Introduction about the Databases

What is a Database?

A **database** is a structured collection of data that allows for easy access, management, and updating. It is used in almost every application to store user data, application data, logs, etc.

SQL (Relational) Databases

Examples: MySQL, PostgreSQL, Oracle, SQL Server

SQL (Structured Query Language) databases are **relational**. Data is stored in **tables (rows and columns)** with predefined schemas.

Why use SQL databases?

- **Structured data:** Fixed schema, strict data types
- **Relationships:** You need to manage relationships between entities (e.g., users and orders)
- **ACID compliance:** Strong consistency, transactions, reliability
- **Complex queries:** Advanced joins, filters, and aggregations

When to use SQL:

- Banking systems
- E-commerce sites
- Enterprise resource planning (ERP)
- Applications needing complex queries and transactions

NoSQL (Non-relational) Databases

Examples: MongoDB, Cassandra, Redis, Firebase, CouchDB

NoSQL databases are **non-relational**. Data can be stored in formats like **JSON, key-value pairs, wide-columns, or graphs**.

Why use NoSQL databases?

- **Flexible schema:** Easy to store unstructured or semi-structured data
- **High scalability:** Better for horizontal scaling (adding more servers)
- **Fast performance** for large-scale data
- **Different data models:** Key-value, document, graph, column-based

When to use NoSQL:

- Real-time big data applications
- Social media platforms
- IoT data storage
- Content management systems (CMS)
- When schema changes frequently

In short:

Feature	SQL	NoSQL
Data Structure	Tables (structured)	JSON, key-value, etc. (flexible)
Schema	Fixed	Dynamic
Scaling	Vertical	Horizontal
Use Case	Transactions, relational data	Big data, real-time analytics
Example	Banking, e-commerce	Social media, chat apps

What is MongoDB?

MongoDB is a popular **NoSQL database** that stores data in a **flexible, JSON-like format** called **BSON** (Binary JSON).

Instead of storing data in tables like SQL, MongoDB stores data in **collections**, and each record is called a **document**.

Key Concepts:

- **Database**: Just like in SQL, a MongoDB database holds collections.
- **Collection**: Similar to a table in SQL.
- **Document**: Similar to a row in SQL, but it's stored in JSON format (key-value pairs).

Example document:

```
{
  "name": "Vinay",
  "age": 21,
  "skills": ["Python", "MongoDB", "Flask"]
}
```

Why Use MongoDB?

1. **Schema-less**: You don't need to define the structure of data up front.
2. **Scalable**: Easy to scale horizontally (add more servers).
3. **Flexible**: Each document can have different fields.
4. **Fast**: Good for real-time applications and large volumes of data.
5. **Good for nested data**: You can store arrays and objects within documents.

When to Use MongoDB?

- Applications with **rapidly changing data**
- **Big Data** and analytics platforms
- **Content Management Systems (CMS)**
- **IoT applications** with varied or large data
- **Real-time apps** like chat applications or social networks

---

Real-life Examples:

- Storing user profiles in a social media app
- Product catalogs in e-commerce
- Logging system or user activity tracking
- Inventory management for large stores

Sure! Let's go step-by-step to connect **MongoDB with Node.js**, starting with the **native MongoDB driver** and then using **Mongoose**.

---

1. Using Native MongoDB Driver

Step 1: Install the MongoDB Driver

```
npm install mongodb
```

Step 2: Connect to MongoDB

Here's a simple example:

```
// db.js
const { MongoClient } = require('mongodb');

const uri = "mongodb://localhost:27017"; // or your MongoDB Atlas URI
const client = new MongoClient(uri);

async function connectDB() {
  try {
    await client.connect();
    console.log("Connected to MongoDB");

    const db = client.db("mydatabase");
    const collection = db.collection("users");

    // Example: Insert a document
    await collection.insertOne({ name: "Vinay", age: 21 });

    // Example: Find all documents
    const users = await collection.find().toArray();
    console.log(users);

  } catch (err) {
    console.error(err);
  } finally {
    await client.close();
  }
}

connectDB();
```

---

2. Using Mongoose

Mongoose is an ODM (Object Data Modeling) library that makes it easier to work with MongoDB in Node.js by providing schema and models.

Step 1: Install Mongoose

```
npm install mongoose
```

Step 2: Connect to MongoDB with Mongoose

```
// app.js
const mongoose = require('mongoose');

mongoose.connect("mongodb://localhost:27017/mydatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB connected"))
.catch(err => console.log(err));

// Define a schema
const userSchema = new mongoose.Schema({
  name: String,
  age: Number
});

// Create a model
const User = mongoose.model('User', userSchema);

// Create and save a user
const user = new User({ name: "Vinay", age: 21 });
user.save().then(() => console.log("User saved"));
```

Difference Summary

Feature	MongoDB Native Driver	Mongoose
Schema	No schema enforcement	Schema-based models
Learning Curve	Lower	Slightly higher (uses abstraction)
Flexibility	High	Moderate
Ideal for	Simple or small projects	Medium-large apps with structure

---

1. CRUD Operations Using MongoDB Native Driver

Step 1: Install MongoDB Driver

If you haven't already installed the MongoDB native driver, do it first:

```
npm install mongodb
```

Step 2: Example CRUD Operations

```
// db.js
const { MongoClient } = require('mongodb');

const uri = "mongodb://localhost:27017"; // or your MongoDB Atlas URI
const client = new MongoClient(uri);
const dbName = 'mydatabase';
const collectionName = 'users';

async function connectDB() {
  try {
    await client.connect();
    console.log("Connected to MongoDB");

    const db = client.db(dbName);
    const collection = db.collection(collectionName);

    // CREATE: Insert a new user
    await collection.insertOne({ name: "Vinay", age: 21 });
    console.log("User inserted");

    // READ: Find all users
    const users = await collection.find().toArray();
    console.log("Users found:", users);

    // UPDATE: Update a user's age
    await collection.updateOne({ name: "Vinay" }, { $set: { age: 22 } });
    console.log("User updated");

    // DELETE: Delete a user
    await collection.deleteOne({ name: "Vinay" });
    console.log("User deleted");

  } catch (err) {
    console.error(err);
  } finally {
    await client.close();
  }
}

connectDB();
```

2. CRUD Operations Using Mongoose

Step 1: Install Mongoose

If you haven’t already installed **Mongoose**, do it first:

```
npm install mongoose
```

Step 2: Example CRUD Operations

```
// app.js
const mongoose = require('mongoose');

mongoose.connect("mongodb://localhost:27017/mydatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB connected"))
.catch(err => console.log(err));

// Define a schema
const userSchema = new mongoose.Schema({
  name: String,
  age: Number
});

// Create a model
const User = mongoose.model('User', userSchema);

// CREATE: Insert a new user
async function createUser() {
  const user = new User({ name: "Vinay", age: 21 });
  await user.save();
  console.log("User inserted:", user);
}

// READ: Find all users
async function readUsers() {
  const users = await User.find();
  console.log("Users found:", users);
}

// UPDATE: Update a user's age
async function updateUser() {
  const updatedUser = await User.updateOne({ name: "Vinay" }, { $set: { age: 22 } });
  console.log("User updated:", updatedUser);
}

// DELETE: Delete a user
async function deleteUser() {
  const result = await User.deleteOne({ name: "Vinay" });
  console.log("User deleted:", result);
}

// Call CRUD functions
async function runCRUDOperations() {
  await createUser();
  await readUsers();
  await updateUser();
  await deleteUser();
}

runCRUDOperations();
```

CRUD Summary

Operation	MongoDB Native Driver	Mongoose
Create	`insertOne()` or `insertMany()`	`new Model()` + `.save()`
Read	`find()`, `findOne()`, `findOneAndUpdate()`	`.find()`, `.findOne()`, `.findById()`

Operation	MongoDB Native Driver	Mongoose
Update	`updateOne()`, `updateMany()`	`.updateOne()`, `.updateMany()`
Delete	`deleteOne()`, `deleteMany()`	`.deleteOne()`, `.deleteMany()`

Which to use?

- Use **MongoDB native driver** for simpler, low-level control over your database.
- Use **Mongoose** when you need more structure (e.g., schema validation, middleware, or relationships between models).