MongoDB/Node.js

Learning Objectives:

- ▸Node.js MongoDB Create Database
- ▸Node.js MongoDB Create Collection
- ▸Node.js MongoDB Insert
- ▸Node.js MongoDB Find
- ▸Node.js MongoDB Query
- ▸Node.js MongoDB Sort
- ▸Node.js MongoDB Delete
- ▸Node.js MongoDB Drop Collection
- ▸Node.js MongoDB Update
- ▸Node.js MongoDB Limit
- ▸Node.js MongoDB Join
- ▸Node.js MongoDB Relationships using mongoose

MongoDB_Node.js

Node.js can be used in database applications. We will be using MongoDB.

Install MongoDB Driver

Let us try to access a MongoDB database with Node.js.

To download and install the official MongoDB driver, open the Command Terminal and execute the following:

Download and install mongodb package:

```
C:\Users\Your Name>npm install mongodb
```

Now you have downloaded and installed a mongodb database driver.

Node.js can use this module to manipulate MongoDB databases:

```
var mongo = require('mongodb');
```

MongoDB_Node.js

Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

Create a database called "mydb":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

Save the code above in a file called "demo_create_mongo_db.js" and run the file:

Run "demo_create_mongo_db.js":

```
C:\Users\Your Name>node demo_create_mongo_db.js
```

MongoDB_Node.js

Which will give you this result:

```
Database created
```

Note: MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).

Creating a Collection

To create a collection in MongoDB, use the createCollection() method:

Example:
Create a collection called "customers":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_mongodb_createcollection.js" and run the file:

```
C:\Users\Your Name>node demo_mongodb_createcollection.js
```

Node.js MongoDB Insert

To insert a record, or document as it is called in MongoDB, into a collection, we use the insertOne() method.

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

Insert a document in the "customers" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_mongodb_insert.js" and run the file:

```
C:\Users\Your Name>node demo_mongodb_insert.js
```

Node.js MongoDB Find

In MongoDB we use the find and findOne methods to find data in a collection.

Just like the SELECT statement is used to find data in a table in a MySQL database.

Find One

To select data from a collection in MongoDB, we can use the findOne() method.

The findOne() method returns the first occurrence in the selection.

The first parameter of the findOne() method is a query object. In this example we use an empty query object, which selects all documents in a collection (but returns only the first document).

Example:

Find the first document in the customers collection:

MongoDB_Node.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").findOne({}, function(err, result) {
    if (err) throw err;
    console.log(result.name);
    db.close();
  });
});
```

Save the code above in a file called "demo_mongodb_findone.js" and run the file:

```
C:\Users\Your Name>node demo_mongodb_findone.js
```

Find All

- To select data from a table in MongoDB, we can also use the find() method.
- The find() method returns all occurrences in the selection.
- The first parameter of the find() method is a query object. In this example we use an empty query object, which selects all documents in the collection.
- Note: No parameters in the find() method gives you the same result as SELECT * in MySQL.

MongoDB_Node.js

Example:
Find all documents in the customers collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find({}).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Save the code above in a file called "demo_mongodb_find.js" and run the file:

```
C:\Users\Your Name>node demo_mongodb_find.js
```

MongoDB_Node.js

Node.js MongoDB Query

Filter the Result

When finding documents in a collection, you can filter the result by using a query object.

The first argument of the find() method is a query object, and is used to limit the search.

Example

Find documents with the address "Park Lane 38":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var query = { address: "Park Lane 38" };
  dbo.collection("customers").find(query).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_mongodb_query.js" and run the file:

```
C:\Users\Your Name>node demo_mongodb_query.js
```

Filter With Regular Expressions

- You can write regular expressions to find exactly what you are searching for.
- Regular expressions can only be used to query strings.
- To find only the documents where the "address" field starts with the letter "S", use the regular expression

Example:
Find documents where the address starts with the letter "S":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var query = { address: /^S/ };
  dbo.collection("customers").find(query).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_mongodb_query_s.js" and run the file:

```
C:\Users\Your Name>node demo_mongodb_query_s.js
```

Node.js MongoDB Sort

Sort the Result

- Use the sort() method to sort the result in ascending or descending order.
- The sort() method takes one parameter, an object defining the sorting order.

Example:
Sort the result alphabetically by name:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var mysort = { name: 1 };
  dbo.collection("customers").find().sort(mysort).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_sort.js" and run the file:

```
C:\Users\Your Name>node demo_sort.js
```

<u>Node.js MongoDB Delete</u>

Delete Document

- To delete a record, or document as it is called in MongoDB, we use the deleteOne() method.
- The first parameter of the deleteOne() method is a query object defining which document to delete.
- Note: If the query finds more than one document, only the first occurrence is deleted.

Example

Delete the document with the address "Mountain 21":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: 'Mountain 21' };
  dbo.collection("customers").deleteOne(myquery, function(err, obj) {
    if (err) throw err;
    console.log("1 document deleted");
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_delete.js" and run the file:

```
C:\Users\Your Name>node demo_delete.js
```

Delete Many

To delete more than one document, use the deleteMany() method.

The first parameter of the deleteMany() method is a query object defining which documents to delete.

Example

Delete all documents were the address starts with the letter "O":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: /^O/ };
  dbo.collection("customers").deleteMany(myquery, function(err, obj) {
    if (err) throw err;
    console.log(obj.result.n + " document(s) deleted");
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_delete_many.js" and run the file:

```
C:\Users\Your Name>node demo_delete_many.js
```

Node.js MongoDB Drop

Drop Collection

- You can delete a table, or collection as it is called in MongoDB, by using the drop() method.
- The drop() method takes a callback function containing the error object and the result parameter which returns true if the collection was dropped successfully, otherwise it returns false.

Example:

Delete the "customers" table:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").drop(function(err, delOK) {
    if (err) throw err;
    if (delOK) console.log("Collection deleted");
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_drop.js" and run the file:

```
C:\Users\Your Name>node demo_drop.js
```

Node.js MongoDB Update

Update Document

- You can update a document by using the updateOne() method.
- The first parameter of the updateOne() method is a query object defining which document to update.
- **Note**: If the query finds more than one record, only the first occurrence is updated.
- The second parameter is an object defining the new values of the document.

Example

Update the document with the address "Valley 345" to name="Mickey" and address="Canyon 123":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: "Valley 345" };
  var newvalues = { $set: {name: "Mickey", address: "Canyon 123" } };
  dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
    if (err) throw err;
    console.log("1 document updated");
    db.close();
  });
```

MongoDB_Node.js

Save the code above in a file called "demo_drop.js" and run the file:

```
C:\Users\Your Name>node demo_drop.js
```

Node.js MongoDB Update

Update Document

- You can update a document by using the updateOne() method.
- The first parameter of the updateOne() method is a query object defining which document to update.
- **Note**: If the query finds more than one record, only the first occurrence is updated.
- The second parameter is an object defining the new values of the document.

Example

Update the document with the address "Valley 345" to name="Mickey" and address="Canyon 123":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: "Valley 345" };
  var newvalues = { $set: {name: "Mickey", address: "Canyon 123" } };
  dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
    if (err) throw err;
    console.log("1 document updated");
    db.close();
  });
});
```

MongoDB_Node.js

Save the code above in a file called "demo_update_one.js" and run the file:

```
C:\Users\Your Name>node demo_update_one.js
```

Update Only Specific Fields

When using the **$set** operator, only the specified fields are updated:

Example

Update the address from "Valley 345" to "Canyon 123":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: "Valley 345" };
  var newvalues = {$set: {address: "Canyon 123"} };
  dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
    if (err) throw err;
    console.log("1 document updated");
    db.close();
  });
});
```

MongoDB_Node.js

Update Many Documents

To update all documents that meets the criteria of the query, use the updateMany() method.

Example

Update all documents where the name starts with the letter "S":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: /^S/ };
  var newvalues = {$set: {name: "Minnie"} };
  dbo.collection("customers").updateMany(myquery, newvalues, function(err, res) {
    if (err) throw err;
    console.log(res.result.nModified + " document(s) updated");
    db.close();
  });
});
```

Save the code above in a file called "demo_update_many.js" and run the file:

```
C:\Users\Your Name>node demo_update_many.js
```

MongoDB_Node.js

Node.js MongoDB Limit

Limit the Result

- To limit the result in MongoDB, we use the **limit**() method.
- The limit() method takes one parameter, a number defining how many documents to return.
- Consider you have a "customers" collection:

Example

Limit the result to only return 5 documents:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find().limit(5).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Save the code above in a file called "demo_mongodb_limit.js" and run the file:

```
C:\Users\Your Name>node demo_mongodb_limit.js
```

MongoDB_Node.js

Node.js MongoDB Join

Join Collections

- MongoDB is not a relational database, but you can perform a left outer join by using the $lookup stage.
- The $lookup stage lets you specify which collection you want to join with the current collection, and which fields that should match.
- Consider you have a "orders" collection and a "products" collection:

orders

```
[
  { _id: 1, product_id: 154, status: 1 }
]
```

products

```
[
  { _id: 154, name: 'Chocolate Heaven' },
  { _id: 155, name: 'Tasty Lemons' },
  { _id: 156, name: 'Vanilla Dreams' }
]
```

MongoDB_Node.js

Example

Join the matching "products" document(s) to the "orders" collection:

```
ar MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection('orders').aggregate([
    { $lookup:
       {
         from: 'products',
         localField: 'product_id',
         foreignField: '_id',
         as: 'orderdetails'
       }
     }
    ]).toArray(function(err, res) {
    if (err) throw err;
    console.log(JSON.stringify(res));
    db.close();
  });
});
```

Save the code above in a file called "demo_mongodb_join.js" and run the file:

MongoDB_Node.js

```
C:\Users\Your Name>node demo_mongodb_join.js
```

Which will give you this result:

```

  { "_id": 1, "product_id": 154, "status": 1, "orderdetails": [
    { "_id": 154, "name": "Chocolate Heaven" } ]
  }
]
```

MongoDB Relationships using mongoose
Mongoose is a third-party package that builds upon the official MongoDB driver, but it makes accessing MongoDB much easier and more convenient. Mongoose use schemas that mongodb does not really use, so we can define how our data should look like and that allows us to conveniently store and fetch data.
This might not be the solution we are looking for if we have the requirement of unstructured data. But often, that's not the case, and therefore, Mongoose is a great tool that could even work with such unstructured data.
Installation:
To install the mongoose package by using the following npm command:

npm install --save Mongoose

Schema
A description of the shape a unit of data should undertake. So for a house isn't the data, but a description of what the data of a house should look like.

MongoDB_Node.js

```
const mongoose = require("mongoose")

const houseSchema = new mongoose.Schema({
    street: String,
    city: String,
    state: String,
    zip: String
})
```

If we want to manage a collection of documents (a bunch of items) of this datatype we then declare a model. This creates a collection and becomes the conduit to add, update, delete and retrieve data from the collection.

```
const House = mongoose.model("House", houseSchema)

// query for all houses
House.find({})
```

MongoDB_Node.js

One to One Relationships

One to One relationships are the simplest. Imagine that every house can only have one owner, and every owner can only own one house. This is a one to one relationship. everything is unique on both sides there really isn't a need for more than one collection. Instead we can nest one type of data in the other.

```
const mongoose = require("mongoose")

const Owner = new mongoose.Schema({
    name: String
})

const houseSchema = new mongoose.Schema({
    street: String,
    city: String,
    state: String,
    zip: String
    owner: Owner
})

const House = mongoose.model("House", houseSchema)
// Create a new house
House.create({
    street: "100 Maple Street",
    city: "Fort Townville,
    state: "New West Virgota",
    zip: "77777"
    owner: {name: "Alex Merced"}
})

// query for all houses, will include the nested owner info
House.find({})
```

MongoDB_Node.js

One to Many

Let's see how we can refactor this to handle a Owner having many Houses, but Houses only having one owner. This is One to Many. So Owners are the "one" side of the relationship, and House is the "many" side. Typically what we do is track the one side from the many side (it's the house data that'll track the owner).
With mongoose we have a special datatype that tells mongoose that the entries in that field are all objects _ids of documents in some other collection. See this at work below.
The populate function when we query the data will make sure mongoose fetches the data from the related table and inserts where needed.
Note: You do also have the option of nesting an array of House in the Owner schema, although there is a maximum size for one document that can cause scaling issues later if you try to nest too much data.

```
const mongoose = require("mongoose")

const ownerSchema = new mongoose.Schema({
    name: String
})

const Owner = mongoose.model("Owner", ownerSchema)

const houseSchema = new mongoose.Schema({
    street: String,
    city: String,
    state: String,
    zip: String
    owner: {type: mongoose.Types.ObjectId, ref: "Owner"}
})

const House = mongoose.model("House", houseSchema)

// Create an Owner
const alex = await Owner.create({name: "Alex Merced"})

// Create a new house
House.create({
    street: "100 Maple Street",
    city: "Fort Townville,
    state: "New West Virgota",
    zip: "77777"
    owner: alex
})

// query for all houses, use populate to include owner info
House.find({}).populate("owner")
```

MongoDB_Node.js

Many to Many

In all reality, houses can have many owners and owners can have many owners, so we truly have a many to many relationship. In this situation we create a third collection to track the different matches.

```
const mongoose = require("mongoose")

const ownerSchema = new mongoose.Schema({
    name: String
})

const Owner = mongoose.model("Owner", ownerSchema)

const houseSchema = new mongoose.Schema({
    street: String,
    city: String,
    state: String,
    zip: String
})

const House = mongoose.model("House", houseSchema)

const houseOwnerSchema = {
    owner: {type: mongoose.Types.ObjectId, ref: "Owner"},
    house: {type: mongoose.Types.ObjectId, ref: "House"}
}

const HouseOwner = mongoose.model("HouseOwner", houseOwnerSchema)
```

```
// Create a Owner
const alex = await Owner.create({name: "Alex Merced"})

// Create a new house
const mapleStreet = await House.create({
    street: "100 Maple Street",
    city: "Fort Townville,
    state: "New West Virgota",
    zip: "77777"
    owner: alex
})

// Create record that the owner owns the house
HouseOwner.create({owner: alex, house: mapleStreet})

// QUery for all houses owned by alex
HouseOwner.find({owner: alex}).populate("house")

//Query for all owners of the Maple Street House
HoseOwner.find({house: mapleStreet}).populate("owner")
```

Questions ?