1. **Introduction to MongoDB**
2. **SQL vs NoSQL**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **SQL (e.g. MySQL)** | **NoSQL (e.g. MongoDB)** |
| 1. | RDBMS is a **relational database management system** and works on relational database. | It is non- relational, **document-oriented database management system** & works on document-based database. |
| 2. | It stores data in form **entity** as **table** | MongoDB stores data in form of **documents** |
| 3. | It uses SQL to query database | MongoDB use **BSON** to query database |
| 4. | It stores data in form **table** which have **columns and row.** | It stores data in form **collections** which have **documents and fields.** |

1. **Issues of JSON for MongoDB**

There are several issues that make JSON less than ideal for usage inside of a database.

1. JSON is a **text-based format, and text parsing is very slow**
2. JSON only **supports a limited number of basic data types**

In order to make MongoDB JSON-first, but still high-performance and general-purpose, BSON was invented to bridge the gap: a **binary representation to store data in JSON format**, optimized for **speed, space, and flexibility**.

1. **What is BSON?**

[**BSON**](http://bsonspec.org/) simply stands for “**Binary JSON**” and that’s exactly what it was invented to be. BSON’s binary structure encodes type and length information, which allows it to be **parsed much more quickly**.

1. **Does MongoDB use BSON or JSON?**

**MongoDB** stores data in **BSON** format **both internally** and **over the network**.

**Ref:** [**https://www.mongodb.com/json-and-bson**](https://www.mongodb.com/json-and-bson)

1. **JSON vs BSON**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **JSON** | **BSON** |
| 1. Type | Standard **file** format. | **Binary** format. |
| 1. Speed | Comparatively **less fast.** | **Faster.** |
| 1. Space | Consumes Comparatively **less space.** | **More space** is consumed. |
| 1. Usages | **Transmission** of data. | **Storage** of data. |
| 1. Encoding and Decoding Technique | No such technique. | **Faster** Encoding and Decoding Technique. |
| 1. Characteristic | **Key value pair only** used for transmission of data. | **Lightweight, fast** and **traversable**. |
| 1. Parse | JSON formats **need not to be parsed** as they are human readable format already. | BSON on the other hand **needs to be parsed** as they are easy for machines to parse and generate. |

**Ref:** [**https://www.educba.com/json-vs-bson/**](https://www.educba.com/json-vs-bson/)

**2. Fundamentals of MongoDB**

1. **Show databases**

The below commands show the available Mongo databases in our system, which have at-list one collection.

**> show dbs**

**admin 0.000GB**

**config 0.000GB**

**local 0.000GB**

1. **Switching to your database**

The below command is use to get switch on the specific database, if it not present then it create that database and then switch on it.

**> use thapatechnical**

**switched to db thapatechnical**

1. **Check active database**

The below command will show current active database or if there is no database is active it simply shows `**test**` in console.

**> db**

**thapatechnical**

**or**

**> db**

**test**

1. **Check collection in active database**

The below command will show the list of collection in current active database.

**> show collections**

**thapadata**

**3. CRUD Operations**

**Create Operations:**

1. **Creating collection and inserting single document:**

The below command will create a collection in current database and insert only one document with random ID.

**> db.thapadata.insertOne({name:"ReactJS", type:"Front End", videos:80, active:true})**

**{**

**"acknowledged" : true,**

**"insertedId" : ObjectId("60afd86c868d48e8b4e7e3eb")**

**}**

1. **Creating collection and inserting multiple document:**

The below command will create a collection in current database and insert multiple document with random ID.

**> db.thapadata.insertMany([{name:"MongoDB", type:"Database", videos:5, active:true}, {name:"NodeJS", type:"Back End", videos:40, active:true}])**

**{**

**"acknowledged" : true,**

**"insertedIds" : [**

**ObjectId("60afe2095f2f7704e6dbdf6d"),**

**ObjectId("60afe2095f2f7704e6dbdf6e")**

**]**

**}**

**Read Operations:**

**Syntax: db.collection.find(query, projection)**

1. **How to show documents present in collection of active database?**

The below command will show the list of documents present in collection of active database.

**> db.thapadata.find()**

**{ "\_id" : ObjectId("60afe2095f2f7704e6dbdf6d"), "name" : "MongoDB", "type" : "Database", "videos" : 5, "active" : true }**

**{ "\_id" : ObjectId("60afe2095f2f7704e6dbdf6e"), "name" : "NodeJS", "type" : "Back End", "videos" : 40, "active" : true }**

**{ "\_id" : ObjectId("60afe6945f2f7704e6dbdf6f"), "name" : "ReactJS", "type" : "Front End", "videos" : 80, "active" : true }**

1. **How to documents present in collection in pretty format**

The below command will show the list of documents present in collection in pretty format.

**> db.thapadata.find().pretty()**

**{**

**"\_id" : ObjectId("60afe2095f2f7704e6dbdf6d"),**

**"name" : "MongoDB",**

**"type" : "Database",**

**"videos" : 5,**

**"active" : true**

**}**

**{**

**"\_id" : ObjectId("60afe2095f2f7704e6dbdf6e"),**

**"name" : "NodeJS",**

**"type" : "Back End",**

**"videos" : 40,**

**"active" : true**

**}**

**{**

**"\_id" : ObjectId("60afe6945f2f7704e6dbdf6f"),**

**"name" : "ReactJS",**

**"type" : "Front End",**

**"videos" : 80,**

**"active" : true**

**}**

1. **Search document with specific field of data e.g. “MongoDB”.**

The below command will show the list of document with specific field of data if **query** matched **else** **blank/null**.

**> db.thapadata.find({name:"MongoDB"})**

**{ "\_id" : ObjectId("60afe2095f2f7704e6dbdf6d"), "name" : "MongoDB", "type" : "Database", "videos" : 5, "active" : true }**

1. **Search document with specific field of data e.g. “MongoDB” with specific field e.g. “name”.**

The below command will show the list of document with specific field of data with specific field **if** **query** matched **else** **blank/null**.

**> db.thapadata.find({name:"MongoDB"}, {name:1})**

**{ "\_id" : ObjectId("60afe2095f2f7704e6dbdf6d"), "name" : "MongoDB" }**

1. **Search document with specific field of data e.g. “MongoDB” with specific field e.g. “name” without “\_id” field in it.**

The below command will show the list of document with specific field of data with specific field and without “\_id” field in it **if** **query** matched **else** **blank/null**.

**> db.thapadata.find({name:"MongoDB"}, {\_id:0, name:1})**

**{ "name" : "MongoDB" }**

1. **Search first or only single document with specific field of data e.g. “active:true” in collection.**

The below command will show the first or only single document with specific field of data with specific field **if** **query** matched **else** **blank/null**.

**> db.thapadata.find({active:true}).limit(1)**

**{ "\_id" : ObjectId("60afe2095f2f7704e6dbdf6d"), "name" : "MongoDB", "type" : "Database", "videos" : 5, "active" : true }**

if you not pass any argument it will show all the documents present inside collection.

**--- Or --- there is another method**

**> db.thapadata.findOne({active:true})**

**{**

**"\_id" : ObjectId("60afe2095f2f7704e6dbdf6d"),**

**"name" : "MongoDB",**

**"type" : "Database",**

**"videos" : 5,**

**"active" : true**

**}**

1. **Search documents by skipping 1 result.**

The below command will show the documents by skipping first result.

**> db.thapadata.find({active:true}).limit(1).skip(1)**

**{ "\_id" : ObjectId("60afe2095f2f7704e6dbdf6e"), "name" : "NodeJS", "type" : "Back End", "videos" : 40, "active" : true }**

**Update Operations:**

**Syntax:**

**To Update One => db.collection\_name.updateOne(<filter>, <update>)**

**To Update Many => db.collection\_name.updateOne(<filter>, <update>)**

1. **Update only one document**

The below command will update fields of the single document in present collection. The **$set** operator replace the value of a field with the specified value.

**> db.thapadata.updateOne({name:"JavaScript"}, {$set: {type:"Full Stack"}})**

**{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }**

1. **Update multiple document**

The below command will update fields of multiple document in present collection. The **$set** operator replace the value of a field with the specified value.

**> db.thapadata.updateMany({type:"Front End"},{$set: {active:false}})**

**{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }**

**Delete Operations:**

**Syntax: db.collection\_name.updateOne(deletion\_criteria)**

1. **Update only one document**

The below command will delete fields from the document in present collection.

**> db.thapadata.deleteMany({type:"Full Stack"})**

**{ "acknowledged" : true, "deletedCount" : 1 }**

1. **Remember if we not pass any deletion criteria it will delete each and every documents from present collection.**

The below command will delete all documents from present collection.

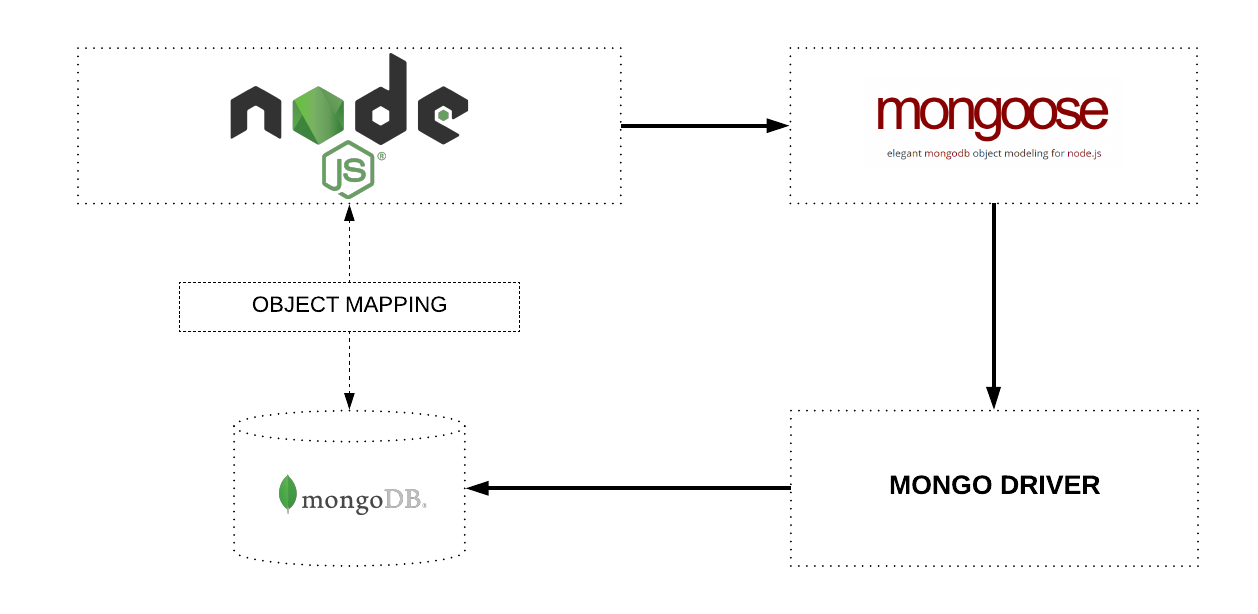
**> db.thapadata.deleteMany({})**

**{ "acknowledged" : true, "deletedCount" : 3 }**

**4. Introduction to Mongoose**

1. **What is Mongoose?**

Mongoose is an Object Data Modelling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.



Mongoose provide an abstraction layer on top of MongoDB that eliminates the need to use named collections.

**Ref:** [**https://medium.com/free-code-camp/introduction-to-mongoose-for-mongodb-d2a7aa593c57#:~:text=Mongoose%20is%20an%20Object%20Data,of%20those%20objects%20in%20MongoDB**](https://medium.com/free-code-camp/introduction-to-mongoose-for-mongodb-d2a7aa593c57#:~:text=Mongoose%20is%20an%20Object%20Data,of%20those%20objects%20in%20MongoDB)

1. **Install Mongoose:**

The following command is use to install npm package of mongoose in your node application.

**> npm i mongoose**

1. **Create connection to your MongoDB:**

The following code is use to make connection from your node application to your MongoDB server.

const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/thapatech",  {useNewUrlParser:true, useUnifiedTopology:true })

.then(() => console.log("Connection successfull..."))

.catch((err) => console.log(err));

**5. Schema and Models Mongoose**

1. **Defining Schema:**

A Mongoose schema defines the structure of document i.e. default values, validators, etc. The following code represent that how we can define the structure of our document for Mongoose.

const playlistSchema = new mongoose.Schema({

    name: {

        type: String,

        required: true

    },

    ctype: String,

    videos: Number,

    author: String,

    active: Boolean,

    date: {

        type:Date,

        default: Date.now

    }

})

1. **Mongoose Model:**

A Mongoose model is a wrapper on the Mongoose schema. A Mongoose schema defines the structure of the document, default values, validators, etc., whereas a Mongoose model provides an interface to the database for creating, querying, updating, deleting records, etc.

const Playlist = new mongoose.model("Playlist", playlistSchema);

Note: The parameter that we have passed in ‘mongoose.model’ function i.e. ‘Playlist’ will automatically converts into ‘Playlists’ in MongoDB database.

**6. Create and Insert Document using Mongoose**

1. **Create and Insert single document:**

The following code will create new document inside MongoDB.

const reactPlaylist = new Playlist({

name: "Node JS",

ctype: "Back End",

      videos: 50,

      author: "Thapa Technical",

      active: true

})

const result = await reactPlaylist.save();

1. **Create and Insert multiple documents:**

The following code will create new documents inside MongoDB.

const mongoosePlaylist = new Playlist({

     name: "Mongoose JS",

      ctype: "Database",

      videos: 4,

      author: "Thapa Technical",

      active: true

})

const expressPlaylist = new Playlist({

      name: "Express JS",

      ctype: "Back End",

      videos: 20,

      author: "Thapa Technical",

      active: true

})

const result = await Playlist.insertMany([jsPlaylist, mongoPlaylist, mongoosePlaylist, expressPlaylist]);

console.log(result);

**7. Reading Document using Mongoose**

1. **Reading documents:**

The following code will read the documents inside MongoDB.

const result = await Playlist.find({name:"React JS"}).select({name:1});

console.log(result);

**8. Reading Document: Comparing Query Operators**

**For comparison of different BSON type values, see the specified BSON comparison order.**

| **Name** | **Description** |
| --- | --- |
| $eq | Matches values that are equal to a specified value. |
| $gt | Matches values that are greater than a specified value. |
| $gte | Matches values that are greater than or equal to a specified value. |
| $in | Matches any of the values specified in an array. |
| $lt | Matches values that are less than a specified value. |
| $lte | Matches values that are less than or equal to a specified value. |
| $ne | Matches all values that are not equal to a specified value. |
| $nin | Matches none of the values specified in an array. |

1. **$gt**

**Syntax:** {field: {$gt: value} }

const result = await Playlist.find({videos: {$gt:50}});

console.log(result);

1. **$in**

**Syntax:** { field: { $in: [<value1>, <value2>, ... <valueN> ] } }

const result = await Playlist.find({ctype: {$in:["Back End", "Front End"]}});

console.log(result);

**9. Reading Document: Logical Query Operators**

| **Name** | **Description** |
| --- | --- |
| [$and](https://docs.mongodb.com/manual/reference/operator/query/and/#mongodb-query-op.-and) | Joins query clauses with a logical AND returns all documents that match the conditions of both clauses. |
| [$not](https://docs.mongodb.com/manual/reference/operator/query/not/#mongodb-query-op.-not) | Inverts the effect of a query expression and returns documents that do *not* match the query expression. |
| [$nor](https://docs.mongodb.com/manual/reference/operator/query/nor/#mongodb-query-op.-nor) | Joins query clauses with a logical NOR returns all documents that fail to match both clauses. |
| [$or](https://docs.mongodb.com/manual/reference/operator/query/or/#mongodb-query-op.-or) | Joins query clauses with a logical OR returns all documents that match the conditions of either clause. |

1. **$or:**

The [$or](https://docs.mongodb.com/manual/reference/operator/query/or/#mongodb-query-op.-or) operator performs a logical OR operation on an array of two or more <expressions> and selects the documents that satisfy at least one of the <expressions>.

**Syntax:** { $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }

const result = await Playlist.find(

     {$or:[{ctype:"Back End"},{author:"true"}]}

);

console.log(result);

1. **$and:**

The $and performs a logical AND operation on an array of one or more expressions (e.g. <expression1>, <expression2>, etc.) and selects the documents that satisfy all the expressions in the array.

**Syntax:** { $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }

const result = await Playlist.find(

      {$and:[{ctype:"Back End"}, {author:"Thapa Technical"}]}

);

console.log(result);

1. **$nor:**

[**$nor**](https://docs.mongodb.com/manual/reference/operator/query/nor/#mongodb-query-op.-nor) performs a logical NOR operation on an array of one or more query expression and selects the documents that **fail** all the query expressions in the array.

const result = await Playlist.find(

      {$nor:[{ctype:"Back End"}, {ctype:"Front End"}]}

);

console.log(result);

1. **$not:**

[**$not**](https://docs.mongodb.com/manual/reference/operator/query/not/#mongodb-query-op.-not) performs a logical NOT operation on the specified <operator-expression> and selects the documents that do **not** match the <operator-expression>. This includes documents that do not contain the field.

const result = await Playlist.find(

      {videos: {$not: {$gte:20}}}

);

console.log(result);

**10. Reading Document: Sorting and Count Query Methods**

1. **Count number of documents:**

The following code will use to count number of documents present in current collection.

const result = await Playlist.find(

{videos: {$not: {$gte:20}}}

)

.countDocuments();

console.log(result);

1. **$sort:**

[$sort](https://docs.mongodb.com/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) takes a document that specifies the field(s) to sort by and the respective sort order. <sort order> can have one of the following values:

| **Value** | **Description** |
| --- | --- |
| 1 | Sort ascending. |
| -1 | Sort descending. |
| { $meta: "textScore" } | Sort by the computed textScore metadata in descending order. See Text Score Metadata Sort for an example. |

const result = await Playlist

        .find({author:"Thapa Technical"})

        .select({name:1})

        .sort({name:-1})

console.log(result);

**11. Update the Document using Mongoose**

1. **updateOne(filter, update, options):**

The below code will update fields of the single document in present collection. The **$set** operator replace the value of a field with the specified value.

const result = await Playlist.updateOne({\_id:"60b3bab16f4ec81ff8718a7d"}, {

     $set : {name: "Javascript"}

});

console.log(result);

**output:**

{ n: 1, nModified: 0, ok: 1 }

1. **findByIdAndUpdate(filter, update, option):**

The below code will find document by its **id** and the update fields of the document in present collection. In **option** parameter **useFindAndModify:false** is compulsory otherwise give deprecated error and **new:true** is use return modified document rather than the original one in output.

const result = await Playlist.findByIdAndUpdate(

{\_id:"60b3bab16f4ec81ff8718a7d"},

{$set : {name: "Javascript Thapa"}},

{useFindAndModify: false, new: true}

);

console.log(result);

**output:**

{

\_id: 60b3bab16f4ec81ff8718a7d,

name: 'Javascript Thapa',

ctype: 'Front End',

videos: 150,

author: 'Thapa Technical',

active: true,

date: 2021-05-30T16:17:53.306Z,

\_\_v: 0

}

**12. Delete the document in mongoose**

1. **deleteOne():**

The below command will delete the document in present collection.

const result = await Playlist.deleteOne(

{\_id: "60b3bab16f4ec81ff8718a7d"}

);

console.log(result);

**output:**

{ n: 1, ok: 1, deletedCount: 1 }

1. **findByIdAndDelete();**

The below command will first find the document by its **Id** and delete the document in present collection.

const result = await Playlist.findByIdAndDelete(

      {\_id: "60b3b59e28bc633ae4f77d6d"}

);

console.log(result);

**output:**

{

\_id: 60b3b59e28bc633ae4f77d6d,

name: 'Node JS',

ctype: 'Back End',

videos: 50,

author: 'Thapa Technical',

active: true,

date: 2021-05-30T15:56:14.979Z,

\_\_v: 0

}

**13. Mongoose built in validation using MongoDB**

1. **unique:**

The **unique** option is not a validator, it’s a convenient helper for building MongoDB unique indexes.

const playlistSchema = new mongoose.Schema({

    name: {

        type: String,

        required: true,

        unique: true

    }

1. **lowercase:**

The **lowercase** option will only work for string to make value of field in lowercase.

const playlistSchema = new mongoose.Schema({

    name: {

        type: String,

        required: true,

        unique: true,

        lowercase: true

    },

1. **trim:**

The **trim** option will only work for string to remove the leading and trailing space from the string.

const playlistSchema = new mongoose.Schema({

    name: {

        type: String,

        required: true,

        unique: true,

        lowercase: true,

        trim: true,

    },

1. **minlength:**

The **minlength** option will limit your string with minimum require length for input value of your fields. The second argument in array is to provide custom validation.

const playlistSchema = new mongoose.Schema({

    name: {

        type: String,

        required: true,

        unique: true,

        lowercase: true,

        trim: true,

        minlength: [2, "minimum 2 letters allowed"]

    },

1. **enum:**

The **enum** option will always be type of array and allowed only that value from input those are present in the array.

ctype: {

        type: String,

        required: true,

        enum: ["frontend", "backend", "database"]

    },

**14. Mongoose custom validation**

1. **validate():**

The **validate** function is use to add the custom validation to validate the values of fields of the documents.

videos: {

    type: Number,

    validate(value){

          if(value < 0){

            throw new Error("Video(s) count should not negative");

        }

    }

}

**15. Mongoose custom validation: validator npm**

1. **validator:**

**This library validates and sanitizes strings only. It can be install using `npm i validation`.**

const validator = require("validator");

email: {

        type: String,

        required: true,

        unique: true,

        validate(value){

            if(!validator.isEmail(value)){

                throw new Error("Email is invalid");

            }

        }

    }