

# 25. MongoDB and Mongoose

## Table of Content

1. MongoDB
2. Mongoose
3. Save data into MongoDB database using APIs

## 1. MongoDB

MongoDB is a rich open-source document-oriented and one of the widely recognised NoSQL database. It is written in C++ programming language.

### ▼ Important Terms

- **Database**

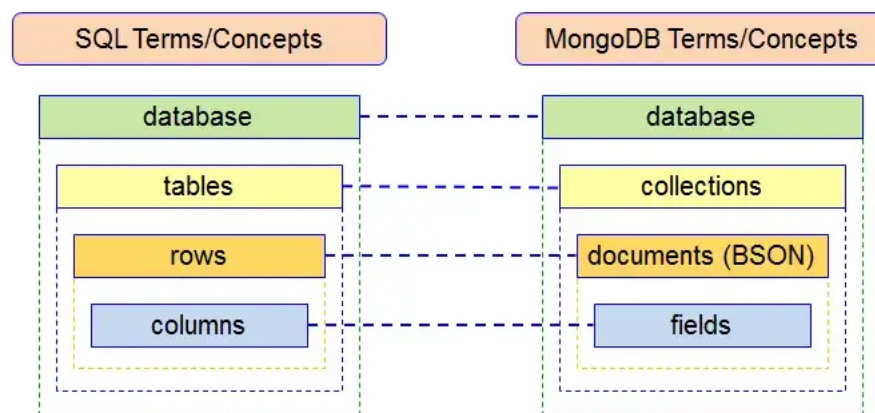
Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

- **Collection**

Collection is a group of documents and is similar to an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields.

- **Document**

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.



BSON stands for **B**inary **J**avascript **O**bject **N**otation. It is a binary-encoded serialization of JSON documents. BSON has been extended to add some optional non-JSON-native data types, like dates and binary data.

### ▼ Data Types

MongoDB supports many datatypes such as:


- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.

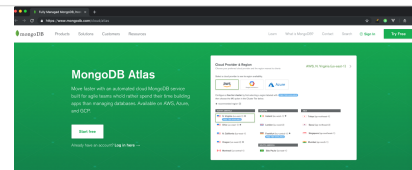
#### ▼ Create your MongoDB Database server

We will be using Atlas to build our MongoDB server. Follow these steps to setup a free server.

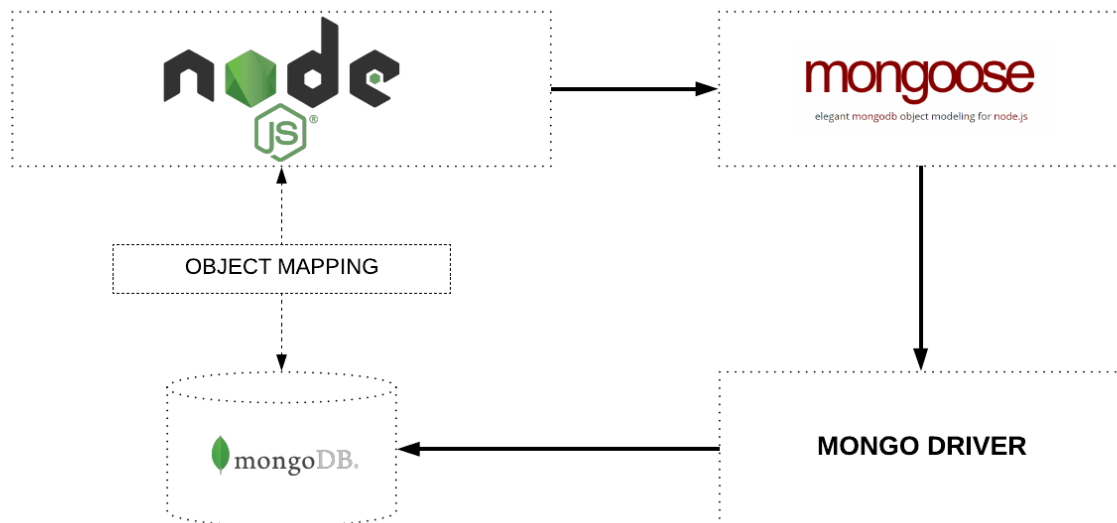
##### Get started with MongoDB Atlas

This article will guide you on how to get started with MongoDB Atlas. 2. Click on Start Free and Complete the form with your email, name, password, accept terms of services and hit the button that says Get started free 3. Then you will be taken to a new page.

 <https://medium.com/nerd-for-tech/get-started-with-mongodb-atlas-dbb734726a7f>



## 2. Mongoose



**Mongoose** is an object modeling tool for MongoDB and Node.js. What this means in practical terms is that you can define your data model in just one place, in your code. It allows defining schemas for our data to fit into, while also abstracting the access to MongoDB. This way we can ensure all saved documents share a structure and contain required properties.

#### ▼ Important Terms

- **Fields**

Fields' or attributes are similar to columns in a SQL table.

- **Schema**

While Mongo is schema-less, SQL defines a schema via the table definition. A Mongoose 'schema' is a document data structure (or shape of the document) that is enforced via the application layer.

- **Models**

Models' are higher-order constructors that take a schema and create an instance of a document equivalent to records in a relational database.

#### ▼ Getting Started with Mongoose

```
npm i mongoose
```

Connecting NodeJS with MongoDB Database using `mongoose`

```
const mongoose = require("mongoose"); // import mongoose

mongoose
  .connect(
    "MONGODB_CONNECTION_STRING", // pass in your connection string here
    {
      useNewUrlParser: true, // default recommended options
      useUnifiedTopology: true,
    }
  )
  .then(e => console.log("MongoDB ready"))
  .catch(console.error); // If you see an error you probably didn't put in the correct connection string/password
```

Make sure to replace **<password>** with the password you created and change **myFirstDatabase** to the name of the database you created in step one.

#### ▼ Creating Models

Each **collection** must have a corresponding model associated with it. For example, if we are storing users in our database, we must create a user model so that Mongoose knows what fields each user has.

Create a new folder called **models**. In this f, make a file called **User.js**. This model will contain all the fields needed to create a user.

Creating a new model is easy with mongoose.js:

```
const { model, Schema } = require("mongoose");

const User = new Schema({ // The Schema constructor is used to define new models
  firstName: String, // You can use type constructors to define field types
  middleName: {
    type: String,
    default: "", // Middle name has a default value of ""
  },
  lastName: String,
  age: Number,
  email: {
    type: String,
    required: true, // All fields are optional unless you specify otherwise
  }
});
```

```

    },
  });

module.exports = model("user", User); // Export the Schema as a mongoose model

```

## ▼ Creating Documents

```

const User = require("../models/User"); // import the User model

const newUser = new User({
  // We can create new documents by using the model as a constructor
  firstName: "Jimmy",
  middleName: "Joseph",
  lastName: "John",
  age: 42,
  email: "jimmy.john@email.com",
});

newUser.save().then(doc => {
  console.log("User added to the database");
});

```

## ▼ Querying the Database

### • .find()

`.find()` is a method that you can call from an exported model, like **User**. The snippet below demonstrates one way you can use the `.find()` method:

```

const User = require("../models/User"); // Import the User model

const findAllUsers = async () => { // Creates an async function since .find() returns a Promise
  const allUsers = await User.find(); // Make sure to use await
  console.log(allUsers);
};

findAllUsers();

```

### ◦ Find with parameters

```

const findUserByFirstName = async firstName => { // take in firstName as a parameter
  // pass in firstName into .find()
  const users = await User.find({ firstName }); // Note this syntax is the same as { firstName: firstName }
  console.log(users);
};

findUserByFirstName("Luke"); // Find all users who's firstName is "Luke"

```



**Note:** `find()` always returns an array of documents. Use `findOne()` when you only need to get one document from your database.

### ◦ find with comparison

```

const findUsersOlderThan = async number => {
  const usersOlderThan50 = await User.find({ age: { $gt: number } }); // You can replace $gt with $gte to find users who are at least
  console.log({ usersOlderThan50 });
};

findUsersOlderThan(50); // finds all users older than 50 years

```

Comparison operators always start with \$

\$eq	
------	--

	Matches values that are equal to a specified value.
<code>\$gt</code>	Matches values that are greater than a specified value.
<code>\$gte</code>	Matches values that are greater than or equal to a specified value.
<code>\$in</code>	Matches any of the values specified in an array.
<code>\$lt</code>	Matches values that are less than a specified value.
<code>\$lte</code>	Matches values that are less than or equal to a specified value.
<code>\$ne</code>	Matches all values that are not equal to a specified value.
<code>\$nin</code>	Matches none of the values specified in an array.

### ▼ Updating Documents

```
const User = require("../models/User"); // Import the User model

const incrementAge = async firstName => {
  const user = await User.findOne({ firstName }); // Find the user by firstName
  if (!user) {
    throw new Error("User not found"); // If no user is found, throw an error
  }
  user.age++; // Mongoose allows you to directly modify the document
  const result = await user.save(); // Saves the document to the database
  console.log(result);
};

incrementAge("Jimmy"); // Increment Jimmy's age
```

- You first need to fetch the document you want to change. This example uses the **firstName** field to get the document.
- In real-world applications, you may want to query by **id** or another unique field.
- After fetching the correct document, you can directly mutate the object and reassign it to any value (as long as it's the correct data type).
- Don't forget to call the **.save()** method to save the updated document in the database.

### ▼ Deleting Documents

The following code sample demonstrates two ways you can delete a document from MongoDB:

```
const User = require("../models/User"); // Import the User model

const deleteUserByFirstName = async firstName => {
  await User.deleteOne({ firstName }); // .deleteOne() won't do anything if the user isn't in the database
  const allUsers = await User.find();
  console.log(allUsers); // console.log all the users after the deletion
};

const deleteUserByFirstName2 = async firstName => {
  const user = await User.findOne({ firstName }); // find the given user
  if (!user) {
    throw new Error("User not found"); // If no user is found, throw an error
  }
  await user.delete();
  const allUsers = await User.find();
  console.log(allUsers);
};

deleteUserByFirstName("Luke"); // Deletes Luke from the database
```

- This snippet shows two ways to delete a document in MongoDB. The first function is much shorter and doesn't require error handling, while the second one does. Depending on your application, one function may suit you better than the other.
- **.deleteOne()** takes in a parameter just like **.findOne()** or **.find()**. You can still use comparison operators and other find operators.

- It is important to note that `.deleteOne()` is a method that can only be called on a model like **User** (notice the capitalization). On the other hand, `.delete()` can only be called on documents like the **user** variable.

## Assignments

1. Build a Database with the following schema and Build a Product Route - `/product`

Product Schema:

productId	ObjectId
title	String
price	Number

1. `POST /add` (pass data in body)
2. `GET /`
3. `DELETE /:productId`
4. `PUT /:productId` (pass data in body)

2. Build a Database with the following schema and Build a Review Route - `/review`

Review Schema:

reviewId	ObjectId
userId	ObjectId
productId	ObjectId
reviewText	String

1. `POST /add` (pass data in body)
2. `GET /getAll/:productId`
3. `DELETE /:reviewId`
4. `PUT /:productId` (pass data in body)