name, price FROM Products***



```sql
SELECT name,price FROM Products
```

name	price
Pen	1.2
Pencil	1.5
Keyboard	3.5

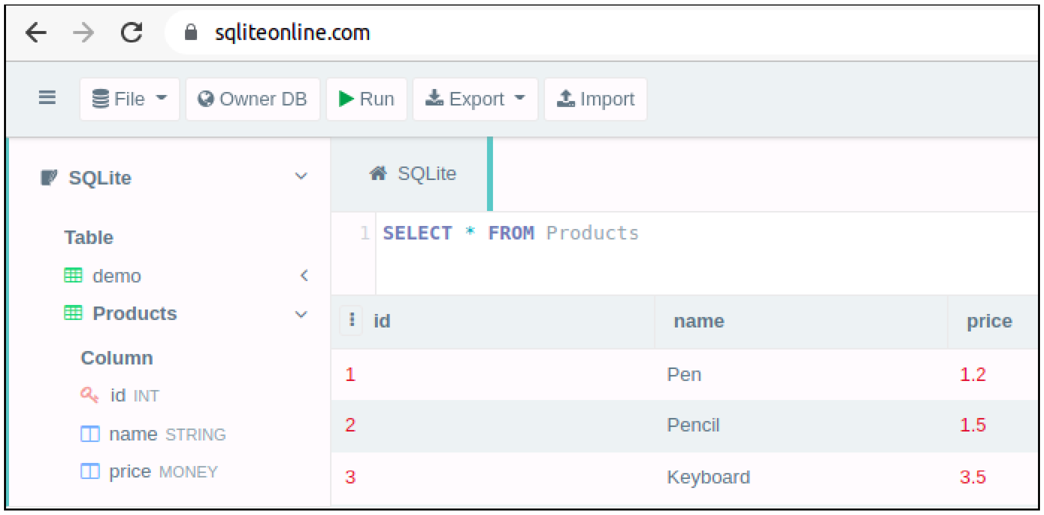
SELECT Using ***WHERE Condition :***



```sql
SELECT * FROM Products
WHERE id=1
```

id	name	price
1	Pen	1.2

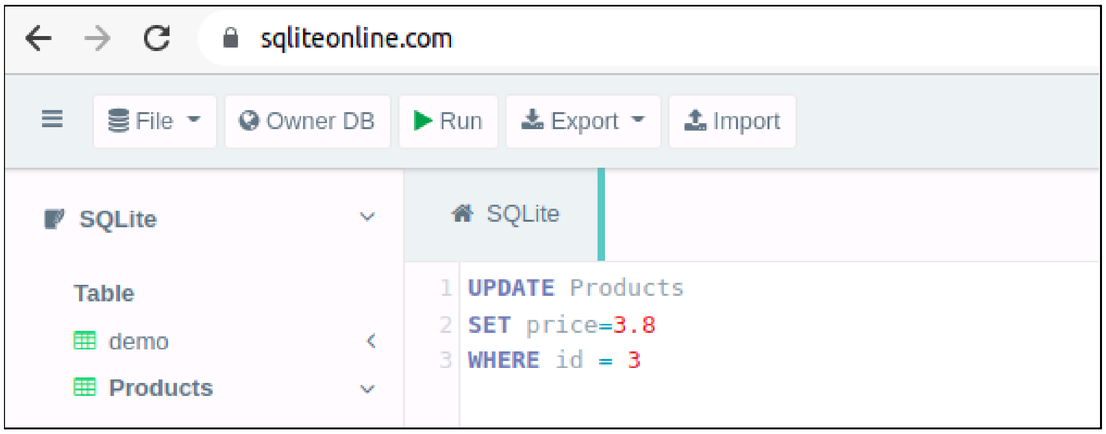
3. UPDATE Operation :



```
SELECT * FROM Products
```

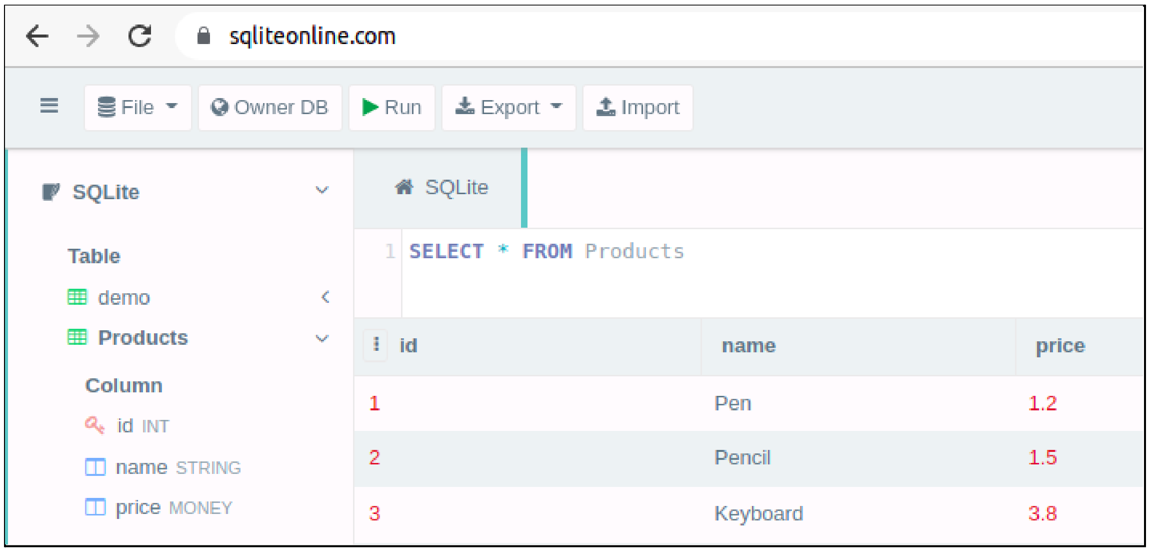
id	name	price
1	Pen	1.2
2	Pencil	1.5
3	Keyboard	3.5

Set the price = 3.8 where id is 3.



```
UPDATE Products
SET price=3.8
WHERE id = 3
```

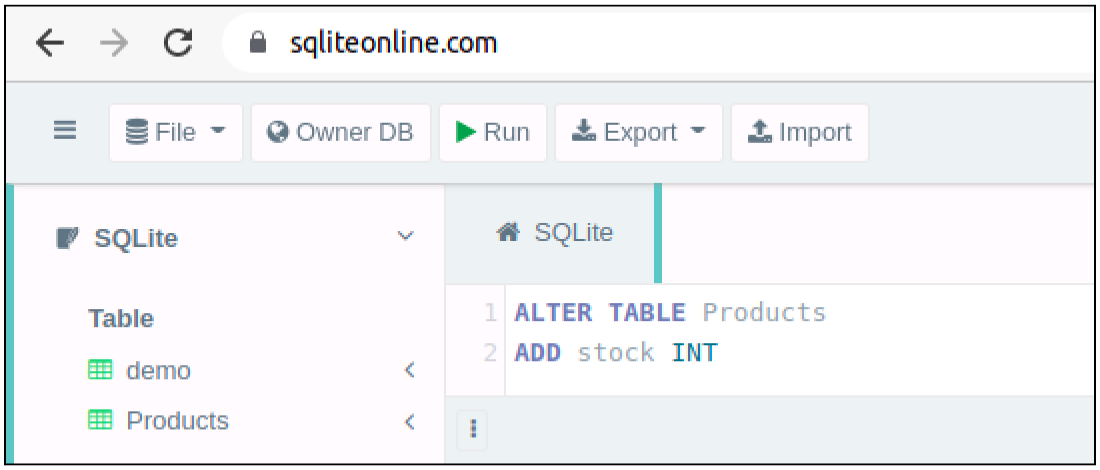
Checking the updated values.



```
SELECT * FROM Products
```

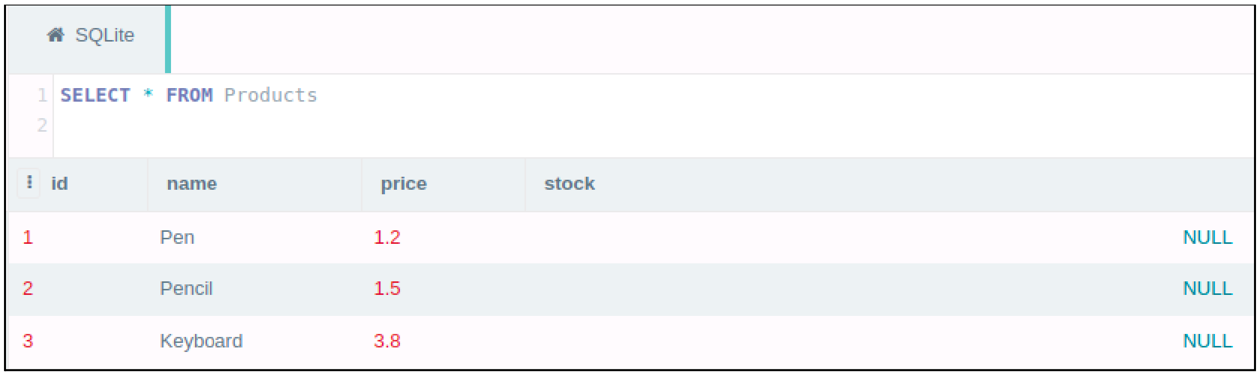
id	name	price
1	Pen	1.2
2	Pencil	1.5
3	Keyboard	3.8

Update can also be performed using the ***ALTER*** command. Added a new column called stock.



```
ALTER TABLE Products
ADD stock INT
```

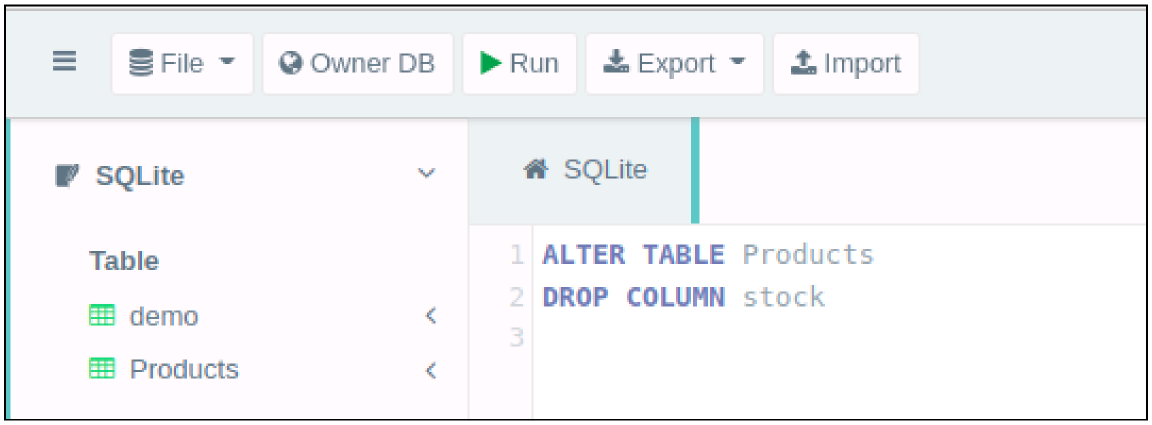
Checked the table values:



```
SELECT * FROM Products
```

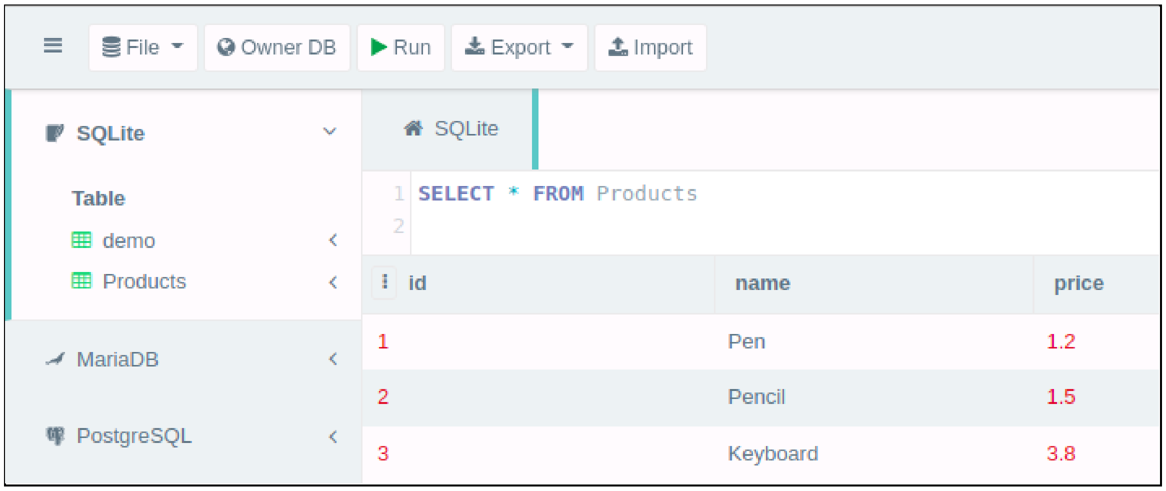
id	name	price	stock
1	Pen	1.2	NULL
2	Pencil	1.5	NULL
3	Keyboard	3.8	NULL

Deleted the newly added column stock:



```
ALTER TABLE Products
DROP COLUMN stock
```

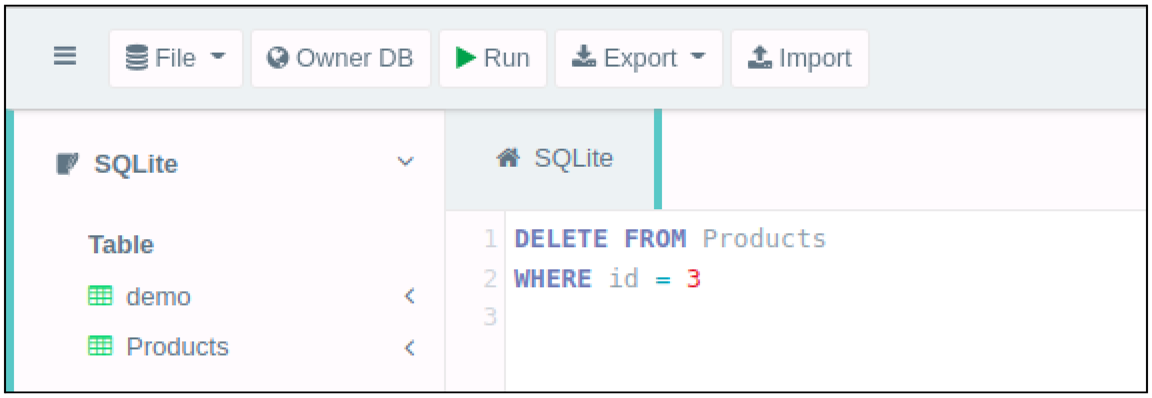
Checked the updated values:



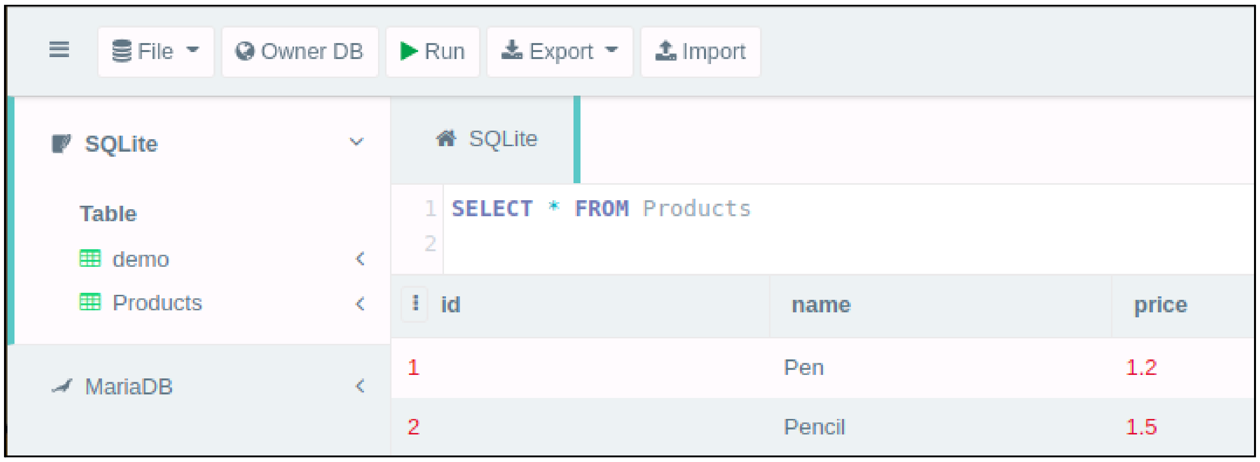
```sql
SELECT * FROM Products
```

id	name	price
1	Pen	1.2
2	Pencil	1.5
3	Keyboard	3.8

4. DELETE Operation :



```sql
DELETE FROM Products
WHERE id = 3
```

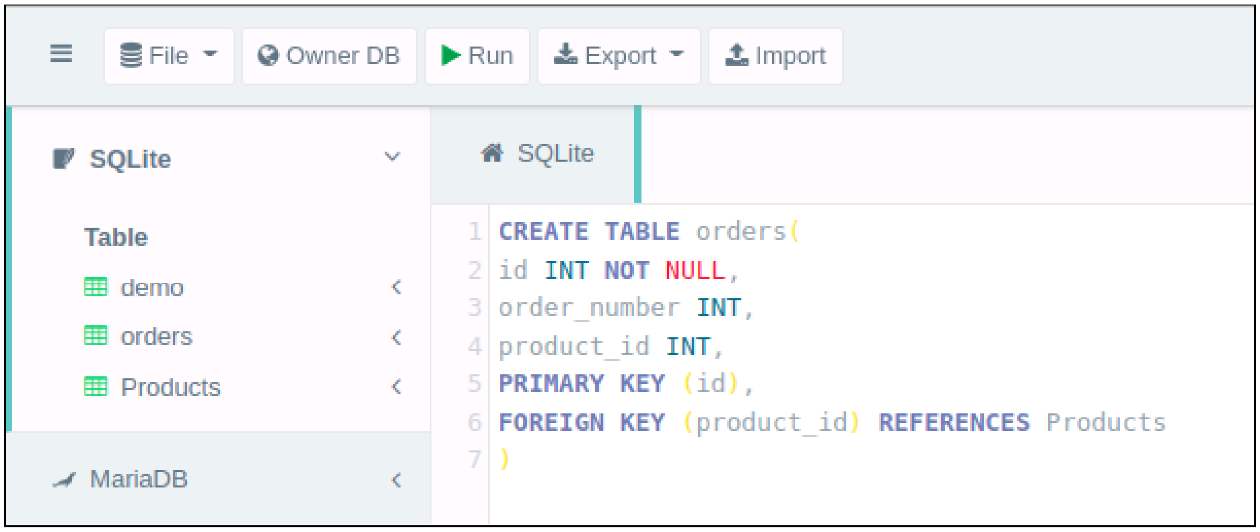


```sql
SELECT * FROM Products
```

id	name	price
1	Pen	1.2
2	Pencil	1.5

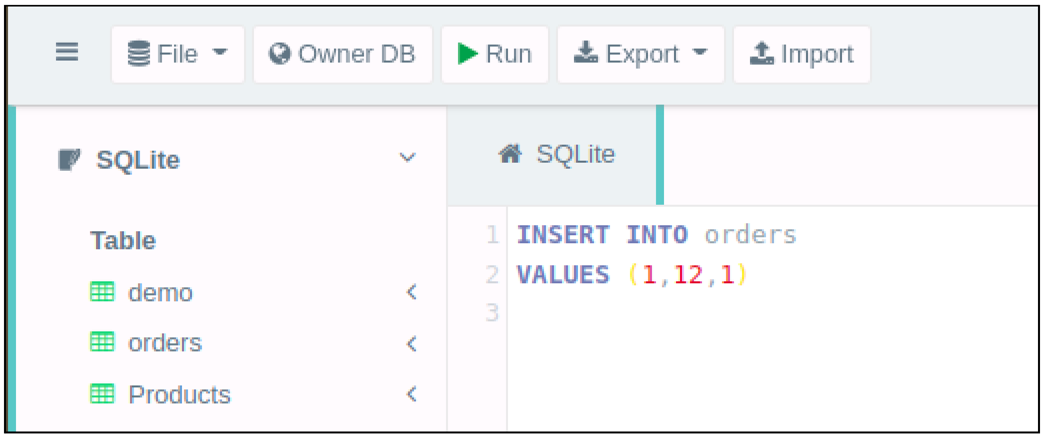
Relationships, Inner Join, Foreign Key :

Created table orders with **Primary key as (id)** and **Foreign key as (product id)** which **references** products table.



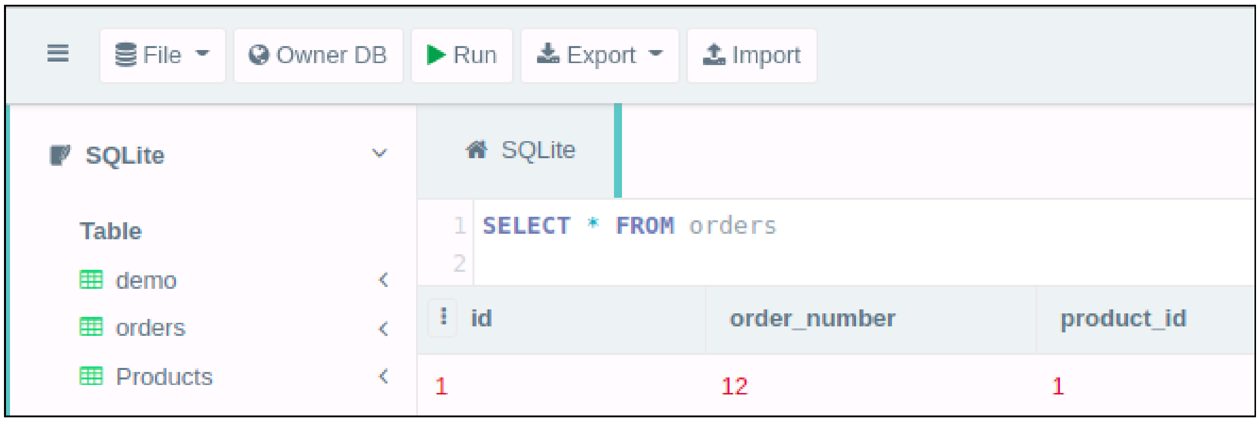
```
CREATE TABLE orders(
id INT NOT NULL,
order_number INT,
product_id INT,
PRIMARY KEY (id),
FOREIGN KEY (product_id) REFERENCES Products
)
```

Inserted values in orders table:



```
INSERT INTO orders
VALUES (1,12,1)
```

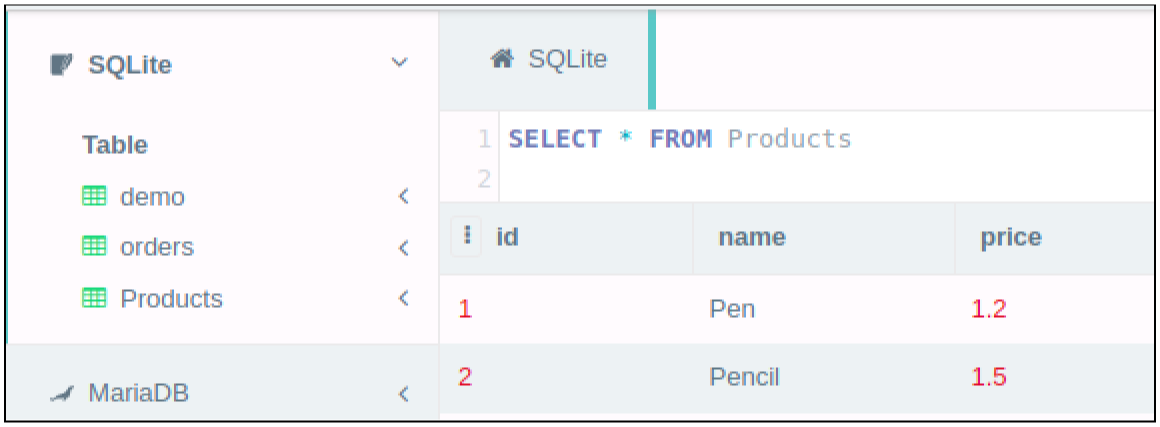
Checking the values of orders table:



```
SELECT * FROM orders
```

id	order_number	product_id
1	12	1

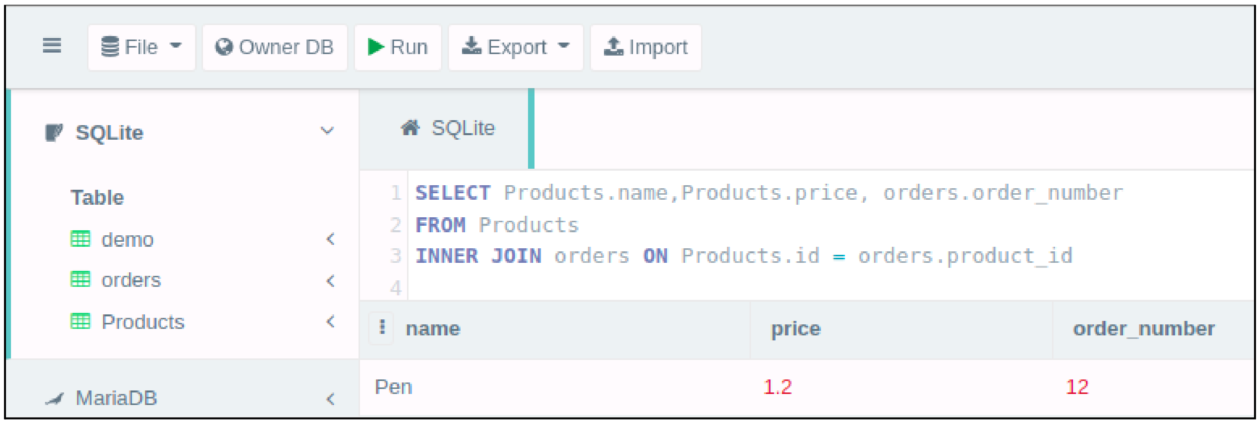
Checking the values of Products table:



```
SELECT * FROM Products
```

id	name	price
1	Pen	1.2
2	Pencil	1.5

Joined both the tables based on id’s :



```
SELECT Products.name,Products.price, orders.order_number
FROM Products
INNER JOIN orders ON Products.id = orders.product_id
```

name	price	order_number
Pen	1.2	12

MongoDB :

- It is an open-source document database and leading NoSQL database.
- It is a cross-platform, document oriented database that provides high performance, high availability, and easy scalability.
- Introduction to MongoDB
- MongoDB CRUD Operations
- MongoDB Tutorial

Some common terminologies :

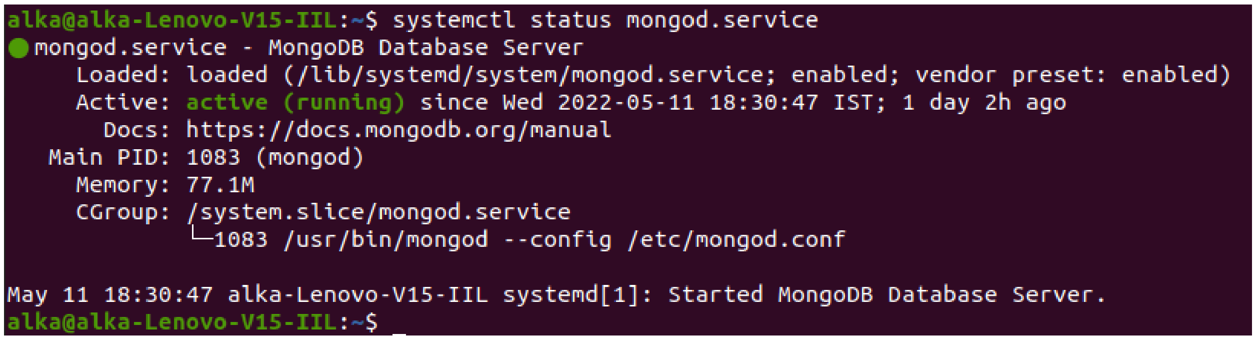
- **Database :** physical container for collections.
- **Collection :** group of MongoDB documents.
- **Document :** set of key-value pairs, have dynamic schema.

The following table shows the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)

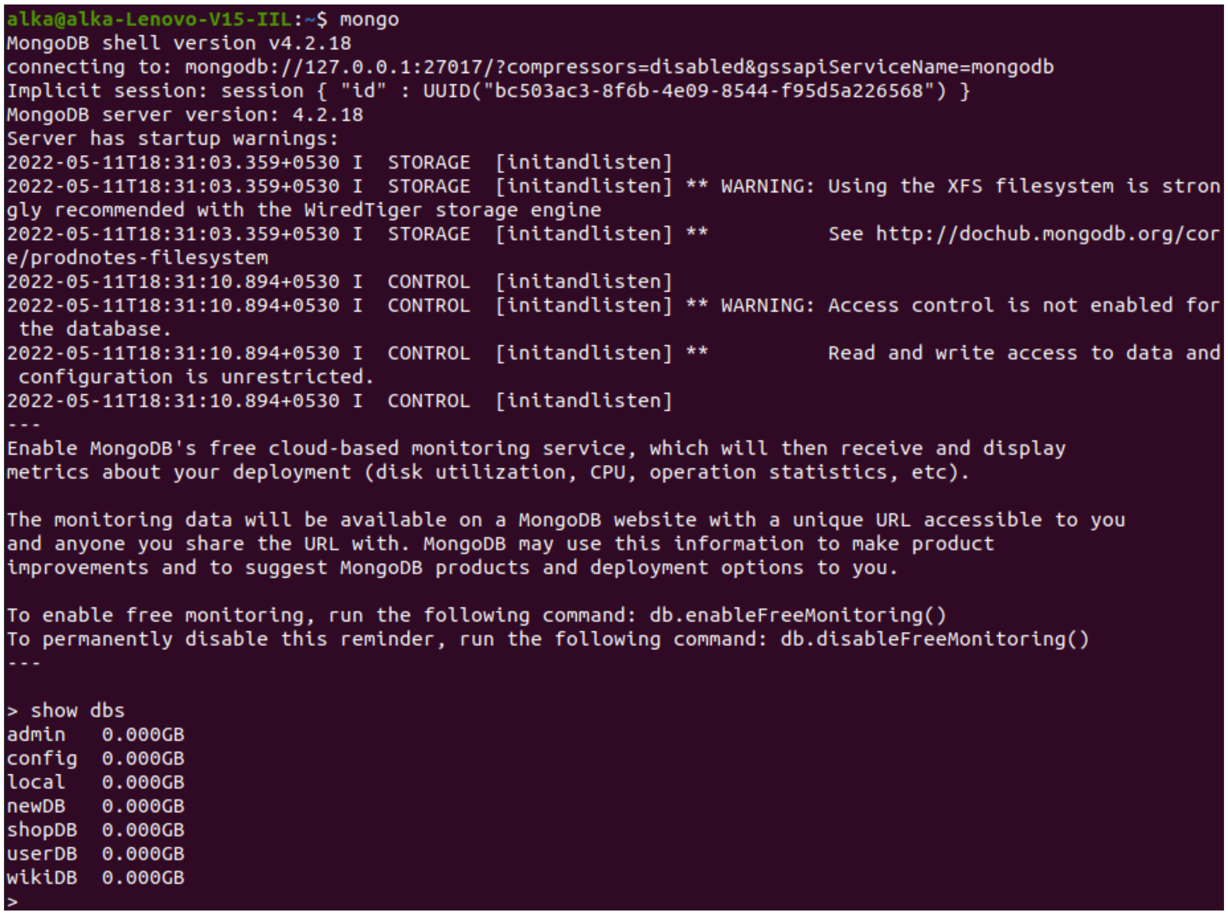
How to use MongoDB :

1. Install mongod.service :



```
alka@alka-Lenovo-V15-IIL:~$ systemctl status mongod.service
● mongod.service - MongoDB Database Server
     Loaded: loaded (/lib/systemd/system/mongod.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2022-05-11 18:30:47 IST; 1 day 2h ago
       Docs: https://docs.mongodb.org/manual
   Main PID: 1083 (mongod)
     Memory: 77.1M
     CGroup: /system.slice/mongod.service
             └─1083 /usr/bin/mongod --config /etc/mongod.conf

May 11 18:30:47 alka-Lenovo-V15-IIL systemd[1]: Started MongoDB Database Server.
alka@alka-Lenovo-V15-IIL:~$ _
```

2. Mongo Shell : Once it is installed we can open mongo shell using mongo command :



```
alka@alka-Lenovo-V15-IIL:~$ mongo
MongoDB shell version v4.2.18
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("bc503ac3-8f6b-4e09-8544-f95d5a226568") }
MongoDB server version: 4.2.18
Server has startup warnings:
2022-05-11T18:31:03.359+0530 I  STORAGE  [initandlisten]
2022-05-11T18:31:03.359+0530 I  STORAGE  [initandlisten] ** WARNING: Using the XFS filesystem is stron
gly recommended with the WiredTiger storage engine
2022-05-11T18:31:03.359+0530 I  STORAGE  [initandlisten] **          See http://dochub.mongodb.org/cor
e/prodnotes-filesystem
2022-05-11T18:31:10.894+0530 I  CONTROL  [initandlisten]
2022-05-11T18:31:10.894+0530 I  CONTROL  [initandlisten] ** WARNING: Access control is not enabled for
 the database.
2022-05-11T18:31:10.894+0530 I  CONTROL  [initandlisten] **          Read and write access to data and
 configuration is unrestricted.
2022-05-11T18:31:10.894+0530 I  CONTROL  [initandlisten]
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---

> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
newDB   0.000GB
shopDB  0.000GB
userDB  0.000GB
wikiDB  0.000GB
>
```

Create Database :

Use command : The command will create a new database if it doesn't exist, otherwise it will return the existing database.

```
>use mydb
switched to db mydb
```

To check currently selected database, use the command ***db***

```
>db
mydb
```

To check databases list, use the command ***show dbs*** :

```
>show dbs
local     0.78125GB
test      0.23012GB
```

Drop Database :

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

CRUD operations :

1. CREATE:

Create collection : The createCollection() Method is used to create a collection.

```
db.createCollection(name, options)
```

```
>use test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
>
```

```
>show collections
mycollection
system.indexes
```

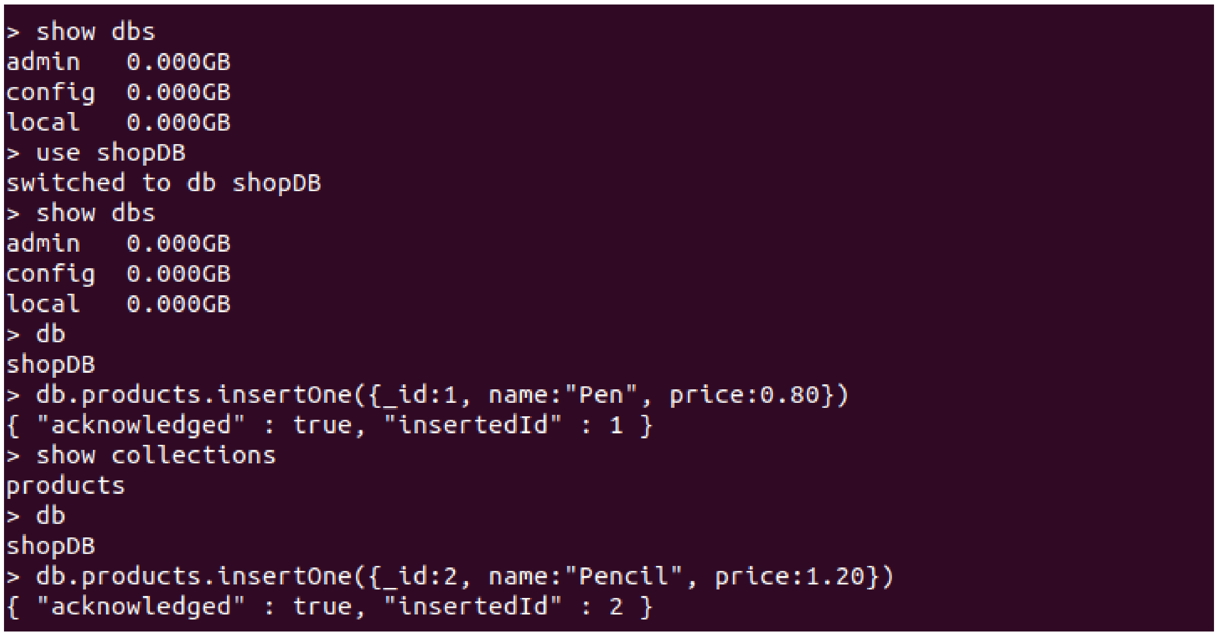
Drop collection : drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

```
db.COLLECTION_NAME.drop()
```

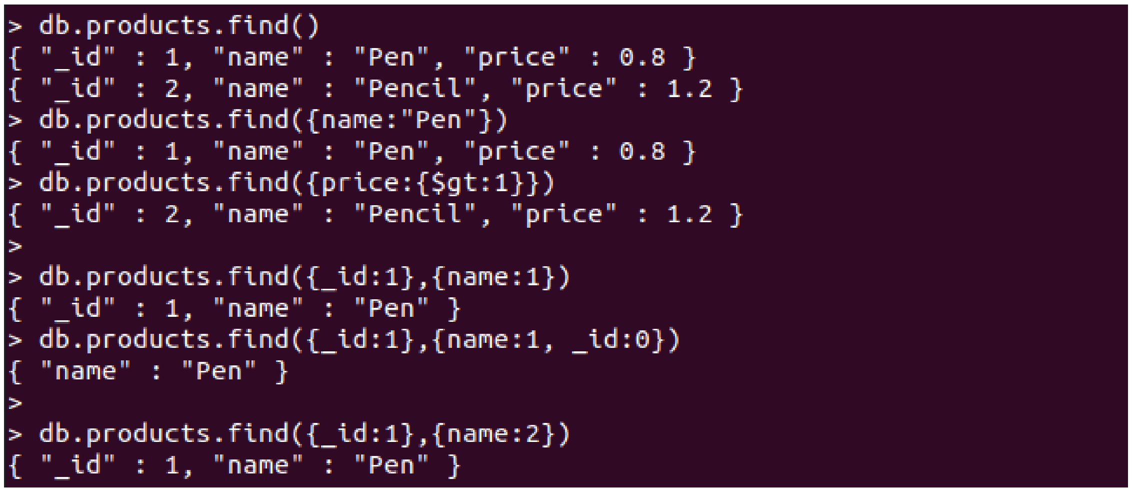
```
>db.mycollection.drop()
true
>
```

Insert a document :

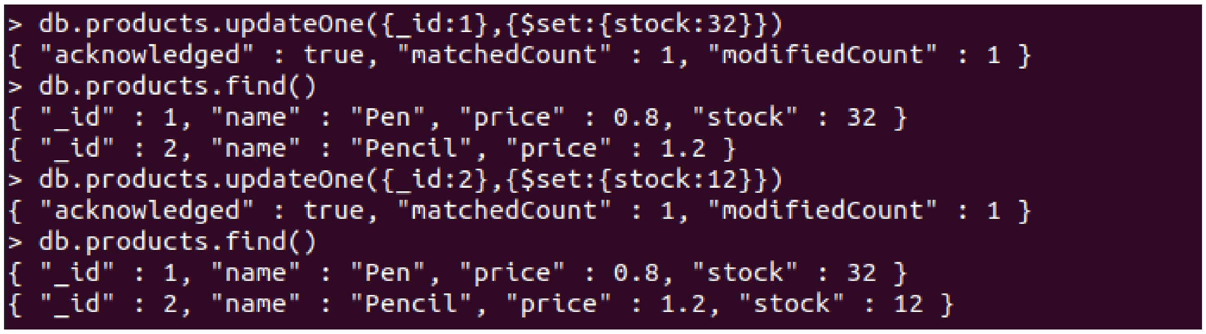
- insertOne() method is used to insert only one document into a collection.
- insertMany() method is used to insert multiple documents. To this method we need to pass an array of documents.


```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> use shopDB
switched to db shopDB
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> db
shopDB
> db.products.insertOne({_id:1, name:"Pen", price:0.80})
{ "acknowledged" : true, "insertedId" : 1 }
> show collections
products
> db
shopDB
> db.products.insertOne({_id:2, name:"Pencil", price:1.20})
{ "acknowledged" : true, "insertedId" : 2 }
```

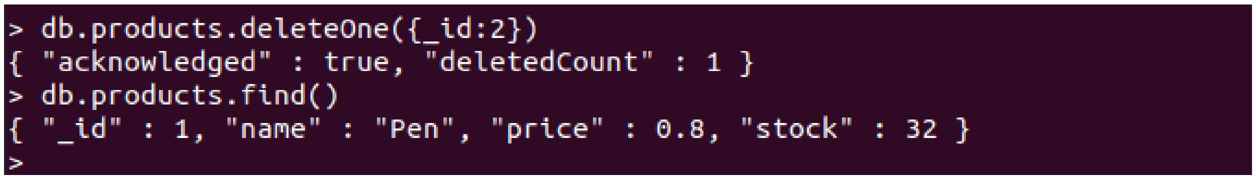
2. READ and QUERY:


```
> db.products.find()
{ "_id" : 1, "name" : "Pen", "price" : 0.8 }
{ "_id" : 2, "name" : "Pencil", "price" : 1.2 }
> db.products.find({name:"Pen"})
{ "_id" : 1, "name" : "Pen", "price" : 0.8 }
> db.products.find({price:{$gt:1}})
{ "_id" : 2, "name" : "Pencil", "price" : 1.2 }
>
> db.products.find({_id:1},{name:1})
{ "_id" : 1, "name" : "Pen" }
> db.products.find({_id:1},{name:1, _id:0})
{ "name" : "Pen" }
>
> db.products.find({_id:1},{name:2})
{ "_id" : 1, "name" : "Pen" }
```

3. UPDATE:


```
> db.products.updateOne({_id:1},{$set:{stock:32}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.products.find()
{ "_id" : 1, "name" : "Pen", "price" : 0.8, "stock" : 32 }
{ "_id" : 2, "name" : "Pencil", "price" : 1.2 }
> db.products.updateOne({_id:2},{$set:{stock:12}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.products.find()
{ "_id" : 1, "name" : "Pen", "price" : 0.8, "stock" : 32 }
{ "_id" : 2, "name" : "Pencil", "price" : 1.2, "stock" : 12 }
```

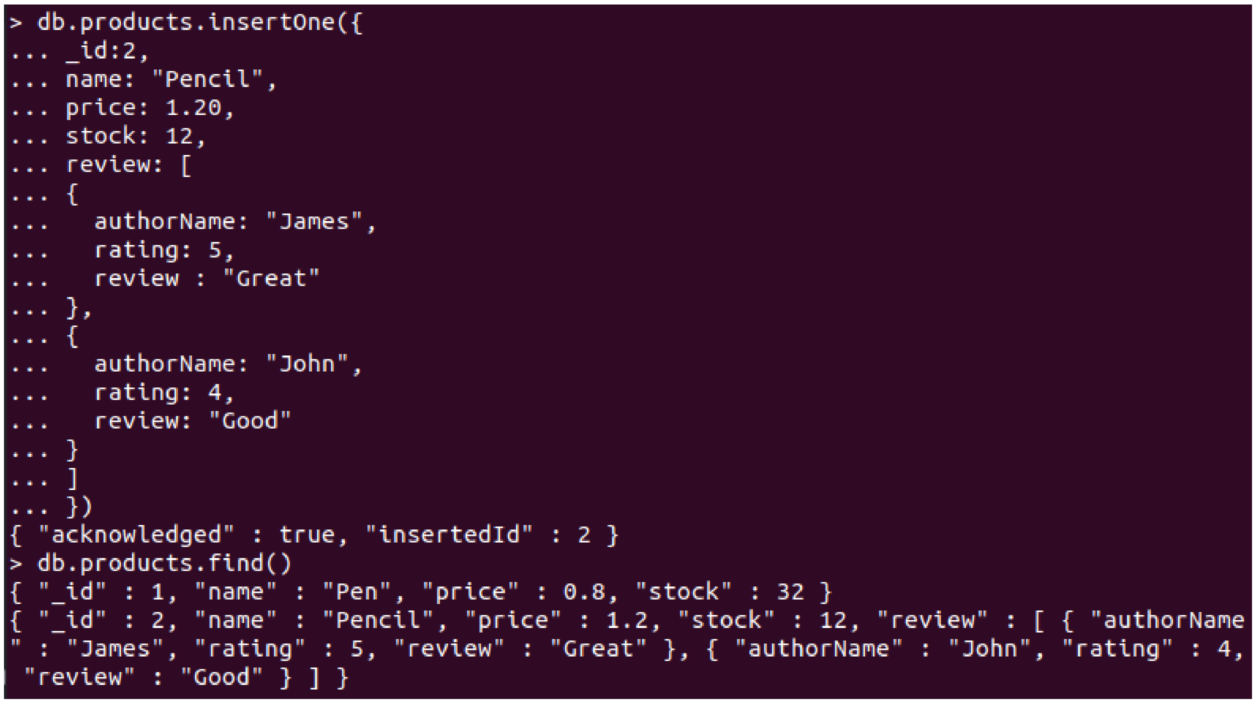
4. DELETE:



```
> db.products.deleteOne({_id:2})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.products.find()
{ "_id" : 1, "name" : "Pen", "price" : 0.8, "stock" : 32 }
>
```

Relationships:

- represent how various documents are logically related to each other.
- can be modeled via **Embedded and Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.



```
> db.products.insertOne({
... _id:2,
... name: "Pencil",
... price: 1.20,
... stock: 12,
... review: [
... {
...   authorName: "James",
...   rating: 5,
...   review : "Great"
... },
... {
...   authorName: "John",
...   rating: 4,
...   review: "Good"
... }
... ]
... })
{ "acknowledged" : true, "insertedId" : 2 }
> db.products.find()
{ "_id" : 1, "name" : "Pen", "price" : 0.8, "stock" : 32 }
{ "_id" : 2, "name" : "Pencil", "price" : 1.2, "stock" : 12, "review" : [ { "authorName
" : "James", "rating" : 5, "review" : "Great" }, { "authorName" : "John", "rating" : 4,
 "review" : "Good" } ] }
```

Consider a User document and Address document :

User document:

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Tom Hanks",
   "contact": "987654321",
   "dob": "01-01-1991"
}
```

Address document :

```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

Modeling Embedded Relationships :
In the embedded approach, we will embed the address document inside the user document.

```
> db.users.insert({
      {
            "_id":ObjectId("52ffc33cd85242f436000001"),
            "contact": "987654321",
            "dob": "01-01-1991",
            "name": "Tom Benzamin",
            "address": [
                  {
                        "building": "22 A, Indiana Apt",
                        "pincode": 123456,
                        "city": "Los Angeles",
                        "state": "California"
                  },
                  {
                        "building": "170 A, Acropolis Apt",
                        "pincode": 456789,
                        "city": "Chicago",
                        "state": "Illinois"
                  }
            ]
      }
})
```

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as −

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Note: that in the above query, db and users are the database and collection respectively.

The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships :

This is the approach of designing a normalized relationship. In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's id field.

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin",
   "address_ids": [
      ObjectId("52ffc4a5d85242602e000000"),
      ObjectId("52ffc4a5d85242602e000001")
   ]
}
```

With this approach, we will need two queries: first to fetch the address_ids fields from the user document and second to fetch these addresses from address collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

MongoDB with Node.js

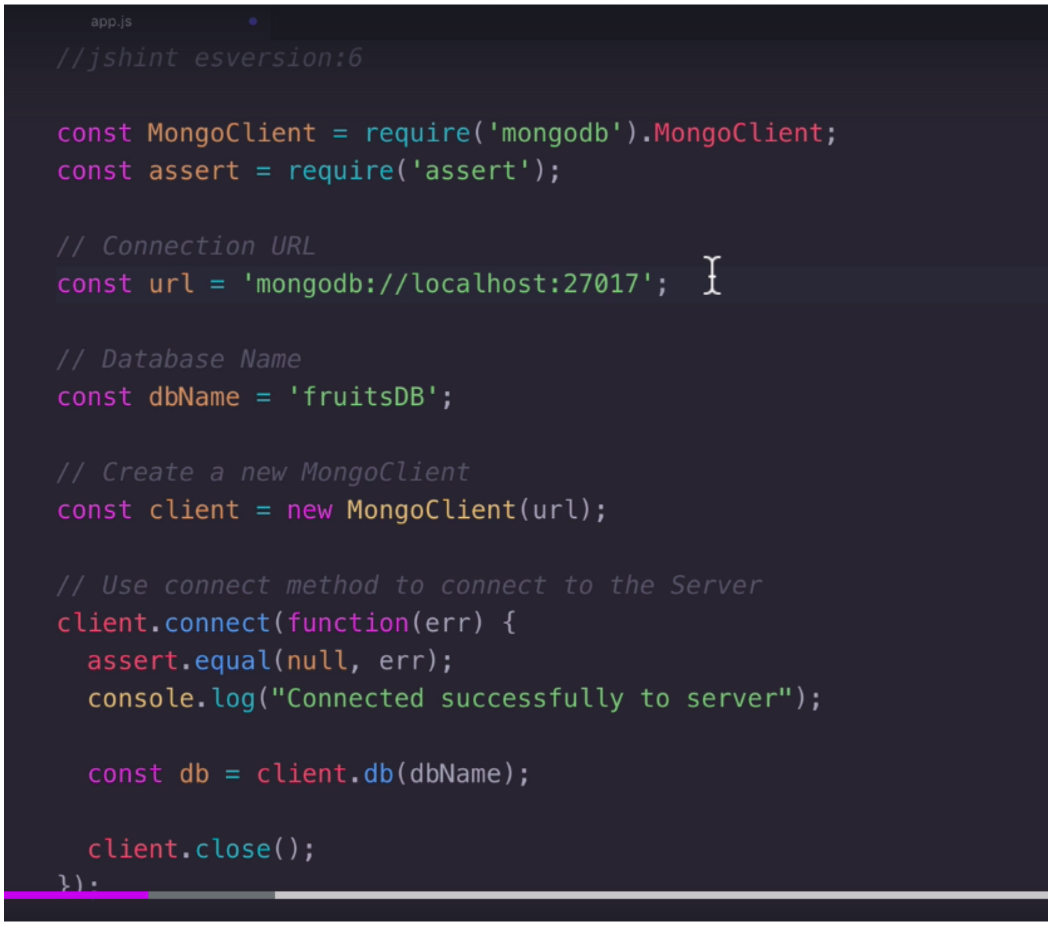
Working with Native MongoDB Driver: It enables the mongodb database to interact with the application.

- Start Developing with MongoDB — MongoDB Drivers

Steps to work with Native Driver :

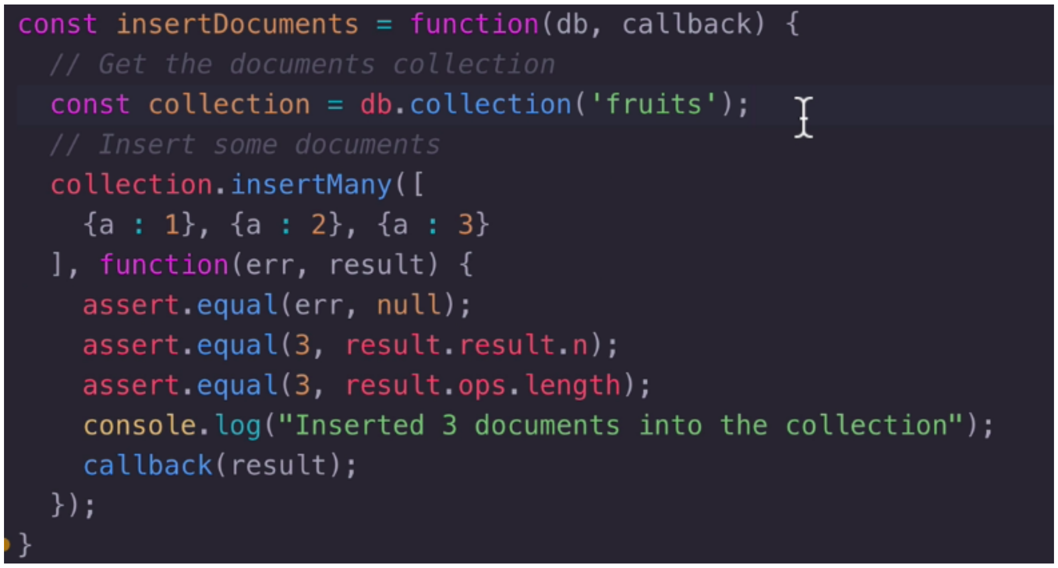
- Install the driver using above link
- Create a project
- Navigate inside the project and create App.js
- Initialize npm : npm init -y
- Install MongoDB native driver : npm i mongodb

Connecting Application to MongoDB :



```
//jshint esversion:6

const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'fruitsDB';

// Create a new MongoClient
const client = new MongoClient(url);

// Use connect method to connect to the Server
client.connect(function(err) {
  assert.equal(null, err);
  console.log("Connected successfully to server");

  const db = client.db(dbName);

  client.close();
});
```

Inserting documents to the collection:



```
const insertDocuments = function(db, callback) {
  // Get the documents collection
  const collection = db.collection('fruits');
  // Insert some documents
  collection.insertMany([
    {a : 1}, {a : 2}, {a : 3}
  ], function(err, result) {
    assert.equal(err, null);
    assert.equal(3, result.result.n);
    assert.equal(3, result.ops.length);
    console.log("Inserted 3 documents into the collection");
    callback(result);
  });
}
```

For other operations please refer this : [Quick Start — Node.js]

Mongoose

Mongoose v6.3.3: Getting Started

- Mongoose is a Node.js-based **Object Data Modeling (ODM)** library for MongoDB.
- It is akin to an Object Relational Mapper (ORM)
- The problem that Mongoose aims to solve is allowing developers to enforce a specific schema at the application layer.
- Mongoose also offers a variety of hooks, model validation, and other features aimed at making it easier to work with MongoDB.

Mongoose Schema and Model :

1. With Mongoose, we can define a Schema object in our application code that maps to a collection in your MongoDB database.
2. The Schema object defines the structure of the documents in your collection.
3. Then, we need to create a Model object out of the schema. The model is used to interact with the collection.

```
const blog = new Schema({
   title: String,
   slug: String,
   published: Boolean,
   content: String,
   tags: [String]
   });

const Blog = mongoose.model('Blog', blog);
```

Executing Operations on MongoDB with Mongoose :

Once we have a Mongoose model defined, we could run queries for fetching, updating, and deleting data against a MongoDB collection that aligns with the Mongoose model. With the above model, we could do things like:

```
// Create a new blog post
const article = new Blog({
   title: 'Awesome Post!',
   slug: 'awesome-post',
   published: true,
   content: 'This is the best post ever',
   tags: ['featured', 'announcement'],
});
```

```
// Insert the article in our MongoDB database
article.save();

// Find a single blog post
Blog.findOne({}, (err, post) => {
   console.log(post);
});
```

Example :

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/newDB');

const schema = new mongoose.Schema({
   name: String,
   age: Number
});

const obj=mongoose.model("Person",schema);

const obj1=new obj({
   name:"Alka",
   age: 24
});

//obj1.save();

obj.find(function(err,result)
{
   if(err)
       console.log(err);
   else
   {
       for(var i=0;i<result.length;i++)
           console.log(result[i].name);
   }
   mongoose.connection.close();
});
```

Mongoose vs MongoDB Node.js Driver: A Comparison

MongoDB & Mongoose: Compatibility and Comparison

Mongoose	MongoDB Node.js Driver
1. The **benefit** of using Mongoose is that we have a schema to work against in our application code and an explicit relationship between our MongoDB documents and the Mongoose models within our application 2. The **downside** is that we can only create specific posts and they have to follow the defined schema. 3. If we change our Mongoose schema, we are changing the relationship completely, and if you're going through rapid development, this can greatly slow you down.	1. On the other hand, if we decided to use just the MongoDB Node.js driver, we could run queries against any collection in our database, or create new ones on the fly. 2. The MongoDB Node.js driver does not have concepts of object data modeling or mapping. 3. We simply write queries against the database and collection we wish to work with to accomplish the business goals.

References :

- SQL Tutorial
- Introduction to MongoDB
- MongoDB CRUD Operations
- MongoDB Tutorial
- Start Developing with MongoDB — MongoDB Drivers
- Quick Start — Node.js
- Mongoose v6.3.3: Getting Started
- https://www.udemy.com/course/the-complete-web-development-bootcamp/learn/lecture/12385850#overview

----
