**MongoDB Tutorial:**

Data is the collection of information. Data can be in the form of text, media, number, bytes etc.

Database: It is the collection of data that is organised in such a way that it can be easily accessed and manipulated.

**Type of Databases:**

1. **Centralised Database**: It is a database stored and managed at a single location, such as a central server or data center. It ensures higher security.
2. **Distributed Database:** Data is distributed over a network.
3. **Relational (SQL) Database:** Data is organized into tables with rows and columns, similar to a spreadsheet. E.g. MySQL, PostgreSQL, Oracle, SQL Server.
4. **Object oriented Database:** Data is stored as objects, which can contain both data and methods (functions). E.g. Gemstone, ObjectStore etc.
5. **Cloud Database:** Databases that are deployed and managed in a cloud environment (e.g., AWS, Azure, Google Cloud).
6. **NoSQL (Not Only SQL) Database: :** Data is organized into various models like document, key-value, graph, and column-oriented, but does not rely on the **traditional tabular structure**. E.g. MongoDB (document), Redis (key-value), Neo4j (graph), Cassandra (column-family). NoSQL is used for handling large volumes of unstructured or semi-structured data, real-time web applications, and high-availability systems that require horizontal scalability.

**MongoDB:**

* MongoDB is an open-source and cross-platform **document-oriented** **NoSQL** database System.
* MongoDB stores data in a [JSON](https://www.geeksforgeeks.org/json/)document structure form which makes it easy to operate with dynamic and unstructured data.
* **MongoDB Compass**is a powerful **graphical user interface** for **MongoDB**which is designed to simplify **database management**. It offers features like **easy querying, index visualization,**an aggregation pipeline builder and schema and structure analysis.
* **MongoDB Atlas** is a fully-managed cloud database service provided by MongoDB. It simplifies database management by hosting your MongoDB database on the cloud, eliminating the need for local hosting.

**MongoDB Installation:**

1. Download and install MongoDB Community Server from internet (<https://www.mongodb.com/try/download/community>)
2. After installation, go to the location where MongoDB installed in your system and copy **bin**path.

(C:\Program Files\MongoDB\Server\8.0\bin)

1. Set Environment Variables: Open **system** **properties >> Environment Variable >> System variable >> path >> Edit >> new** -> paste the copied link to your environment system and **click Ok**
2. open the command prompt and run the command: mongod
3. If you get error, create “data” folder in ‘C’ drive and then create ‘db’ folder in data folder.
4. Download and install MongoDB Shell from internet (<https://www.mongodb.com/try/download/shell>)
5. Run the MongoDB Shell using command: mongosh on terminal window
6. You are now connected to the MongoDB shell.

**Commands for working with databases:**

1. Create a Database: use database\_name

The **use** **Database\_name**command makes a new database in the system if it does not exist, if the database exists it uses that database

1. To see all databases, use command: show dbs
2. To see name of database in which we are currently working: db
3. To delete a database: db.dropDatabase()
4. To see all collections in a database: use database\_name -> show collections
5. To create new collection: db.createCollection(<collection\_name>)
6. To delete a collection: db.<collection\_name>.drop()
7. Insert 1 document to a Collection: db.<collection\_name>.insertOne({“field”: “value”})

The**db.Collection\_name** command makes a new collection in the database and the insertOne() method inserts the document in the collection.

1. Insert multiple documents:

db.<collection\_name>.insertMany([

{ field: value },

{ field: value }

])

1. To see all data or documents in a collection: db.<collection\_name>.find()
2. To find a particular document: db.<collection\_name>.find({‘field’: ‘value’})
3. To view only 3 document: db. <collection\_name>.find().limit(3)
4. To find number of documents in a collection: db. <collection\_name>.find().count()
5. Display only a particular field of the document:

db.<collection\_name>.find({‘field’: ‘value’}, {‘field\_print’: 1} )

1 = true i.e. show only that field, 0 = false i.e. show all fields except this

1. Update value of a field:

db.<collection\_name>.updateOne({‘field’:’oldValue’}, {$set: {‘field’:‘newValue’}})

1. Update value of multiple fields:

db.<collection\_name>.updateMany({‘field’:’oldValue’}, {$set: {‘field’:‘newValue’}})

1. Delete one document:

db.<collection\_name>.deleteOne({‘field’: ‘value’})

1. Delete multiple documents:

db.<collection\_name>.deleteMany({‘field’: ‘value’})

1. Delete all documents from a collection:

db.<collection\_name>.deleteMany({})

1. Sorting Documents: db.<collection\_name>.find().sort({‘field\_to\_sort’: 1})

1 for ascending order, & -1 for descending order

1. Greater than / less than: db.<collection\_name>.find({‘field’: {$gt: ‘value’}})

$gt = greater than, $gte = greater than equal, $lt = less than, $lte = less than equal

**Mongoose JS:**

Mongoose is an Object Document Mapper (ORM), which means that we can write javascript objects and it will convert them into JSON and store them in database.

We use Mongoose because:

* It is built on MongoDB for seamless integration with Node.js applications.
* Provides schema-based modeling to define document structure.
* Includes built-in validation to ensure data consistency.
* Enables easy querying and data relationships with chain query methods.

Node.js 🡪 Mongoose 🡪 Mongo drivers 🡪 MongoDB

**Connecting Mongoose with Database:**

1. Initialize Your Node.js Project with command: npm init -y
2. Install Dependencies: npm install mongoose express
3. Import Mongoose: const mongoose = require('mongoose');
4. Connect to Database: mongoose.connect(mongodb://localhost:27017/<database\_name>)
5. Mongoose.connect() returns a promise.

Example:

const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017")

.then((res)=>{

    console.log("Connection established successfully.")

})

.catch((err)=>{

    console.log(err);

})

**Schema:**

A mongoose schema defines the structure of the document, default values, validations etc.

It is a way to express expected properties and values.

Syntax:

const <schema\_name> = mongoose.Schema({

key1: Data type,

key2: { type: Data type,

default: “default value”

}

})

**Model:**

A mongoose model is a wrapper on mongoose schema.

It defines a programming interface for interacting with the database like operations – read, insert, update, delete etc.

Schema answer- “ what will the data look like in the collection?”

Model provides functionality like CRUD in the document.

Syntax:

const <modal\_name> = new mongoose.model(“<collection\_name>”, <schema\_name>);

**Add Data to the Database Using Mongoose:**

1. Inserting 1 document at a time:

const <Document\_name> = new <model\_name>({

key1 : ‘value’,

key2 : ‘value’

})

<Document\_name>.save();

2. Inserting multiple document at once:

const <Document\_name1> = new <model\_name>({ key : ‘value’ })

const <Document\_name2> = new <model\_name>({ key : ‘value’ })

<model\_name>.insertMany([

<Document\_name1>,

<Document\_name2>

])

**Note**: Inserting a document is an asynchronous task, so always use them in async function with try-catch block.

**Reading or Querying Documents:**

Since this is an asynchronous process, we will create a async function to perform the task.

const getDocument = async()=>{

const data = await <model.name>.find();

console.log(data);

}

getDocument();

**Updating Documents:**

Since this is an asynchronous process, we will create an async function to perform the task.

const updateDoc = async (id) =>{

try {

const result = await <modelName>.updateOne({\_id: id}, {$set: {field: 'value'}});

        console.log(result);    // returns status

    }

    catch (error) {

        console.log(error);

    }

}

updateDoc(“id”);

**Note:** updateOne() does not return the updated data. So, we use “findByIdAndUpdate()” with parameter {new: true} to get updated data.

const result = await <modelName>.findByIdAndUpdate({\_id: id}, {$set: {field: 'value'}}, {new: true});

**Deleting Documents:**

Since this is an asynchronous process, we will create an async function to perform the task.

const deleteDocument = async (id) =>{

const result = await <modelName>.deleteOne({\_id: id});

console.log(result);    // returns status

}

deleteDocument (“id”);

**Schema Validations:**

Schema validation lets you create validation rules for your fields, such as allowed data types and value ranges. This helps maintain data consistency and integrity by verifying the data before it is saved to the database.

Properties:

1. Required: It will make the field mandatory to be filled.
2. Lower Case: It will convert all the string data into lowercase.
3. Upper Case: It will convert all the string data into uppercase.
4. Trim: it trims down the empty spaces from starting and ending.
5. Minlength: It checks if the input string has minimum character or not.
6. Maxlength: It checks if the input string has maximum character or not.
7. Min: check if the input number is greater than or equal to min number.
8. Max: check if the input number is less than or equal to max number.
9. Enum: Provide arrays of possible values, if input does not match any value it shows error.

https://bitbuilder.hashnode.space/

1. **Operators in MongoDB**
2. **Indexing in MongoDB**
3. **Query Optimization in MongoDB**
4. **Aggregation**
5. **ACID Properties**
6. CAP Theorem
7. Embeded vs Reference Documents
8. Sharding and Replica Set
9. Authentication and Authorization
10. Json Web Tokens
11. OAuth 2.0
12. CORS (Cross Origin Resource Sharing)
13. Http Status code
14. Idempotency

**APIs and Authentications:**

* The primary purpose of a GET request in API communication is to retrieve data or resources from the server. A GET request is read-only, meaning it does not modify any data on the server.
* The body of a POST request typically contains the data that is sent to the server, such as JSON or XML data. This is used when creating or updating resources on the server.
* express.json() is the middleware in Express.js used to parse incoming JSON data in the body of the request. It is used to automatically parse the JSON payload from a POST request so that it can be accessed via req.body.
* **express-validator** is a popular middleware in Express.js used to validate and sanitize user input data sent in the request body. It allows you to define validation rules for the input fields and handle errors effectively in your Express route
* **JSON Web Tokens (JWT)** are commonly used in MERN stack applications for **authentication** and **authorization**. JWTs allow a server to authenticate users via a signed token, ensuring that users are who they say they are. After successful login, the server issues a JWT, which is then sent with every API request (usually in the Authorization header). This helps the server verify the user's identity and permissions for each request.
* OAuth is a protocol used for delegated access, typically involving third-party authentication (like Google or Facebook login), but it's not the most common method for authentication in MERN applications.
* Basic Auth is less secure compared to JWTs and is typically not recommended for handling user authentication in modern web applications due to its vulnerability to interception.
* In a MERN stack application, soft deletion is commonly implemented by adding a deleted or isDeleted field to the MongoDB schema, and then using Mongoose hooks (or middleware) to intercept delete operations. Instead of actually removing the resource from the database, the deleted field is set to true or a timestamp is recorded. This allows you to "soft delete" resources while keeping the data intact.