# MONGOOSE & NODEJS VS MONGODB

HAJAR BENDHIBA

## SUMMARY

## INTRODUCTION

Using MongoDB with Node.js involves leveraging the MongoDB Node.js client library to interact with a MongoDB database. Here's a comprehensive guide to get you started:

- Node.js installed
- MongoDB installed and running or use a cloud service like MongoDB Atlas.
- MongoDB URI: For a local database, it’s typically mongodb://127.0.0.1:27017, and for Atlas mongodb+srv://<username>:<password>@cluster.mongodb.net.
- Install the MongoDB Node.js driver via npm:

```
npm install mongodb
```

# CONNECTION TO MONGODB

```
const { MongoClient } = require('mongodb');
const client = new MongoClient("mongodb://localhost:27017");

(async () => {
  try {
    await client.connect();
    const collection = client.db("myDatabase").collection("myCollection");
    console.log(await collection.find({}).toArray());
  } finally {
    await client.close();
  }
})
();
```

# BASIC OPERATIONS

## CRUD Operations

- Insert Document:

```
await collection.insertOne({ name: 'Ali', age: 25 });
```

- Query Documents:

```
const result = await collection.find({ age: { Sgte: 18 } }).toArray();
```

- Update Documents:

```
await collection.updateOne({ name: 'Ali' }, { Sset: { age: 26 } });
```

- Delete Documents:

```
await collection.deleteOne({ name: 'Ali' });
```

## Aggregation Pipeline

- Smatch: Filter documents.
- Sgroup: Group documents.
- Ssort: Sort documents.
- Sproject: Reshape output.

```
const pipeline = [
  { Smatch: { age: { Sgte: 18 } } },
  { Sgroup: { _id: "Sprofession", totalAge: { Ssum: "Sage" } } },
  { Ssort: { totalAge: -1 } },
  { Sproject: { profession: "S_id", totalAge: 1, _id: 0 } }
];
```

# BASIC OPERATIONS

## Transactions & Error Handling

MongoDB supports transactions, so you can wrap multiple operations in a transaction to ensure atomicity.

```
async function runTransaction() {
const session = client.startSession();
try { session.startTransaction();
```

## Finalizing

Fetch results from aggregation and close connection:

```
const result = await collection.aggregate(pipeline).toArray();
console.log(result);
await client.close();
```

# WHAT IS MONGOOSE?

Mongoose is an ODM library for MongoDB and Node.js, enabling easy schema definition, data validation, and CRUD operations. It provides models for interacting with MongoDB, simplifying database management in Node.js applications.

## Mongoose Installation

- Initialize project:
  npm init -y

- Install Mongoose:
  npm install mongoose

- Import Mongoose in your project:
  `const mongoose=require('mongoose');`

## Database Connection:

Create ./src/database.js to connect to MongoDB:

```
const mongoose = require('mongoose');
const server = '127.0.0.1:27017';
const database = 'myDB';
class Database {
 constructor() {
   this._connect();
 }
 _connect() {
   mongoose.connect( `mongodb://S{server}/S{database}`)
    .then(() => console.log('Database connection successful'))
    .catch(err => console.error('Database connection error'));
 }
}
module.exports = new Database();
```

# MONGOOSE SCHEMA VS MODEL

- Schema: Defines the structure of documents, types, and validation.
- Model: Provides an interface for interacting with MongoDB (create, query, update, delete).

Creating a Mongoose Model

- Import Mongoose

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

- Define the Schema

```
let emailSchema = new mongoose.Schema({email: String});
```

- Export the Model

```
module.exports = mongoose.model('Email', emailSchema);
```

- Instance of Model

```
let EmailModel = require('./email');
let msg= newEmailModel({ email: 'ada.lovelace@gmail.com'});
```

# MONGOOSE SCHEMA VS MODEL

## Enhanced Schema Example

Make the email property required, unique, lowercase, and validate with the validator library:

```
let validator = require('validator');

let emailSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true,
    validate: (value) => validator.isEmail(value)
  }
});
module.exports = mongoose.model('Email', emailSchema);
```

## Schema Types in Mongoose

Array
Boolean
Buffer
Date
Mixed
Number
ObjectId
String

# BASIC MONGOOSE OPERATIONS

## Create Record

Create and save a new document:

```
let EmailModel = require('./email');
letmsg=newEmailModel({email:'ADA.LOVELACE@GMAIL.COM'
});
msg.save()
  .then(doc => console.log(doc))
  .catch(err => console.error(err));
```

## Update Record

Update a document's email and add a new field:

```
EmailModel.findOneAndUpdate(
  { email: 'ada.lovelace@gmail.com' },
  { email: 'theoutlander@live.com' },
  { new: true, runValidators: true }
)
.then(doc => console.log(doc))
.catch(err => console.error(err));
```

## Fetch Record

Retrieve the saved record by querying the email:

```
EmailModel.find({ email: 'ada.lovelace@gmail.com' })
 .then(doc => console.log(doc))
 .catch(err => console.error(err));
```

## Delete Record

Remove a document using findOneAndRemove:

```
EmailModel.findOneAndRemove({email:theoutlande@live.com
})
  .then(response => console.log(response))
  .catch(err => console.error(err));
```

# MONGOOSE HELPERS

## Virtual Property

- A virtual property allows you to define fields that are not stored in the database but can be used to get or set values.

```
userSchema.virtual('fullName')
.get(function() {
  return this.firstName + '   ' + this.lastName;})
.set(function(name) {
  let str = name.split('   ');
  this.firstName = str[0];
  this.lastName = str[1];
});
```

## Static Methods

- Custom methods that operate on the entire model (static methods).

```
userSchema.statics.getUsers = function() {
return new Promise((resolve, reject) => {
 this.find((err, docs) => {
   if (err) return reject(err);
   resolve(docs);
 });
});};
```

## Instance Methods

Custom methods that operate on individual documents (instances).

```
userSchema.methods.getInitials = function() {
 return this.firstName[0] + this.lastName[0];};
```

# MONGOOSE HELPERS

## Middleware

Middleware functions run at specific points in the operation pipeline (e.g., before or after save).

```
userSchema.pre('save', function(next) {
  let now = Date.now();
  this.updatedAt = now;
  if (!this.createdAt) this.createdAt = now;
  next();
});
```

## Plugins

Reusable pieces of functionality that can be applied to multiple schemas.

```
module.exports = function timestamp(schema) {
  schema.add({ createdAt: Date, updatedAt: Date });
  schema.pre('save', function(next) {
    let now = Date.now();
    this.updatedAt = now;
    if (!this.createdAt) this.createdAt = now;
    next();});
};
```

These helpers allow for a more flexible and efficient way to interact with data, simplifying tasks such as field manipulation, custom methods, and adding reusable functionality across schemas.

I APPRECIATE YOUR ATTENTION.

HAJAR BENDHIBA