A Developer's Handbook: Connecting MySQL with Node.js

Welcome to this comprehensive guide on connecting and interacting with a MySQL database using Node.js. Whether you are a beginner taking your first steps into full-stack development or an experienced programmer looking for a quick reference, this handbook will provide you with the theory, practical steps, and working code you need to get started.
By the end of this guide, you will be able to:

- Understand the fundamental concepts of relational databases and Node.js.
- Set up and configure a MySQL database and a Node.js project.
- Write Node.js code to connect to your MySQL database.
- Perform basic database operations like creating tables and inserting data.
- Generate and insert random data for testing purposes.
- Implement best practices to write secure and efficient code.

Let’s begin our journey into the world of database-driven applications!

1. Introduction: The Dynamic Duo

Overview of Relational Databases and Node.js

- **Relational Databases (like MySQL):** A relational database stores data in a structured way using tables, rows, and columns. Each table has a defined schema, meaning every row has the same columns. This structure ensures data integrity and consistency, making it ideal for applications that require well-organized data, such as e-commerce platforms, user management systems, and financial applications. MySQL is one of the most popular and widely-used open-source relational database management systems (RDBMS).
- **Node.js:** Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It is built on Google Chrome's V8 JavaScript engine and is highly efficient, non-blocking, and asynchronous. This makes Node.js an excellent choice for building fast, scalable, and data-intensive applications, especially web servers and APIs.

Why use MySQL with Node.js?

The combination of MySQL and Node.js is a powerful one. Here's why they are often used together:

- **Asynchronous Nature:** Both Node.js and the mysql2 driver (which we will use) are asynchronous. This means they can handle multiple database queries concurrently without blocking the main program thread, leading to high performance and scalability.
- **Widespread Adoption:** Both technologies have large, active communities, extensive documentation, and a vast ecosystem of libraries and tools. This makes it easy to find solutions to problems and get support.
- **Full-Stack JavaScript:** Using Node.js on the backend allows you to write your entire application, from the server logic to the front-end, in a single language: JavaScript. This

can simplify development, reduce context switching, and allow for code reuse.
- **Security and Stability:** MySQL is a mature and secure database system. When paired with the security features of the mysql2 driver, it provides a robust and reliable foundation for your application's data layer.

2. Setting Up Your Environment

Before we can start writing code, we need to have both MySQL and Node.js installed on your machine.

Installing MySQL

On Windows

1. **Download:** Go to the official MySQL website and download the **MySQL Installer for Windows**.
2. **Run Installer:** Run the downloaded .exe file. Choose a **Developer Default** setup.
3. **Configuration:** Follow the on-screen instructions. You will be asked to set a root password. **Remember this password!** It is the administrative password for your MySQL server.
4. **Finish:** Complete the installation. The installer will also set up MySQL Workbench and the MySQL Shell, which are very useful tools.

On macOS

1. **Download:** Go to the official MySQL website and download the **DMG Archive** for macOS.
2. **Install:** Open the downloaded .dmg file and run the installer.
3. **Configuration:** During the installation, you will be asked to set a root password. **Remember this password!**
4. **Start the Server:** After installation, go to **System Preferences** and find the MySQL icon. Click it to open the control panel, where you can start or stop the MySQL server.

On Linux (Ubuntu/Debian)

1. **Update Package List:** Open your terminal and run the following command to make sure your package list is up to date:
   ```
   sudo apt update
   ```
2. **Install MySQL Server:** Install the MySQL server package.
   ```
   sudo apt install mysql-server
   ```
3. **Run Security Script:** Run the security script to set the root password and other security options.
   ```
   sudo mysql_secure_installation
   ```
4. **Follow Prompts:** The script will ask you to set a root password, remove anonymous

users, disallow remote root login, and remove the test database. Follow the prompts.

Using MySQL Workbench

MySQL Workbench is a visual tool that makes managing your database much easier.

1. **Open Workbench:** Launch the application.
2. **Connect:** Click the + icon next to "MySQL Connections" to create a new connection.
   - **Connection Name:** Give it a name (e.g., My Local Connection).
   - **Hostname:** localhost
   - **Port:** 3306
   - **Username:** root
   - **Password:** Click **Store in Keychain** and enter the root password you set during installation.
3. **Test Connection:** Click **Test Connection** to ensure everything works.
4. **Creating a Database:** Once connected, you will see a query window. To create a new database, run the following SQL command:
   ```
   CREATE DATABASE delta_db;
   ```
5. **Creating a Table:** To create a table, you must first select the database you want to work with.
   ```
   USE delta_db;

   CREATE TABLE user (
     id VARCHAR(36) NOT NULL PRIMARY KEY,
     username VARCHAR(255) NOT NULL UNIQUE,
     email VARCHAR(255) NOT NULL UNIQUE,
     password VARCHAR(255) NOT NULL
   );
   ```

Using MySQL Command-Line Interface (CLI)

The CLI is a powerful way to interact with your database.

1. **Log in:** Open your terminal and log in to MySQL.
   ```
   mysql -u root -p
   ```
   You will be prompted to enter your root password.
2. **Navigate Databases:** You can see all existing databases and switch to the one you want to use.
   ```
   SHOW DATABASES;
   USE delta_db;
   ```
3. **Run .sql files:** If you have a file with SQL commands (like a schema file), you can run it directly.
   ```
   SOURCE /path/to/your/schema.sql;
   ```
   Tip: You must be logged into MySQL to use the SOURCE command.

3. Node.js Project Setup

Now let’s set up our Node.js project and install the packages we need.

Installing Node.js and npm

1. **Download:** Go to the official Node.js website and download the recommended **LTS** (Long-Term Support) version for your operating system.
2. **Install:** Run the installer and follow the on-screen instructions. npm (Node Package Manager) is installed automatically with Node.js.
3. **Verify:** Open your terminal and run the following commands to ensure everything is installed correctly.
   ```
   node -v
   npm -v
   ```
   You should see the version numbers printed.

Setting up a project

1. **Create Directory:** Create a new directory for your project.
   ```
   mkdir mysql-app
   cd mysql-app
   ```

2. **Initialize Project:** Initialize a new Node.js project.
   ```
   npm init -y
   ```
   The -y flag answers "yes" to all the setup questions, creating a package.json file with default values.

Installing Necessary Packages

We need to install several packages to build our server:
- **express**: A web framework for building APIs.
- **cors**: A middleware to enable Cross-Origin Resource Sharing.
- **mysql2**: The Node.js driver for MySQL. It's a modern, promise-based driver.
- **@faker-js/faker**: A library for generating massive amounts of fake data.

Run the following command in your terminal to install them:
```
npm install express cors mysql2 @faker-js/faker
```

4. Creating a MySQL Connection in Node.js

The core of our application is the database connection. The mysql2 library makes this straightforward.

Using mysql2.createConnection()

We use the createConnection() method to establish a connection to our MySQL database. This method takes an object with connection options.

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

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  database: 'delta_db',
  password: 'Roomee@26'
});

// A simple check to see if the connection is successful
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to the database:', err.stack);
    return;
  }
  console.log('Successfully connected to the database with ID:',
connection.threadId);
});
```

- host: The server's hostname. localhost is used for a local server.
- user: The username you use to log in to MySQL. We'll use root here, but for production, you should use a different, less-privileged user.
- database: The name of the database you want to connect to. We created delta_db earlier.
- password: The password for the specified user. **Warning:** Hardcoding passwords is not a good practice for production. For this example, it's fine.

Promises vs. Callbacks

The mysql2 library supports both traditional callbacks and modern promises. Promises are generally preferred as they make code cleaner and easier to read, especially when dealing with multiple asynchronous operations.

Callback Syntax (used in the server example):

```
connection.query('SELECT * FROM user', (err, results, fields) => {
  // Handle error or results
});
```

Promise Syntax:

```
// To use promises, you need to import the promise-based version of
the library
const mysql = require('mysql2/promise');

const connection = await mysql.createConnection({ ... });

try {
  const [rows, fields] = await connection.execute('SELECT * FROM
user');
  // Handle rows
} catch (err) {
  // Handle error
}
```

The server example uses callbacks as they are the most common in tutorials, but in the advanced section, we will show you how to refactor your code to use promises.

5. Running Queries

Once connected, we can run SQL queries against our database using the connection.query() method.

Creating Tables and Inserting Rows

We can create a table using an INSERT query.
Example 1: Single Insert

```
const query = `
  INSERT INTO user (id, username, email, password)
  VALUES ('some-uuid-123', 'john_doe', 'john.doe@example.com',
'password123');
`;

connection.query(query, (err, result) => {
  if (err) throw err;
  console.log('Inserted 1 row:', result);
});
```

Using Placeholders (?) to Prevent SQL Injection

SQL injection is a major security vulnerability. **NEVER** build your query strings by concatenating user input directly. Instead, use placeholders (?) in your query and pass the data in an array.
Example 2: Safe Single Insert

```
const userData = [
  'some-other-uuid',
  'jane_doe',
  'jane.doe@example.com',
  'password456'
];
const query = `
  INSERT INTO user (id, username, email, password) VALUES (?, ?, ?,
?);
`;

connection.query(query, userData, (err, result) => {
  if (err) throw err;
  console.log('Inserted 1 row securely:', result);
});
```

The mysql2 library will automatically handle escaping and quoting the values in the userData array, making your application safe.

Handling Errors (err) and Results (res)

The callback function in connection.query(query, callback) receives three arguments:

- err: An error object if the query failed. It will be null if the query was successful.
- results: An object containing information about the query execution. For INSERT, it will have properties like affectedRows and insertId. For SELECT, it will be an array of rows.
- fields: An array of field descriptions (metadata) for SELECT queries.

A common pattern is to check for an error immediately:

```
connection.query(query, (err, result) => {
  if (err) {
    // An error occurred, log it and handle it gracefully
    console.error('An error occurred during the query:', err.message);
    return;
  }
  // The query was successful, you can now use the result
  console.log('Query successful:', result);
});
```

Closing the Connection (connection.end())

It is important to close your connection after you are finished with it to free up resources.

```
connection.end((err) => {
  if (err) {
    console.error('Error closing the connection:', err.stack);
    return;
  }
  console.log('Connection closed successfully.');
});
```

Warning: In a web server, you typically do not close the connection after every query. Instead, you manage a pool of connections (which we'll discuss later) so that connections can be reused for multiple requests. For this simple example, we are closing the connection after a single operation.

6. Generating Random Data with Faker

Creating test data manually can be tedious. Faker is a powerful library that can generate a variety of realistic-looking data.

Installing and Importing Faker

You already installed it in Section 3. Now you just need to import it into your Node.js file.

```
const { faker } = require('@faker-js/faker');
```

Generating Random User Data

Faker has a wide range of functions. For our user table, we'll need a UUID, username, email, and password.

```
// Generate a single random user
const randomUser = {
  id: faker.string.uuid(),
  username: faker.internet.userName(),
  email: faker.internet.email(),
  password: faker.internet.password(),
};
console.log(randomUser);
```

Using Faker to Populate MySQL Tables

We can combine a loop with Faker to generate a large amount of data and then insert it into our database. The most efficient way to do this is with a bulk insert, which we'll cover in the next section's full example.

7. Practical Example with Full Server Code

Here is the complete, working Node.js server code that combines all the concepts we have covered. We will then explain it line-by-line.

```
const express = require('express');
const cors = require('cors');
const { faker } = require('@faker-js/faker');
const mysql = require('mysql2');
const app = express();
const port = 8080;

app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Generate random user data using Faker
const getRandomUser = () => {
  return [
    faker.string.uuid(),
    faker.internet.userName(),
    faker.internet.email(),
    faker.internet.password(),
  ];
};

// Create MySQL connection
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  database: 'delta_db',
  password: 'Roomee@26'
});

// Generate 100 random users
let data = [];
for (let i = 0; i < 100; i++) {
  data.push(getRandomUser());
}

// Insert random users into MySQL using a bulk insert
let q = "INSERT INTO `user` (id, username, email, password) VALUES ?";
connection.query(q, [data], (err, result) => {
  if (err) throw err;
  console.log('Inserted ' + result.affectedRows + ' rows.');
});

connection.end();

// Start the Express server
app.listen(port, () => {
  console.log("Server Listening at port: " + port);
});
```

8. Explaining the Full Server Code

Let's break down the code from the previous section.

Express, CORS, and Faker Setup

```
const express = require('express');
const cors = require('cors');
const { faker } = require('@faker-js/faker');
const mysql = require('mysql2');
const app = express();
const port = 8080;

app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

- const express = require('express');: Imports the Express library.
- const cors = require('cors');: Imports the CORS middleware.
- const { faker } = require('@faker-js/faker');: Imports the faker object from the Faker library, which we will use to generate data.
- const mysql = require('mysql2');: Imports the mysql2 driver.
- const app = express();: Creates an instance of the Express application.
- const port = 8080;: Sets the port for our server.
- app.use(cors());: Tells Express to use the CORS middleware, allowing requests from different origins (useful for front-end development).
- app.use(express.json());: Allows Express to parse incoming requests with a JSON body.
- app.use(express.urlencoded({ extended: true }));: Allows Express to parse incoming requests with URL-encoded payloads.

Faker Usage for Generating Random Data

```
const getRandomUser = () => {
  return [
    faker.string.uuid(),
    faker.internet.userName(),
    faker.internet.email(),
    faker.internet.password(),
  ];
};
```

- const getRandomUser = () => { ... };: This is a function that generates a single random user.
- return [...];: The function returns an array of values. The order of these values **must match** the order of the columns in our INSERT query later: id, username, email, password.
- faker.string.uuid(): Generates a Universally Unique Identifier string, ensuring each ID is unique.
- faker.internet.userName(), faker.internet.email(), faker.internet.password(): These are functions from the Faker library that generate realistic but fake data for each field.

Creating and Configuring the MySQL Connection

```
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  database: 'delta_db',
  password: 'Roomee@26'
});
```

- const connection = mysql.createConnection({ ... });: This line creates the connection object. It is a single connection instance that will be used for all our database interactions in this script.
- host, user, database, password: These are the credentials and location of your database.

You will need to change the password to match the one you set up during MySQL installation.

Generating Data and Bulk Inserting into MySQL

```
let data = [];
for (let i = 0; i < 100; i++) {
  data.push(getRandomUser());
}

let q = "INSERT INTO `user` (id, username, email, password) VALUES ?";
connection.query(q, [data], (err, result) => {
  if (err) throw err;
  console.log('Inserted ' + result.affectedRows + ' rows.');
});
```

- let data = [];: We initialize an empty array to hold our user data.
- for (let i = 0; i < 100; i++) { ... }: A simple loop that runs 100 times.
- data.push(getRandomUser());: In each iteration, we call our getRandomUser function and push the resulting array (the user data) into our data array. By the end of this loop, data will be an array of arrays, where each inner array contains a single user's data.
- let q = "INSERT INTO user (id, username, email, password) VALUES ?";: This is our SQL query string. The single placeholder ? is special here. When an array of arrays is passed as the second argument to connection.query(), the mysql2 driver intelligently recognizes the pattern and performs a **bulk insert**, inserting all 100 rows in a single, highly efficient query.
- connection.query(q, [data], ...);: Executes the query. The data array is wrapped in another array [data], which is the correct syntax for the bulk insert placeholder.

Handling Errors and Results

```
connection.query(q, [data], (err, result) => {
  if (err) throw err;
  console.log('Inserted ' + result.affectedRows + ' rows.');
});
```

- if (err) throw err;: This is a simple error-handling line. If the err object is not null, it means an error occurred, and we throw it to stop the program and show the error message. A real-world application would handle this more gracefully.
- console.log('Inserted ' + result.affectedRows + ' rows.');: If there is no error, the result object contains information about the query. result.affectedRows tells us how many rows were successfully inserted.

Closing the Connection and Starting the Server

```
connection.end();

app.listen(port, () => {
  console.log("Server Listening at port: " + port);
});
```

- connection.end();: Closes the database connection. Since our script only performs a single, one-time insert operation, we close the connection immediately after the query.
- app.listen(port, () => { ... });: This starts the Express server. The server will listen for incoming HTTP requests on the specified port. This line is actually unnecessary for our current script, as it only runs the database query and then exits. This code is shown here to demonstrate how you would start an Express server in a more complete application.

9. Verifying Your Data

Once you have run the Node.js script, you can verify that the data was successfully inserted into your database.

Using MySQL CLI

1. Log in to the MySQL CLI.
   ```
   mysql -u root -p
   ```
2. Select your database.
   ```
   USE delta_db;
   ```
3. Query the user table.
   ```
   SELECT * FROM user;
   ```
   You should see the 100 rows of random user data.

Using MySQL Workbench

1. Open your delta_db connection.
2. In the Navigator panel on the left, expand the Tables section under delta_db and double-click the user table.
3. The results grid will show the contents of your table, allowing you to visually inspect the data.

10. Best Practices

Using Backticks for Reserved Keywords

MySQL has reserved keywords like user, password, and group. It's a good practice to wrap your table and column names in backticks (`) to avoid conflicts.

```
SELECT `user`, `password` FROM `users_table`;
```

In our example, the table name user is a reserved keyword, so we used user.

Managing Database Connections Efficiently

In a real-world web application, you will have many users making requests at the same time. Creating and closing a connection for every single request is inefficient and can overwhelm your database. The solution is to use a **connection pool**.
A connection pool is a collection of pre-established database connections that your application can reuse.
Example of a Connection Pool:
```
const mysql = require('mysql2/promise');
const pool = mysql.createPool({
  host: 'localhost',
  user: 'root',
  database: 'delta_db',
  password: 'Roomee@26',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

// A query function that uses the pool
async function getUsers() {
  const [rows, fields] = await pool.execute('SELECT * FROM user');
  return rows;
}

// In your Express route handler:
app.get('/users', async (req, res) => {
  try {
    const users = await getUsers();
    res.json(users);
  } catch (err) {
    res.status(500).send('Error fetching users');
  }
});
```

This is a much better approach for a scalable web server.

Avoiding SQL Injection

We already covered this in Section 5. Always use ? placeholders and pass data in an array. mysql2 will handle the rest.

11. Advanced Topics

Bulk Inserts

We saw a simple bulk insert in our main example. The mysql2 driver handles the heavy lifting, taking an array of arrays and efficiently creating a single large INSERT statement. This is significantly faster than running 100 individual INSERT queries.

Using Promises or async/await with mysql2

As mentioned earlier, promises make your code much cleaner, especially when you have a sequence of database operations.

Refactoring the Main Example with async/await

```
const express = require('express');
const { faker } = require('@faker-js/faker');
// Note the promise version import
const mysql = require('mysql2/promise');
const app = express();
const port = 8080;

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

const getRandomUser = () => {
  return [
    faker.string.uuid(),
    faker.internet.userName(),
    faker.internet.email(),
    faker.internet.password(),
  ];
};

async function runDatabaseOperations() {
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    database: 'delta_db',
    password: 'Roomee@26'
  });

  let data = [];
  for (let i = 0; i < 100; i++) {
    data.push(getRandomUser());
  }

  try {
    const q = "INSERT INTO `user` (id, username, email, password)
VALUES ?";
    const [result] = await connection.query(q, [data]);
    console.log('Inserted ' + result.affectedRows + ' rows.');

    // Now let's try a SELECT query using async/await
    const [rows, fields] = await connection.execute('SELECT * FROM
`user` LIMIT 5');
```

```
    console.log('--- First 5 Users ---');
    console.table(rows);
    console.log('---------------------');

  } catch (err) {
    console.error('An error occurred:', err);
  } finally {
    // Ensure the connection is closed even if there's an error
    connection.end();
  }
}

// We call the async function to run our logic
runDatabaseOperations();

// The server would be here if this were a web app
app.listen(port, () => {
  console.log("Server Listening at port: " + port);
});
```

This version of the code is more readable and handles errors cleanly. The try...catch...finally block ensures the connection is closed no matter what.

Structuring a Node.js Project with Multiple Database Queries

For a simple example, a single file is fine. However, for a larger application, you should structure your code. A common pattern is to separate your concerns into different files:

```
/project-root
├── node_modules/
├── src/
│   ├── routes/
│   │   └── user-routes.js   // Handles API endpoints for users
│   ├── controllers/
│   │   └── user-controller.js // Contains the logic for routes
│   └── services/
│       └── db-service.js    // Manages the database connection pool
│
├── .env                  // Environment variables for credentials
├── package.json
└── server.js             // The main server file
```

This structure makes your application easier to maintain, scale, and test.

12. Summary and References

We've covered the complete process of setting up and connecting a MySQL database to a Node.js application. You've learned how to:

1. Set up your development environment by installing Node.js and MySQL.
2. Create a project and install the necessary npm packages.
3. Establish a basic database connection.
4. Run SQL queries and safely insert data using placeholders.
5. Generate a large amount of test data with the Faker library.
6. Understand a complete working server example, line by line.
7. Implement best practices like connection pooling and bulk inserts.

Further Reading and Documentation

- **Node.js Official Documentation:** The official Node.js documentation is a fantastic resource for learning about the runtime and its built-in modules.
  - https://nodejs.org/en/docs/
- **MySQL Official Documentation:** For in-depth information about MySQL, SQL syntax, and administration.
  - https://dev.mysql.com/doc/
- **mysql2 Documentation:** The official mysql2 driver documentation is essential for mastering its features, especially the promise-based API and connection pooling.
  - https://github.com/sidorares/node-mysql2
- **Faker.js Documentation:** Learn about all the different types of fake data you can generate.
  - https://fakerjs.dev/

By continuing to explore these resources, you will be well on your way to becoming a proficient full-stack developer. Happy coding!