Introduction to Node.js

What is Node.js?

- Node.js is an open-source, cross-platform runtime for JavaScript.
- It is built on Chrome's V8 JavaScript engine.
- Uses event-driven, non-blocking I/O model, making it efficient and scalable.
- Used for building APIs, microservices, real-time applications, and server-side scripting.

How to Install Node.js?

1. Download **Node.js** from the official website: https://nodejs.org/
2. Install it (LTS version is recommended).

Verify installation:

```
node -v
npm -v
```

Your First Node.js Program

1. Create a new file: **app.js**

Write this simple script:

```
console.log("Hello, Node.js!");
```

2.

Run it using:

```
node app.js
```

2. Node.js Modules

Node.js has built-in modules to handle different functionalities.

Types of Modules in Node.js

1. **Core Modules** (Built-in like fs, http, path, events)
2. **Local Modules** (Custom modules you create)
3. **Third-party Modules** (Installed via NPM like express, mongoose)

How to Use Core Modules?

Example: Using the os module

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

console.log("OS Type:", os.type());
console.log("Total Memory:", os.totalmem());
console.log("Free Memory:", os.freemem());
console.log("Home Directory:", os.homedir());
```

3. HTTP Module (Creating a Web Server)

Node.js has an http module that allows you to create web servers.

Example: Creating a Simple Web Server

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

const server = http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello, World!');
});

server.listen(3000, () => {
    console.log('Server running at http://localhost:3000/');
});
```

- Run the file: node app.js
- Open http://localhost:3000 in a browser.

4. URL Module (Handling URLs)

The url module helps in parsing and resolving URLs.

Example: Parsing a URL

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

const address = 'http://localhost:3000/about?name=Ved&age=30';
const parsedUrl = url.parse(address, true);

console.log("Pathname:", parsedUrl.pathname);
console.log("Query Params:", parsedUrl.query);
console.log("Name:", parsedUrl.query.name);
console.log("Age:", parsedUrl.query.age);
```

5. File System (FS Module)

The fs module allows you to read, write, and manipulate files.

Example: Reading a File

Create test.txt with some content.

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

fs.readFile('test.txt', 'utf8', (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
});

Example: Writing to a File
javascript
CopyEdit
fs.writeFile('output.txt', 'Hello from Node.js!', (err) => {
    if (err) throw err;
    console.log("File written successfully!");
});
```

6. NPM (Node Package Manager)

NPM helps in managing dependencies (libraries).

Check NPM Version

```
npm -v
```

Installing a Package

Example: Install axios (for making HTTP requests)

```
npm install axios
```

Using Installed Package

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

axios.get('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => console.log(response.data))
    .catch(error => console.error(error));
```

7. Events and Event Emitters

Node.js follows an **event-driven architecture** using the events module.

Example: Creating a Custom Event

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();

eventEmitter.on('greet', (name) => {
    console.log(`Hello, ${name}!`);
});

eventEmitter.emit('greet', 'Ved');
```

8. Exception Handling (Error Handling)

Errors should be handled properly using try...catch.

Example: Handling Errors

```
try {
    let x = y + 10; // y is not defined
} catch (error) {
    console.error("Error Occurred:", error.message);
}
```

Handling Async Errors

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

fs.readFile('nonexistent.txt', 'utf8', (err, data) => {
    if (err) {
        console.error("Error:", err.message);
        return;
    }
    console.log(data);
});
```

Axios in Node.js

Axios is a popular **promise-based HTTP client** for making **asynchronous HTTP requests** in Node.js and the browser.

Why Use Axios?

- Supports **Promise-based API** (easy to use with async/await).
- Handles **automatic JSON parsing**.
- Provides **error handling** and **request cancellation**.
- Supports **interceptors** to modify requests/responses globally.
- Works in **both browser and Node.js**.

1. Installing Axios

Before using Axios, install it using NPM:

```
npm install axios
```

2. Making GET Requests

A **GET request** is used to fetch data from an API.

Example: Fetching Data from a Public API

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

axios.get('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => {
        console.log("Data:", response.data);
    })
    .catch(error => {
        console.error("Error fetching data:", error.message);
    });
```

Output:

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi
optio reprehenderit",
  "body": "quia et suscipit..."
}
```

response.data contains the actual API response.
catch(error) handles errors (like network issues or invalid URLs).

3. Making GET Requests with Query Parameters

You can pass query parameters using params.

Example: Fetching Posts of a Specific User

```
axios.get('https://jsonplaceholder.typicode.com/posts', {
    params: { userId: 1 }
})
.then(response => {
    console.log(response.data);
})
.catch(error => {
    console.error("Error:", error.message);
});
```

This fetches **all posts** where userId = 1.

4. Making POST Requests (Sending Data)

A **POST request** is used to send data to a server.

Example: Creating a New Post

```
const newPost = {
    title: 'My First Post',
    body: 'This is the content of my first post.',
    userId: 1
};

axios.post('https://jsonplaceholder.typicode.com/posts', newPost)
    .then(response => {
        console.log("Post Created:", response.data);
    })
    .catch(error => {
        console.error("Error creating post:", error.message);
    });
```

The **request body** is sent in JSON format.
The server **creates a new post** and returns it.

5. Making PUT Requests (Updating Data)

A **PUT request** updates an existing resource.

Example: Updating an Existing Post

```
const updatedPost = {
    title: 'Updated Post Title',
    body: 'Updated post content.',
    userId: 1
};

axios.put('https://jsonplaceholder.typicode.com/posts/1',
updatedPost)
    .then(response => {
        console.log("Post Updated:", response.data);
    })
    .catch(error => {
        console.error("Error updating post:", error.message);
    });
```

The **entire resource is replaced** with the new data.

6. Making PATCH Requests (Partially Updating Data)

A **PATCH request** updates **only specific fields**.

Example: Updating Only the Title

```
axios.patch('https://jsonplaceholder.typicode.com/posts/1', {
title: 'New Title' })
    .then(response => {
        console.log("Post Updated:", response.data);
    })
    .catch(error => {
        console.error("Error updating post:", error.message);
    });
```

Only **the title** is updated (unlike PUT, which replaces everything).

7. Making DELETE Requests

A **DELETE request** removes a resource.

Example: Deleting a Post

```
axios.delete('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => {
        console.log("Post Deleted:", response.data);
    })
    .catch(error => {
        console.error("Error deleting post:", error.message);
    });
```

The post is removed, and the response usually returns {} or a success message.

8. Using Axios with async/await

Instead of using .then() and .catch(), you can use async/await.

Example: Fetching Data with async/await

```
async function fetchData() {
    try {
        const response = await
axios.get('https://jsonplaceholder.typicode.com/posts/1');
        console.log("Data:", response.data);
    } catch (error) {
        console.error("Error fetching data:", error.message);
    }
}

fetchData();
```

await pauses execution until the request is complete.
try/catch is used for error handling.

9. Setting Global Defaults

You can set **default headers, base URLs, and timeouts** globally.

Example: Setting Global Defaults

```
axios.defaults.baseURL = 'https://jsonplaceholder.typicode.com';
axios.defaults.headers.common['Authorization'] = 'Bearer
my-token';
axios.defaults.timeout = 5000; // 5 seconds timeout

axios.get('/posts/1')
    .then(response => console.log(response.data))
    .catch(error => console.error("Error:", error.message));
```

10. Axios Interceptors

Interceptors **modify requests or responses globally**.

Example: Logging Every Request

```
axios.interceptors.request.use(request => {
    console.log("Request Made:", request);
    return request;
}, error => {
    return Promise.reject(error);
});

axios.get('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => console.log("Response:", response.data));
```

11. Handling Errors in Axios

You can **handle errors** based on response status.

Example: Checking HTTP Status Codes

```
axios.get('https://jsonplaceholder.typicode.com/invalid-endpoint')
    .then(response => console.log(response.data))
    .catch(error => {
        if (error.response) {
            console.log("Error Status:", error.response.status);
            console.log("Error Data:", error.response.data);
        } else {
            console.log("Network Error:", error.message);
        }
    });
```

- error.response exists if the server sent a response.
- If no response, it's a **network issue**.

Feature	**Method**
Fetch data	axios.get(url)
Send data	axios.post(url, data)
Update full data	axios.put(url, data)
Update partial data	axios.patch(url, data)

Delete data	axios.delete(url)
Async/Await	await axios.get(url)
Global Defaults	axios.defaults
Interceptors	axios.interceptors.request/use
Error Handling	try/catch or .catch(error => { ... })

Why Use MySQL with Node.js?

- **Structured Data:** Ideal for storing relational data.
- **Scalability:** Can handle large datasets efficiently.
- **Performance:** Fast queries with indexes and optimizations.

1. Introduction

Why Use MySQL with Node.js?

- **Structured Data:** Ideal for storing relational data.
- **Scalability:** Can handle large datasets efficiently.
- **Performance:** Fast queries with indexes and optimizations.
- **Compatibility:** Works well with Node.js for backend applications.

2. Setup MySQL & Node.js

Step 1: Install MySQL

1. Download MySQL from: https://dev.mysql.com/downloads/
2. Install MySQL Server & MySQL Workbench.
3. Start MySQL Server.

Open **MySQL Workbench** or **Command Line** and log in:

```
mysql -u root -p
```

(Enter your password when prompted.)

Step 2: Install Node.js

- Download & install from: https://nodejs.org/

Check installation:

```
node -v
```

Step 3: Create a New Node.js Project

Open a terminal and create a new folder for the project:

```
mkdir mysql-node-app
cd mysql-node-app
```

Initialize Node.js:

```
npm init -y
```

Install mysql2 package:

```
npm install mysql2
```

3. Connect Node.js to MySQL

Create a db.js File

Create a new file **db.js** inside your project folder.

```
const mysql = require('mysql2');
// Create MySQL Connection
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'your_password', // Change this to your MySQL
password
    database: 'testdb' // We will create this database in the next
step
});
// Connect to MySQL
connection.connect((err) => {
    if (err) {
        console.error('Database Connection Failed:', err.message);
        return;
    }
    console.log('Connected to MySQL Database!');
});

module.exports = connection;
```

Change **your_password** to your actual MySQL password.

4. Create a Database in MySQL

Run This in MySQL Workbench or Terminal

```
CREATE DATABASE testdb;
```

This creates a **database named testdb**.

Modify db.js to Use This Database

```
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'your_password',
    database: 'testdb' // Updated
});
```

5. Create a Table

Let's create a table named **users**.

Create a createTable.js File

Create a new file **createTable.js** in your project.

```
const connection = require('./db');
const createTableQuery = `
CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    age INT
);
`;
```

```
connection.query(createTableQuery, (err, result) => {
    if (err) {
        console.error('Error Creating Table:', err.message);
        return;
    }
    console.log('Users Table Created Successfully!');
    connection.end();
});
```

Run It

```
node createTable.js
```

6. Insert Data into Table

Create an insert.js File

```
const connection = require('./db');
const insertUserQuery = `INSERT INTO users (name, email, age)
VALUES (?, ?, ?)`;
const userData = ['Leena Nadkar', 'ln@gmail.com', 30];
connection.query(insertUserQuery, userData, (err, result) => {
    if (err) {
        console.error('Error Inserting Data:', err.message);
        return;
    }
    console.log('User Inserted Successfully! ID:',
result.insertId);
    connection.end();
});
```

Run It

```
node insert.js
```

7. Select Data from Table

Create a select.js File

```
const connection = require('./db');
const selectQuery = `SELECT * FROM users`;
connection.query(selectQuery, (err, results) => {
    if (err) {
        console.error('Error Fetching Data:', err.message);
        return;
    }
    console.log('Users:', results);
    connection.end();
});
```

Run It

```
node select.js
```

8. Update Data in Table

Create an update.js File

```
const connection = require('./db');
const updateQuery = `UPDATE users SET age = ? WHERE email = ?`;
const updateData = [35, 'vedraut11@gmail.com'];
connection.query(updateQuery, updateData, (err, result) => {
    if (err) {
        console.error('Error Updating Data:', err.message);
        return;
    }
    console.log('Rows Affected:', result.affectedRows);
    connection.end();
});
```

Run It

```
node update.js
```

9. Delete Data from Table

Create a delete.js File

```
const connection = require('./db');

const deleteQuery = `DELETE FROM users WHERE email = ?`;
const deleteData = ['vedraut11@gmail.com'];

connection.query(deleteQuery, deleteData, (err, result) => {
    if (err) {
        console.error('Error Deleting Data:', err.message);
        return;
    }
    console.log('Rows Deleted:', result.affectedRows);
    connection.end();
});
```

Run It

```
node delete.js
```

10. Using WHERE Clause

```
Get Users with Age > 25
const connection = require('./db');

const query = `SELECT * FROM users WHERE age > ?`;
const age = [25];

connection.query(query, age, (err, results) => {
    if (err) {
        console.error('Error Fetching Data:', err.message);
        return;
    }
    console.log('Filtered Users:', results);
    connection.end();
});
```

11. Using ORDER BY Clause

```
Get Users Sorted by Age (Descending)
const connection = require('./db');

const query = `SELECT * FROM users ORDER BY age DESC`;

connection.query(query, (err, results) => {
    if (err) {
        console.error('Error Fetching Data:', err.message);
        return;
    }
    console.log('Sorted Users:', results);
    connection.end();
});
```

12. Drop a Table

To delete the users table completely.

```
Create a dropTable.js File
const connection = require('./db');
const dropTableQuery = `DROP TABLE IF EXISTS users`;
connection.query(dropTableQuery, (err, result) => {
    if (err) {
        console.error('Error Dropping Table:', err.message);
        return;
    }
    console.log('Users Table Dropped!');
    connection.end();
});
```

Run It

```
node dropTable.js
```

1. Install Homebrew (Package Manager)

If you haven't installed Homebrew, install it first:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh
)"
```

Verify installation:

```
brew -v
```

2. Install MySQL (Community Server)

Install MySQL using Homebrew:

```
brew install mysql
```

After installation, start the MySQL server:

```
brew services start mysql
```

Check if MySQL is running:

```
mysql -u root -p
```

(If prompted for a password, just press Enter for default installation.)

3. Install MySQL Workbench (Optional)

Download from MySQL Workbench Official Site

Alternatively, install via Homebrew:

```
brew install --cask mysqlworkbench
```

4. Install Node.js and NPM

```
brew install node
```

Verify installation:

```
node -v
npm -v
```

5. Install MySQL Node.js Connector

```
npm install mysql2
```

6. Test MySQL Connection in Node.js

Create a file **test-db.js**:

```
const mysql = require('mysql2');
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '', // Use your MySQL root password (leave empty if
none)
    database: 'testdb'
});

connection.connect((err) => {
    if (err) {
        console.error('Database connection failed:', err.message);
        return;
    }
    console.log('Connected to MySQL!');
    connection.end();
});
```

Run it:

```
node test-db.js
```

1. Restart MySQL in Safe Mode (Skip Password Authentication)

```
brew services stop mysql
mysqld_safe --skip-grant-tables &
```

2. Login Without Password

Since grant tables are skipped, you can now log in without a password.

```
mysql -u root
```

3. Reset Root Password

Once inside MySQL, run:

```
USE mysql;

ALTER USER 'root'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'newpassword';

FLUSH PRIVILEGES;
```

```
EXIT;
```

Replace **newpassword** with your preferred password.

4. Restart MySQL

Stop the safe mode and restart MySQL normally:

```
brew services stop mysql
brew services start mysql
```

5. Test Connection

Try logging in again with the new password:

```
mysql -u root -p
```

(Enter your new password)

Or test it in Node.js:

```
const mysql = require('mysql2');
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'newpassword', // Use your updated password
    database: 'testdb'
});

connection.connect((err) => {
    if (err) {
        console.error('Database connection failed:', err.message);
        return;
    }
    console.log('Connected to MySQL!');
    connection.end();
});
```

Run:

```
node test-db.js
```

6. If You Still Face Issues

Try setting up MySQL with **socket authentication**:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH auth_socket;
FLUSH PRIVILEGES;
```

Then log in without a password:

```
mysql -u root
```

Step 1: Open MySQL Command Line

Option 1: Using Terminal (Recommended)

Stop MySQL Service:
brew services stop mysql

```
brew services stop mysql
```

Start MySQL in Safe Mode (Without Password Authentication):

```
mysqld_safe --skip-grant-tables &
```

This starts MySQL without checking passwords, allowing you to reset them.

Login to MySQL (No Password Required):

```
mysql -u root
```

If successful, you will see a **MySQL prompt** (mysql>), meaning you are inside the MySQL CLI.

Step 2: Reset MySQL Root Password

Once inside the MySQL CLI, run these SQL commands:

```
USE mysql;
```

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'newpassword';
```

```
FLUSH PRIVILEGES;
EXIT;
```

Replace 'newpassword' with your desired password.

Step 3: Restart MySQL

Now, stop MySQL safe mode and restart it normally:

```
brew services stop mysql
brew services start mysql
```

Step 4: Verify Password

Try logging in with the new password:

```
mysql -u root -p
```

(Enter the new password when prompted.)

How to Get Your Existing MySQL Password?

If MySQL was installed with a password and you forgot it:

Check **MySQL config file**:

```
cat ~/.my.cnf
```

If it contains something like:

```
[client]
user=root
password=yourpassword
```

- You can use that password.

If no password is set, MySQL may be using **socket authentication**:

```
SELECT user, host, plugin FROM mysql.user;
```

If the plugin is **auth_socket**, you don’t need a password, just run:

```
mysql -u root
```

Step 5: Update Your Node.js MySQL Connection

Now, update your **db.js** in your Node.js project:

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'newpassword', // Use your new MySQL password
    database: 'testdb'
});

connection.connect((err) => {
    if (err) {
        console.error('Database connection failed:', err.message);
        return;
    }
    console.log('Connected to MySQL!');
    connection.end();
});
```

Run:

```
node db.js
```

If your MySQL installation does **not require a password**, update your **db.js** file like this:

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '', // Empty string for no password
    database: 'testdb'
});
```

```
connection.connect((err) => {
    if (err) {
        console.error('Database connection failed:', err.message);
        return;
    }
    console.log('Connected to MySQL!');
    connection.end();
});
```

Run

```
node db.js
```

Change MySQL to Use Password Authentication (Optional)

If you want to **enable password-based login**, run this inside MySQL:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'mypassword';
FLUSH PRIVILEGES;
EXIT;
```

Then, update your **db.js** to:

```
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'mypassword',
    database: 'testdb'
})
```

1. Introduction to NoSQL Databases

What is NoSQL?

NoSQL (Not Only SQL) databases are designed to handle unstructured and semi-structured data. Unlike traditional relational databases (SQL), NoSQL databases do not rely on fixed schema tables.

Types of NoSQL Databases

1. **Document-Based** (e.g., MongoDB) → Stores data as JSON-like documents.
2. **Key-Value Stores** (e.g., Redis) → Uses key-value pairs.
3. **Column-Family Stores** (e.g., Apache Cassandra) → Uses columns and rows but in a distributed manner.
4. **Graph-Based** (e.g., Neo4j) → Uses nodes and relationships.

Why MongoDB?

- **Flexible Schema**: JSON-like storage (BSON format).
- **Scalability**: Horizontal scaling via sharding.
- **Fast Reads/Writes**: Uses indexing for better performance.
- **Easy Integration**: Works well with Node.js and other modern applications.

2. Installing MongoDB on macOS

Step 1: Install Homebrew (if not installed)

Homebrew is a package manager for macOS.

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh
)"
```

Verify the installation:

```
brew --version
```

Step 2: Install MongoDB

Run the following command:

```
brew tap mongodb/brew
brew install mongodb-community@7.0
```

Verify the installation:

```
mongod --version
```

Step 3: Start MongoDB Service

To start MongoDB as a background service:

```
brew services start mongodb-community@7.0
```

To stop MongoDB:

```
brew services stop mongodb-community@7.0
```

To restart MongoDB:

```
brew services restart mongodb-community@7.0
```

Step 4: Check MongoDB is Running

Open a new terminal window and type:

```
mongosh
```

If it works, you should see a MongoDB shell prompt (>) where you can run commands.

3. MongoDB Overview

Key Features

- **Collections**: Groups of documents (similar to SQL tables).
- **Documents**: Individual records (similar to SQL rows).
- **Indexes**: Improve search speed.
- **Aggregation**: Performs operations like SUM, AVG, COUNT.
- **Replication**: Ensures data redundancy.

Data Storage Format

MongoDB uses **BSON (Binary JSON)**, which extends JSON with additional data types.

Example document:

```
{
  "_id": ObjectId("507f191e810c19729de860ea"),
  "name": "Ved Raut",
  "age": 30,
  "city": "Mumbai",
  "skills": ["React", "Node.js", "MongoDB"]
}
```

4. MongoDB Data Types

Common data types in MongoDB:

- **String** → "name": "Ved"
- **Number** → "age": 30
- **Boolean** → "isActive": true
- **Array** → "skills": ["React", "Node.js"]
- **Object** → { "address": { "city": "Mumbai", "pincode": 400092 }}
- **Date** → "createdAt": ISODate("2024-03-07T00:00:00Z")
- **ObjectId** → "id": ObjectId("507f191e810c19729de860ea")

5. MongoDB CRUD Operations

Create (Insert)

```
use myDatabase
db.users.insertOne({ "name": "Ved", "age": 30, "city": "Mumbai" })
db.users.insertMany([
  { "name": "Amit", "age": 28, "city": "Delhi" },
  { "name": "Pooja", "age": 25, "city": "Pune" }
])
```

Read (Find)

```
db.users.find()  # Get all documents
db.users.find({ "city": "Mumbai" })  # Get specific documents
db.users.find().pretty()  # Pretty-print the output
Update
db.users.updateOne({ "name": "Ved" }, { $set: { "age": 31 } })
db.users.updateMany({ "city": "Mumbai" }, { $set: { "status":
"Active" } })
```

Delete

```
db.users.deleteOne({ "name": "Ved" })
db.users.deleteMany({ "city": "Mumbai" })
```

6. Indexing in MongoDB

Indexes improve search performance.

Create an Index

```
db.users.createIndex({ "name": 1 })  # 1 for ascending order
db.users.createIndex({ "city": -1 })  # -1 for descending order
```

Check Indexes

```
db.users.getIndexes()
```

7. Aggregation in MongoDB

Aggregation is like SQL's GROUP BY.

Example: Count Users by City

```
db.users.aggregate([
  { $group: { _id: "$city", totalUsers: { $sum: 1 } } }
])
```

Example: Average Age by City

```
db.users.aggregate([
  { $group: { _id: "$city", avgAge: { $avg: "$age" } } }
])
```

8. MongoDB with Node.js

Step 1: Install Node.js

Check if you have Node.js installed:

```
node -v
```

If not installed, install it using Homebrew:

```
brew install node
```

Step 2: Install MongoDB Driver

Inside your project folder, run:

```
npm init -y
npm install mongodb
```

Step 3: Connect to MongoDB

Create a file app.js and add the following code:

```
const { MongoClient } = require("mongodb");

const url = "mongodb://localhost:27017";
const client = new MongoClient(url);

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

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

    // Insert a document
    await collection.insertOne({ name: "Ved", age: 30, city:
"Mumbai" });

    // Fetch all users
    const users = await collection.find().toArray();
    console.log(users);
  } finally {
    await client.close();
  }
}

run().catch(console.dir);
```

Step 4: Run the Node.js Script

```
node app.js
```

"SyntaxError: Missing semicolon."

db.users.find()

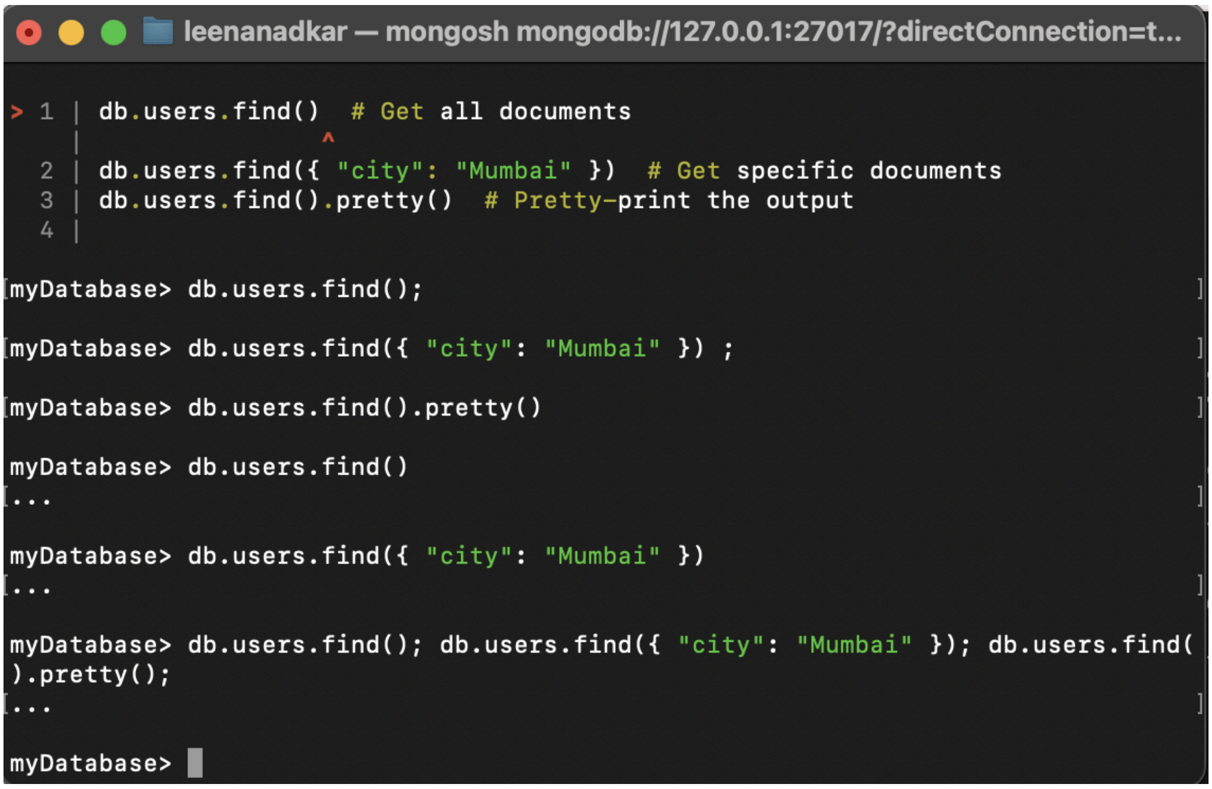
db.users.find({ "city": "Mumbai" })

db.users.find().pretty()

db.users.find(); db.users.find({ "city": "Mumbai" }); db.users.find().pretty();

Why Did This Happen?

- The **MongoDB shell (mongosh) follows JavaScript syntax**, and it expects a semicolon (;) to separate multiple commands.
- If you write multiple statements on separate lines **without pressing Enter** in between, the shell treats it as a multi-line command.
- If the syntax is incorrect, it results in an **uncaught SyntaxError**.



```
leenanadkar — mongosh mongodb://127.0.0.1:27017/?directConnection=t...

> 1 | db.users.find()  # Get all documents
    |                  ^
  2 | db.users.find({ "city": "Mumbai" })  # Get specific documents
  3 | db.users.find().pretty()  # Pretty-print the output
  4 |

myDatabase> db.users.find();

myDatabase> db.users.find({ "city": "Mumbai" }) ;

myDatabase> db.users.find().pretty()

myDatabase> db.users.find()
...

myDatabase> db.users.find({ "city": "Mumbai" })
...

myDatabase> db.users.find(); db.users.find({ "city": "Mumbai" }); db.users.find(
).pretty();
...

myDatabase> 
```

1. Install MySQL on Windows

If MySQL is not installed, download it from the official website:

🔗 **Download MySQL:**
https://dev.mysql.com/downloads/installer/

- Choose the **MySQL Installer for Windows**.
- Install **MySQL Server** and **MySQL Workbench** (optional).

2. Add MySQL to System PATH (Optional)

By default, MySQL commands won't work globally in CMD. You can add MySQL to your system PATH to run MySQL commands from any location.

Steps:

1. **Find the MySQL bin path:**

   Open File Explorer and navigate to:
   C:\Program Files\MySQL\MySQL Server 8.0\bin

   - Copy this path.
2. **Add to Windows Environment Variables:**
   - Search for **"Environment Variables"** in the Windows Start menu.
   - Click on **"Edit the system environment variables"**.
   - In the **System Properties** window, go to the **Advanced** tab and click **Environment Variables**.
   - Under **System Variables**, select **Path** → Click **Edit**.
   - Click **New**, then paste the MySQL **bin path** (C:\Program Files\MySQL\MySQL Server 8.0\bin).
   - Click **OK** to save.
3. **Restart CMD** to apply changes.

3. Open MySQL in CMD

Method 1: If MySQL is in System PATH

1. Open **Command Prompt (cmd)**.

   Type:

```
mysql -u root -p
```

2. Enter your **MySQL root password** when prompted.

Method 2: If MySQL is NOT in System PATH

Navigate to the **MySQL bin folder** in CMD:

```
cd "C:\Program Files\MySQL\MySQL Server 8.0\bin"
```

Run MySQL:

```
mysql -u root -p
```

3. Enter your password when prompted.

4. Basic MySQL Commands

Check Available Databases

```
SHOW DATABASES;
Create a New Database
CREATE DATABASE mydatabase;
Use a Database
USE mydatabase;
```

Create a Table

```
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    email VARCHAR(100),
    age INT
);
```

Insert Data

```
INSERT INTO users (name, email, age) VALUES
('Ved', 'ved@example.com', 30),
('Amit', 'amit@example.com', 28);
```

View Data

```
SELECT * FROM users;
```

Exit MySQL

```
exit
```

5. Restart MySQL Service (If Needed)

If MySQL is not running, restart the service:

```
net stop mysql
net start mysql
```

Or, use PowerShell as Administrator:

Restart-Service MySQL80

1. Install MongoDB on Windows

Step 1: Download MongoDB

🔗 **Official Download Page:**
https://www.mongodb.com/try/download/community

- Choose **MongoDB Community Server**.
- Select the **Windows MSI Installer**.
- Install with the **Complete** setup option.

2. Add MongoDB to System PATH (Optional)

By default, MongoDB is installed in

```
C:\Program Files\MongoDB\Server\7.0\bin
```

To use mongo or mongod commands from **anywhere in CMD**, add this path to your **System Environment Variables**:

Steps:

1. Open **Start Menu** → Search **"Environment Variables"**.
2. Click **"Edit the system environment variables"**.
3. In the **System Properties** window, go to the **Advanced** tab → Click **Environment Variables**.

4. Under **System Variables**, find and select **Path** → Click **Edit**.

   Click **New** → Paste the path:

```
C:\Program Files\MongoDB\Server\7.0\bin
```

5. Click **OK** and restart CMD.

   3. Start MongoDB Server

   Method 1: Using CMD

1. Open **Command Prompt (cmd)**.

   Start the MongoDB service:
   net start MongoDB

   If this command fails, try:

```
net start MongoDB
```

2. (You must create the C:\data\db folder if it doesn't exist.)
3. **Keep this window open**, as it runs the database server.

   4. Open MongoDB Shell

1. Open **a new CMD window**.

   Run:

```
mongosh
```

   OR, if using an older version:

```
mongo
```

2.

If connected successfully, you should see:

   MongoDB shell version v7.0

Basic MongoDB Commands in CMD

Check Available Databases

```
show databases
```

Create a Database

```
use myDatabase
```

Create a Collection & Insert Data

```
db.users.insertOne({ "name": "Ved", "city": "Mumbai", "age": 30 })
```

View Data

```
db.users.find().pretty()
```

Exit MongoDB Shell

```
exit
```

6. Stop MongoDB Server

To stop MongoDB

```
net stop MongoDB
```

Why Use Express.js?

- Fast and lightweight
- Easy to set up and configure
- Middleware support for request processing
- Built-in routing system
- Simple to create RESTful APIs
- Large ecosystem and community support

Installing Express.js

1. Initialize a Node.js project:

```
mkdir express-demo
cd express-demo
npm init -y
```

2. Install Express.js:

```
npm install express
```

Project Structure

```
express-demo/
├── node_modules/
├── package.json
├── server.js
```

Basic Express.js Application

Create a server.js file and write the following code:

```
const express = require('express');
const app = express();
const PORT = 3000;
// Basic route
app.get('/', (req, res) => {
    res.send('Hello, Express!');
});
// Start the server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Run the Server

```
node server.js
```

Output:

Open the browser and go to http://localhost:3000
You should see **"Hello, Express!"**

Handling Different HTTP Methods

Express allows handling various HTTP methods:

GET Request

```
app.get('/get-example', (req, res) => {
    res.send('This is a GET request');
});
```

POST Request

```
app.post('/post-example', (req, res) => {
    res.send('This is a POST request');
});
```

PUT Request

```
app.put('/put-example', (req, res) => {
    res.send('This is a PUT request');
});
```

DELETE Request

```
app.delete('/delete-example', (req, res) => {
    res.send('This is a DELETE request');
});
```

Route Parameters

You can define dynamic routes using : syntax:

```
app.get('/user/:id', (req, res) => {
    const userId = req.params.id;
    res.send(`User ID: ${userId}`);
});
```

Example:
http://localhost:3000/user/123 → Output → User ID: 123

Query Parameters

You can handle query parameters using req.query:

```
app.get('/search', (req, res) => {
    const { q, page } = req.query;
    res.send(`Search Query: ${q}, Page: ${page}`);
});
```

Example:

```
http://localhost:3000/search?q=nodejs&page=2
```

Output → Search Query: nodejs, Page: 2
Middleware

Middleware functions are functions that have access to the request and response objects.

Example of Custom Middleware:

```
const logger = (req, res, next) => {
    console.log(`${req.method} ${req.url}`);
    next(); // Move to the next middleware/route handler
};
app.use(logger);
```

Built-in Middleware:

1. **express.json()** – Parses JSON request body
2. **express.urlencoded()** – Parses URL-encoded data
3. **express.static()** – Serves static files

Example:

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static('public'));
Handling JSON Data (POST Request)
Example:
app.post('/data', (req, res) => {
    const data = req.body;
    res.send(`Received data: ${JSON.stringify(data)}`);
});
```

Request:

```
{
    "name": "Ved",
    "age": 30
}
```

Error Handling

Express provides a simple way to handle errors:

```
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something went wrong!');
});
```

Static Files Handling

You can serve static files like HTML, CSS, JS from a folder:

```
app.use(express.static('public'));
```

Create a public folder and add index.html inside it.

Routing with Router

Instead of defining all routes in server.js, you can separate them using express.Router():

routes.js

```
const express = require('express');
const router = express.Router();
```

```
router.get('/about', (req, res) => {
    res.send('About Page');
});
module.exports = router;
```

server.js

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

CORS (Cross-Origin Resource Sharing)

Use the cors package to allow requests from different domains:

```
npm install cors
```

Example:

```
const cors = require('cors');
app.use(cors());
```

Environment Variables

Use the dotenv package to manage environment variables:

```
npm install dotenv
```

Example:

```
require('dotenv').config();

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on ${PORT}`));
```

.env file:

```
PORT=4000
Complete Example
require('dotenv').config();
const express = require('express');
const cors = require('cors');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(express.json());
app.use(cors());
app.use(express.static('public'));

// Routes
app.get('/', (req, res) => {
    res.send('Welcome to Express.js');
});

app.post('/data', (req, res) => {
    const { name, age } = req.body;
    res.send(`Received data: Name - ${name}, Age - ${age}`);
});

// Error Handling
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something went wrong!');
});

// Start Server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Express, Node, MySQL Project

Step 1: Install Dependencies

Create a new project folder and initialize it:

```
mkdir playschool-management
cd playschool-management
npm init -y
```

Install necessary packages:

```
npm install express mysql2 body-parser
```

Step 2: Create MySQL Database and Table

Create the Database:

Open MySQL and create a database:

```
CREATE DATABASE playschool;
```

Create the students Table:

```
USE playschool;
```

```
CREATE TABLE students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT NOT NULL,
    class VARCHAR(50) NOT NULL,
    guardian_name VARCHAR(100) NOT NULL,
    contact_number VARCHAR(15) NOT NULL
);
```

Step 3: Create db.js for Database Connection

Create a db.js file to set up the connection with MySQL:

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'password', // Your MySQL password
  database: 'playschool'
});

connection.connect(err => {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log('Connected to MySQL database');
});

module.exports = connection;
```

Step 4: Create server.js for CRUD Operations

Create a server.js file for handling CRUD operations using Express.js:

```
const express = require('express');
const bodyParser = require('body-parser');
const db = require('./db');

const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// CREATE: Add a new student
app.post('/students', (req, res) => {
  const { name, age, class: className, guardian_name,
contact_number } = req.body;

  if (!name || !age || !className || !guardian_name ||
!contact_number) {
    return res.status(400).json({ error: 'All fields are required'
});
  }
  const query = `INSERT INTO students (name, age, class,
guardian_name, contact_number) VALUES (?, ?, ?, ?, ?)`;
  db.query(query, [name, age, className, guardian_name,
contact_number], (err, result) => {
    if (err) {
      console.error('Error inserting record:', err);
      return res.status(500).json({ error: 'Database error' });
    }

    const insertedId = result.insertId;
    db.query(`SELECT * FROM students WHERE id = ?`, [insertedId],
(err, rows) => {
      if (err) {
        console.error('Error fetching record:', err);
        return res.status(500).json({ error: 'Database error' });
      }
      res.status(201).json(rows[0]);
    });
  });
});

// READ: Get all students
app.get('/students', (req, res) => {
  const query = `SELECT * FROM students`;
  db.query(query, (err, rows) => {
    if (err) {
      console.error('Error fetching data:', err);
      return res.status(500).json({ error: 'Database error' });
    }
    res.status(200).json(rows);
  });
});

// READ: Get a specific student by ID
app.get('/students/:id', (req, res) => {
  const { id } = req.params;

  const query = `SELECT * FROM students WHERE id = ?`;
  db.query(query, [id], (err, rows) => {
    if (err) {
      console.error('Error fetching data:', err);
      return res.status(500).json({ error: 'Database error' });
    }

    if (rows.length === 0) {
      return res.status(404).json({ error: 'Student not found' });
    }

    res.status(200).json(rows[0]);
  });
});

// UPDATE: Update a student's record
app.put('/students/:id', (req, res) => {
  const { id } = req.params;
  const { name, age, class: className, guardian_name,
contact_number } = req.body;

  if (!name || !age || !className || !guardian_name ||
!contact_number) {
    return res.status(400).json({ error: 'All fields are required'
});
  }

  const query = `UPDATE students SET name = ?, age = ?, class = ?,
guardian_name = ?, contact_number = ? WHERE id = ?`;
  db.query(query, [name, age, className, guardian_name,
contact_number, id], (err, result) => {
    if (err) {
      console.error('Error updating record:', err);
      return res.status(500).json({ error: 'Database error' });
    }

    if (result.affectedRows === 0) {
      return res.status(404).json({ error: 'Student not found' });
    }

    db.query(`SELECT * FROM students WHERE id = ?`, [id], (err,
rows) => {
      if (err) {
        console.error('Error fetching data:', err);
        return res.status(500).json({ error: 'Database error' });
      }
      res.status(200).json(rows[0]);
    });
  });
});

// DELETE: Delete a student's record
app.delete('/students/:id', (req, res) => {
  const { id } = req.params;

  const query = `DELETE FROM students WHERE id = ?`;
  db.query(query, [id], (err, result) => {
    if (err) {
      console.error('Error deleting record:', err);
      return res.status(500).json({ error: 'Database error' });
    }

    if (result.affectedRows === 0) {
      return res.status(404).json({ error: 'Student not found' });
    }

    res.status(200).json({ message: 'Student deleted successfully'
});
  });
});
```

```
// Start the server
const PORT = 3000;
app.listen(PORT, () => console.log(`Server running on port
${PORT}`));
```

Step 5: Start the Server

Run the server using:

```
node server.js
```

Step 6: Test the API

1. Create a Student Record

```
curl -X POST http://localhost:3000/students \
-H "Content-Type: application/json" \
-d '{
  "name": "Aarav",
  "age": 4,
  "class": "Nursery",
  "guardian_name": "Ravi Sharma",
  "contact_number": "9876543210"
}'
```

2. Get All Students

```
curl -X GET http://localhost:3000/students
```

3. Get a Single Student by ID

```
curl -X GET http://localhost:3000/students/1
```

4. Update a Student Record

```
curl -X PUT http://localhost:3000/students/1 \
-H "Content-Type: application/json" \
-d '{
  "name": "Aarav Sharma",
  "age": 5,
  "class": "Kindergarten",
  "guardian_name": "Ravi Sharma",
  "contact_number": "9876543210"
}'
```

5. Delete a Student Record

```
curl -X DELETE http://localhost:3000/students/1
```

How to work with POSTMAN

1. Open Postman

- Open Postman on your system.
- Make sure your Node.js server is running:

```
node server.js
```

You should see:

Connected to MySQL database

Server running on port 3000

2. Create a New Student Record (POST)

Request Type: POST

Endpoint:

```
http://localhost:3000/students
```

Headers:

Key	**Value**
Content-Type	application/json

Body:

1. Select **raw → JSON** in Postman.
2. Add the following JSON data:

```
{
  "name": "Aarav",
  "age": 4,
  "class": "Nursery",
  "guardian_name": "Ravi Sharma",
  "contact_number": "9876543210"
}
```

Click "Send"

Expected Response:

- **Status: 201 Created**

{

"id": 1,

"name": "Aarav",

"age": 4,

"class": "Nursery",

"guardian_name": "Ravi Sharma",

"contact_number": "9876543210"

}

3. Get All Students (GET)

Request Type: GET

Endpoint:

```
http://localhost:3000/students
```

Click "Send"

Expected Response:

- **Status: 200 OK**

[

{

"id": 1,

"name": "Aarav",

"age": 4,

"class": "Nursery",

"guardian_name": "Ravi Sharma",

"contact_number": "9876543210"

}

]

4. Get a Single Student by ID (GET)

Request Type: GET

Endpoint:

```
http://localhost:3000/students/1
```

Click "Send"

Expected Response:

- **Status: 200 OK**

```
{
 "id": 1,
 "name": "Aarav",
 "age": 4,
 "class": "Nursery",
 "guardian_name": "Ravi Sharma",
 "contact_number": "9876543210"
}
```

5. Update a Student Record (PUT)

Request Type: PUT

Endpoint:

```
http://localhost:3000/students/1
```

Headers:

Key	**Value**
Content-Type	application/json

Body:

1. Select **raw** → **JSON** in Postman.
2. Add the following updated JSON data:

```
{
  "name": "Aarav Sharma",
  "age": 5,
  "class": "Kindergarten",
  "guardian_name": "Ravi Sharma",
  "contact_number": "9876543210"
}
```

Click "Send"

Expected Response:

- **Status: 200 OK**

{

"id": 1,

"name": "Aarav Sharma",

"age": 5,

"class": "Kindergarten",

"guardian_name": "Ravi Sharma",

"contact_number": "9876543210"

}

6. Delete a Student Record (DELETE)

Request Type: DELETE

Endpoint:

```
http://localhost:3000/students/1
```

Click "Send"

Expected Response:

- **Status: 200 OK**

  {

  "message": "Student deleted successfully"

  }

7. Test for Error Cases

Missing Fields:

- Try to send a POST request without some required fields:

  {

  "name": "Aarav"

  }

  Expected Response:

- **Status: 400 Bad Request**

```
{
  "error": "All fields are required"
}
```

Record Not Found (for GET/PUT/DELETE):

- Try to get or delete a student record with a non-existent ID:

  http://localhost:3000/students/999

Expected Response:

- **Status: 404 Not Found**

```
{
  "error": "Student not found"
}
```