
Chapter 5: Introduction to Databases (MongoDB)

A server alone cannot make a web application functional. While it can respond to requests, perform calculations, or serve HTML pages, it cannot remember anything without a **database**. Imagine building an online store that can display products, manage customer orders, or track users—without a database, all the information would vanish when the server restarts. Databases are the foundation of dynamic web applications because they store, organize, and allow manipulation of persistent data.

In this chapter, we will explore **databases**, why they are necessary, the differences between relational and non-relational databases, and how **MongoDB**—a popular, beginner-friendly NoSQL database—works. We will go step by step, starting from the basics, and cover practical implementation using Node.js and Mongoose.

---

1. What is a Database and Why is it Important?

A database is an organized collection of data stored in a structured manner, allowing you to store, retrieve, update, and delete information efficiently. Every application, from social media platforms to e-commerce websites, relies on databases to maintain data like:

- User profiles
- Product listings
- Messages or posts
- Transaction history

Without a database, the server has **no memory**, and every request is independent. Databases ensure that data is:

1. **Persistent**: Stored permanently and not lost when the server restarts.
2. **Organized**: Structured in a way that makes it easy to access and manage.
3. **Secure**: Protected from unauthorized access.

4. **Efficient**: Quickly accessible, even with large amounts of data.
5. **Scalable**: Able to grow as your application grows.

---

2. Types of Databases

Databases are broadly classified into two categories:

A. Relational Databases (SQL)

- Data is stored in **tables** with rows and columns.
- Relationships between tables are defined (foreign keys).
- Schema is **fixed**, meaning each table has a predefined structure.
- Example: MySQL, PostgreSQL, Oracle.

Example Table: Users

id	name	email	age
1	John Doe	john@example.com	30
2	Alice Smith	alice@example.com	28

B. Non-Relational Databases (NoSQL)

- Data is stored in **documents**, **key-value pairs**, or **graphs**.
- Schema is **flexible**, meaning each record can have a different structure.
- Ideal for large-scale applications.
- Example: MongoDB, Firebase.

MongoDB is a **document-based NoSQL database**, meaning it stores data in JSON-like documents, which is perfect for JavaScript developers since it resembles JavaScript objects.

---

3. Understanding MongoDB

MongoDB organizes data in a hierarchical structure:

- **Database:** A container for collections. Example: `mydatabase`.
- **Collection:** A group of documents. Example: `users`.
- **Document:** An individual record stored in JSON-like format. Example:

```
{
  "_id": "unique_id",
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30
}
```

Key Features of MongoDB

- **Flexible schema:** Each document can have different fields.
- **Scalable:** Can handle large amounts of data across multiple servers.
- **Indexing:** Speeds up queries.
- **Aggregation:** Allows complex queries and data analysis.

---

4. Setting Up MongoDB

Step 1: Installing MongoDB

You can install MongoDB locally on your computer. After installation, start the MongoDB service:

```
mongod
```

This command runs the MongoDB server in the background.

Step 2: Creating a Node.js Project

Initialize a Node.js project:

npm init -y

This creates a `package.json` file to manage dependencies.

Step 3: Installing Mongoose

Mongoose is a library that simplifies connecting Node.js to MongoDB and provides a structured way to define data models:

npm install mongoose

---

5. Connecting the Server to MongoDB

Create a file named `server.js`:

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

// Connection URL
const dbURI = 'mongodb://127.0.0.1:27017/mydatabase';

// Connect to MongoDB
mongoose.connect(dbURI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB successfully!'))
  .catch(error => console.error('Error connecting to MongoDB:', error));
```

Explanation:

- `mongoose.connect()` establishes the connection.
- Options `useNewUrlParser` and `useUnifiedTopology` ensure a stable connection.
- `.then()` runs if the connection is successful.
- `.catch()` handles errors during connection.

Run the server:

node server.js

You should see:
“Connected to MongoDB successfully!”

---

6. Creating a Schema and Model

In MongoDB, data is stored in **collections**. A schema defines the structure of documents in a collection, while a model provides a way to interact with that collection.

Create a folder `models` and a file `User.js`:

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

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

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

module.exports = User;
```

Explanation:

- `mongoose.Schema` defines the fields and their types.
- `mongoose.model` creates a model for interacting with the collection.
- The model will be used to create, read, update, and delete documents.

---

7. CRUD Operations in MongoDB

CRUD stands for **Create, Read, Update, Delete**, the fundamental operations for any database.

A. Create

```
const User = require('./models/User');

const newUser = new User({
  name: 'Alice Smith',
  email: 'alice@example.com',
```

```
  age: 28
});

newUser.save()
  .then(user => console.log('User saved:', user))
  .catch(error => console.error('Error saving user:', error));
```

- `.save()` inserts the document into the collection.
- Errors are caught and displayed.

B. Read

Retrieve all users:

```
User.find()
  .then(users => console.log('All users:', users))
  .catch(error => console.error('Error retrieving users:', error));
```

Retrieve a single user:

```
User.findOne({ email: 'alice@example.com' })
  .then(user => console.log('Found user:', user))
  .catch(error => console.error('Error finding user:', error));
```

C. Update

```
User.updateOne({ email: 'alice@example.com' }, { age: 29 })
  .then(result => console.log('Update result:', result))
  .catch(error => console.error('Error updating user:', error));
```

- Updates the `age` field of the matching document.

D. Delete

```
User.deleteOne({ email: 'alice@example.com' })
  .then(result => console.log('User deleted:', result))
  .catch(error => console.error('Error deleting user:', error));
```

- Deletes the document that matches the criteria.

---

8. Connecting CRUD Operations to the Server

We can integrate MongoDB operations into an **Express server** to make the database interactive.

Install Express:

npm install express body-parser

Create `server.js`:

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const User = require('./models/User');

const app = express();
const dbURI = 'mongodb://127.0.0.1:27017/mydatabase';

app.use(bodyParser.json());

mongoose.connect(dbURI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB!'))
  .catch(err => console.error(err));

// POST - Create user
app.post('/users', (req, res) => {
  const newUser = new User(req.body);
  newUser.save()
    .then(user => res.status(201).json({ message: 'User created!', user }))
    .catch(err => res.status(400).json({ error: err.message }));
});

// GET - Retrieve all users
app.get('/users', (req, res) => {
  User.find()
    .then(users => res.status(200).json(users))
    .catch(err => res.status(500).json({ error: err.message }));
});

// PUT - Update a user
app.put('/users/:email', (req, res) => {
  User.updateOne({ email: req.params.email }, req.body)
    .then(result => res.status(200).json({ message: 'User updated!', result }))
    .catch(err => res.status(400).json({ error: err.message }));
});
```

```
// DELETE - Delete a user
app.delete('/users/:email', (req, res) => {
  User.deleteOne({ email: req.params.email })
    .then(result => res.status(200).json({ message: 'User deleted!', result }))
    .catch(err => res.status(400).json({ error: err.message }));
});

app.listen(3000, () => console.log('Server running on http://localhost:3000'));
```

Explanation:

- The server listens for HTTP requests (GET, POST, PUT, DELETE).
- Each route performs a MongoDB operation using the model.
- The server sends JSON responses to the client.
- Front-end applications can interact with these endpoints to manage data dynamically.

---

9. Practical Example: Contact Form Integration

Imagine a website with a contact form where users submit messages. MongoDB can store these messages:

1. Define a `Message` schema:

```
const messageSchema = new mongoose.Schema({
  name: String,
  email: String,
  message: String,
  date: { type: Date, default: Date.now }
});

const Message = mongoose.model('Message', messageSchema);
```

2. Create a POST route to save messages:

```
app.post('/messages', (req, res) => {
  const newMessage = new Message(req.body);
  newMessage.save()
    .then(msg => res.status(201).json({ message: 'Message received!', msg }))
```

```
    .catch(err => res.status(400).json({ error: err.message }));
});
```

3. The server saves messages, and the front-end can display confirmation to the user.

---

10. Security and Best Practices

1. **Validate Data:** Always validate inputs to prevent malicious data.
2. **Sanitize Data:** Remove harmful code from user input.
3. **Use HTTPS:** Encrypt data sent over the network.
4. **Use Environment Variables:** Store database credentials securely, not in plain code.
5. **Error Handling:** Handle errors gracefully and avoid exposing sensitive information.

Example of using environment variables:

```
require('dotenv').config();
const dbURI = process.env.MONGO_URI;
mongoose.connect(dbURI);
```

---

11. Summary

By the end of this chapter, you have learned:

- The role of a **database** in web applications.
- Differences between **relational** and **non-relational databases**.
- How **MongoDB** stores data in documents, collections, and databases.
- How to set up MongoDB locally and connect it to a Node.js server.
- Creating schemas and models with **Mongoose**.
- Performing **CRUD operations** (Create, Read, Update, Delete).

- Integrating MongoDB operations with **Express routes**.
- Practical real-world examples such as storing contact form submissions.
- Basic security best practices.

MongoDB provides the backbone for dynamic, data-driven web applications. Understanding databases is essential before building complete back-end systems, handling user accounts, or connecting complex front-end interfaces.
