Creating a Powerful I
MySQL, and Express: A Step-by-Step Guide

Kiran Kumal · Follow

9 min read · Jul 29

Listen Share

In the world of web development, two technologies stand out for their immense popularity and versatility — MySQL and Node.js. MySQL is an open-source relational database management system, while Node.js is a powerful runtime environment built on Chrome's V8 JavaScript engine. Together, they form a dynamic duo that empowers developers to build robust and scalable web applications with ease.

MySQL, with its proven track record and widespread adoption, serves as a trusted and efficient data storage solution. Its relational nature allows developers to define and manage structured data with ease, ensuring data integrity and consistency.

MySQL has been a cornerstone of the web ecosystem for decades, providing the foundation for countless applications and websites worldwide.

Node.js, on the other hand, has taken the JavaScript language, traditionally known for front-end development, to the backend. With its event-driven, non-blocking I/O model, Node.js excels at handling concurrent connections, making it ideal for

When these two powerhouses come together, magic happens. Node.js can seamlessly interact with MySQL, enabling developers to access, retrieve, and manipulate data with ease. The combination of Node.js's asynchronous capabilities and MySQL's performance and scalability make it a preferred choice for modern web applications that require real-time updates, responsiveness, and reliability.

In this article, we will delve into the symbiotic relationship between MySQL and Node.js, exploring how to set up a database connection, perform CRUD (Create, Read, Update, Delete) operations, and handle data efficiently. We will also touch upon best practices, security considerations, and tips to optimize the performance of MySQL and Node.js applications.

Creating a Node.js application

Now, we'll initialize a new node.js project with the command below:

```
mkdir nodejs-mysql-crud && cd nodejs-mysql-crud
npm init
```

Next, we'll install the dependencies that are require for our project.

```
npm install cors dotenv express mysql2
npm install nodemon -D
```

- `cors`: This package enables Cross-Origin Resource Sharing, which allows a server to specify who can access its resources.

- `dotenv` : This package loads environment variables from a file named ".env" into the Node.js process, making it easy to manage configurations.
- `express` : This package is a popular web framework for Node.js, simplifying the process of building web applications and APIs.
- `mysql2` : This package provides a fast and efficient way to interact with MySQL databases in Node.js.
- `nodemon` : This package is a development tool that monitors your Node.js application for changes and automatically restarts the server when code changes are detected. It's very useful during the development process as you don't have to manually restart the server after each code modification.

Folder Structure:

Finally, let's look at our project structure. At the end of this tutorial, our project structure will look like this:

```
nodejs-mysql-crud/
├— node_modules/
├— src/
│  ├— db/
│  │  ├— connect.js
│  ├— errors/
│  │  ├— customError.s
│  ├— middlewares/
│  │  ├— handleError.js
│  │  ├— tryCatchWrapper.js
│  │  ├— notFound.js
│  ├— resources/
│  │  ├— notes/
│  │  │  ├— notes.controllers.js
│  │  │  ├— notes.routes.js
├— .gitignore
├— package.json
├— .env
├— README.md
```

Setting up Expres Server

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

```
import express from "express";
import dotenv from "dotenv";
import { notFound } from "./src/middlewares/notFound.js";
import { handleError } from "./src/middlewares/handleError.js";
import notesRoute from "./src/resources/notes/notes.routes.js";
import cors from "cors";
dotenv.config();

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

const corsOptions = {
  origin: "http://localhost:5173", // for vite application
  optionsSuccessStatus: 200,
};

//middleware
app.use(cors(corsOptions));
app.use(express.json());

// api routes
app.use("/notes", notesRoute);

app.use(notFound);
app.use(handleError);

app.listen(port, () => {
  console.log(`server running on port ${port}`);
});
```

- `app`: This variable creates an instance of the Express application, which allows you to define routes and middleware for handling requests.
- `port`: This variable specifies the port number the server will listen on. If the "PORT" environment variable is provided in the ".env" file, it will use that value; otherwise, it defaults to port 3000.
- `cors`: This middleware is used to enable CORS for requests coming from "http://localhost:5173" (a Vite application in this case). CORS allows the specified origin to access the server's resources.

- `express.json()`: This middleware parses incoming JSON data from the request body and makes it available on the `req.body` object.
- `notFound`: This middleware handles 404 responses, meaning that if no route matches the incoming request, this middleware will be triggered and send a proper 404 response.
- `handleError`: This middleware is responsible for handling errors that occur during the request-response cycle. It helps provide consistent error responses for various situations.

In summary, this code sets up an Express web server with CORS enabled for a specific origin and handles API routes related to managing notes. It also includes middleware for handling JSON data, 404 responses, and errors. The server will listen on the specified port, allowing clients to interact with the defined API routes.

Setup MySQL

Now, we have our Express server set up. Let's go ahead and set up our MySQL Database. I'll be using phpMyAdmin for creating databases and tables. To run the lampp server you can use following command:

```
sudo /opt/lampp/lampp start
```

or else you can create the database and table using cmd also, but you must have mysql installed in your system.

```
CREATE DATABASE notes_app

CREATE TABLE `notes` (
  `id` int(11) NOT NULL,
  `title` varchar(255) NOT NULL,
  `contents` text NOT NULL,
  `created` timestamp NOT NULL DEFAULT current_timestamp(),
  PRIMARY KEY (`id`)
);
```

Next, create *.env* file and add mysql config.

```
MYSQL_HOST="localhost"
MYSQL_USER="root"
MYSQL_PASSWORD=""
MYSQL_DATABASE="notes_app"
PORT = 5000
```

Then, create db folder and inside it create ***connect.js*** and add the following code.

```
import mysql from "mysql2";
import dotenv from "dotenv";
dotenv.config();

export const pool = mysql
  .createPool({
    host: process.env.MYSQL_HOST,
    database: process.env.MYSQL_DATABASE,
    user: process.env.MYSQL_USER,
    password: process.env.MYSQL_PASSWORD,
  })
  .promise();
```

`mysql.createPool()` : This method is used to create a connection pool for MySQL database interactions. A connection pool is a collection of reusable database connections, and it helps improve performance by reusing connections rather than creating new ones for each request.

Creating Controller

Now, let's create the **CRUD** for note entity. create ***notes.controllers.js*** inside notes folder on resouces.

Create:

```
/**
 * @description Create note
 * @route POST /notes
 */
export const createNote = tryCatchWrapper(async function (req, res, next) {
  const { title, contents } = req.body;

  if (!title || !contents)
```

```
    return next(createCustomError("All fields are required", 400));

  let sql = "INSERT INTO notes (title, contents) VALUES (?, ?)";
  await pool.query(sql, [title, contents]);

  return res.status(201).json({ message: "note has been created" });
});
```

This `createNote` function handles the creation of a new note by extracting the necessary data from the request body, validating the input, executing an SQL query to insert the note into the database, and sending a response back to the client. It implements error handling using a custom error creation function and the try-catch wrapper to handle any errors that might occur during the database operation.

The question marks are placeholders used in prepared statements.In prepared statements, question marks act as placeholders for parameterized values that will be supplied later when executing the query. The values for these placeholders are provided separately from the query, which allows for better security against SQL injection attacks and improved performance

Read:

```
/**
 * @returns note object
 */
async function getNote(id) {
  let sql = "SELECT * FROM notes WHERE id = ?";
  const [rows] = await pool.query(sql, [id]);
  return rows[0];
}

/**
 * @description Get All note
 * @route GET /notes
 */
export const getAllNotes = tryCatchWrapper(async function (req, res, next) {
  let sql = "SELECT * from notes";
  const [rows] = await pool.query(sql);
  if (!rows.length) return res.status(204).json({ message: "empty list" });

  return res.status(200).json({ notes: rows });
});
```

```
/**
 * @description Get Single note
 * @route GET /notes/:id
 */
export const getSingleNote = tryCatchWrapper(async function (req, res, next) {
  const { id } = req.params;

  const note = await getNote(id);
  if (!note) return next(createCustomError("note not found", 404));

  return res.status(200).json(note);
});
```

1. `getNote(id)` : Retrieves a single note by its ID from the "notes" table in the database using a prepared SQL statement with a placeholder.
2. `getAllNotes` : Handles the request to retrieve all notes from the "notes" table, sending a response with the retrieved notes as a JSON object or a 204 status code if the list is empty.
3. `getSingleNote` : Handles the request to retrieve a single note by its ID from the "notes" table, calling `getNote(id)` and sending the retrieved note as a JSON object or a 404 status code if the note is not found.

Just like above, let's create update and delete handler.

Update:

```
/**
 * @description Update note
 * @route PATCH /notes/:id
 */
export const updateNote = tryCatchWrapper(async function (req, res, next) {
  const { id } = req.params;
  const { title, contents } = req.body;

  if (!id || !title || !contents)
    return next(createCustomError("All fields are required", 400));

  const note = await getNote(id);
  if (!note) return next(createCustomError("note not found", 404));

  let sql = "UPDATE notes SET title = ? , contents = ? WHERE id = ?";
  await pool.query(sql, [title, contents, id]);
```

```
  return res.status(201).json({ message: "note has been updated" });
});
```

Delete:

```
/**
 * @description Delete note
 * @route DELETE /notes/:id
 */
export const deleteNote = tryCatchWrapper(async function (req, res, next) {
  const { id } = req.params;

  if (!id) return next(createCustomError("Id is required", 400));

  const note = await getNote(id);
  if (!note) return next(createCustomError("note not found", 404));

  let sql = "DELETE FROM notes WHERE id = ?";
  await pool.query(sql, [id]);

  return res.status(200).json({ message: "note has been deleted" });
});
```

Above controllers, implements error handling using a custom error creation function and the try-catch wrapper to handle any errors that might occur during the database operation. Let's create it.

create *customError.js* inside errors folder.

```
export class CustomError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
  }
}

export const createCustomError = (message, statusCode) => {
```

```
    return new CustomError(message, statusCode);
};
```

`CustomError`, which inherits from the built-in `Error` class in JavaScript. It also includes a utility function `createCustomError` to create instances of this custom error class easily.

Also, let's create ***tryCatchWrapper.js*** inside middlewares.

```
import { createCustomError } from "../errors/customErrors.js";

export function tryCatchWrapper(func) {
  return async (req, res, next) => {
    try {
      await func(req, res, next);
    } catch (error) {
      return next(createCustomError(error, 400));
    }
  };
}
```

`tryCatchWrapper` function wraps other asynchronous functions (typically Express middleware) and adds error handling to them. If an error occurs during the execution of the wrapped function, it creates a custom error and passes it to the Express error-handling middleware for further processing and appropriate response to the client. This ensures that the application can handle errors more effectively and provide meaningful error responses to the users.

Routes:

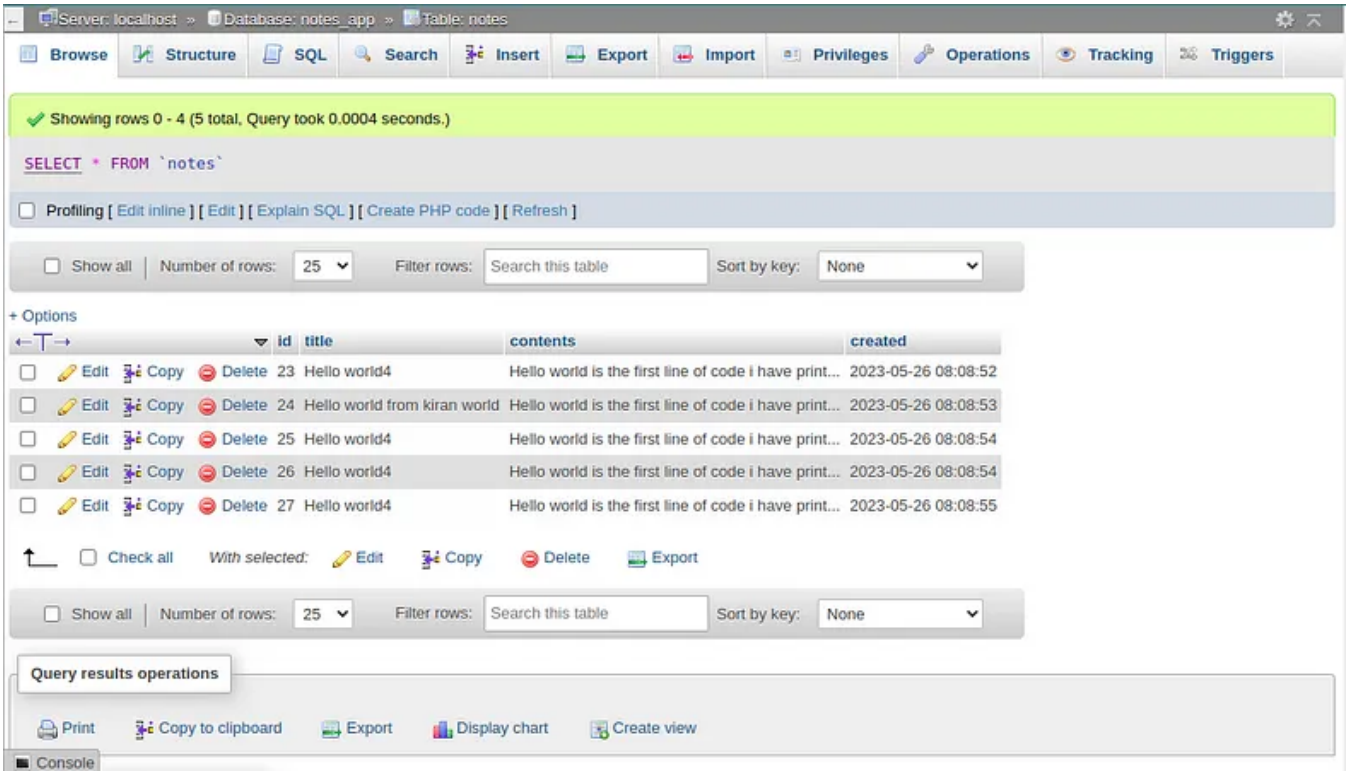
We have to create routes for the controllers. Now, let's create ***notes.routes.js*** next to **notes.controllers.js**

```
import express from "express";
import {
  createNote,
  deleteNote,
  getAllNotes,
  getSingleNote,
  updateNote,
```

```
} from "./notes.controllers.js";

const router = express.Router();

router.route("/").get(getAllNotes).post(createNote);
router.route("/:id").get(getSingleNote).patch(updateNote).delete(deleteNote);

export default router;
```

This express router handles different HTTP routes related to note management. It defines routes for listing all notes (`GET /notes`), creating a new note (`POST /notes`), retrieving a single note by ID (`GET /notes/:id`), updating a note by ID (`PATCH /notes/:id`), and deleting a note by ID (`DELETE /notes/:id`).

That's it , see how simple it is to create REST api using node and express js. Now you will be able to CRUD notes using any other client application or using postman. It will create and store data in mysql as shown below.



Conclusion:

Throughout this step-by-step guide, we learned how to set up an Express server, connect it to a MySQL database, and handle CRUD operations efficiently. We explored best practices, security considerations, and tips to optimize the performance of MySQL and Node.js applications. By combining the power of

Node.js and MySQL, developers gain access to a versatile toolkit that opens up endless possibilities for web development.

I leave here a link to the GitHub repository if you want to check the completed code.

Thank you for reading!

Follow

Written by Kiran Kumal

4 Followers

Programmer and creator | Building on the web and sharing knowledge | https://kiran-mocha.vercel.app/

More from Kiran Kumal

 Kiran Kumal

Best Way to Handle Form Validation: React Hook Form and Zod integration with React Select and React...

Implementing form validation in React from scratch can be very tricky, especially when you need to check different types of inputs, error...

12 min read · Jun 18

16

Kiran Kumal

Next.js 13 and NextAuth.js: A Power Duo for Modern Web Development

Authentication is a cornerstone of modern web applications, and NextAuth.js has emerged as a robust solution to handle this critical aspect...

8 min read · Aug 23

23 1

Kiran Kumal

The Art of Frontend Prototyping: Building Without Backend Dependencies

Being a front-end developer is like being a chef with a unique recipe in mind. However, sometimes you need a key ingredient (backend...

5 min read · Sep 24

23 1

See all from Kiran Kumal

Recommended from Medium

Karthik Reddy Singireddy

Building a RESTful API with Node.js, Express.js, Sequelize(ORM), and Swagger for CRUD Operations

A Step-by-Step Guide to Creating a Powerful RESTful API with Modern Technologies

6 min read · Jul 21

17

Badih Barakat in Stackademic

Next.js API: Connect to MySQL Database

Next.js is a React framework that can be used to build static and dynamic websites and web applications. It is known for its performance...

✨ · 10 min read · Oct 23

48

Lists

Staff Picks

526 stories · 494 saves

Stories to Help You Level-Up at Work

19 stories · 343 saves

Self-Improvement 101

20 stories · 998 saves

Productivity 101

20 stories · 899 saves

Theodore John.S

A Guide to Display PDF documents in React

Create an interactive PDF viewing experience in your React app! Learn different approaches and libraries to display and customize PDF...

5 min read · Jul 10

55

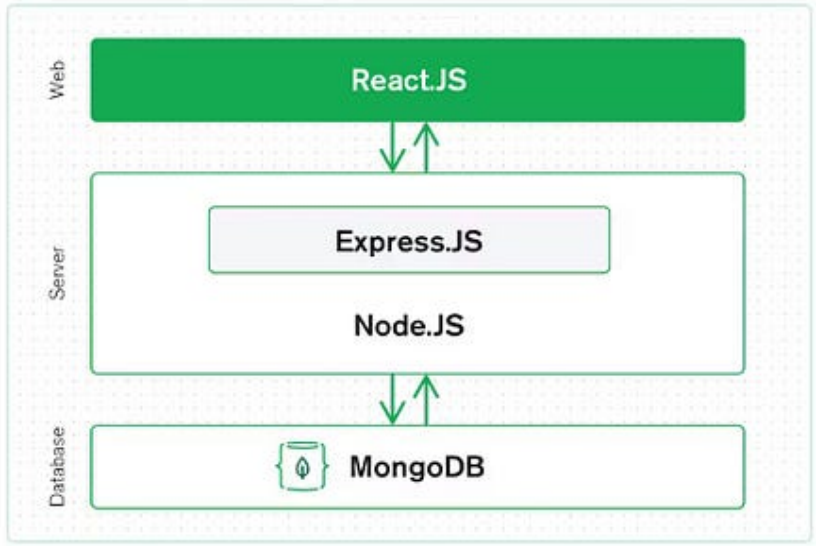
Saurabh Chodvadiya

Building a Payment Gateway with Node.js and Stripe: A Step-by-Step Guide

Introduction:

2 min read · Jul 25

5



Rahul Kaklotar

Step-by-Step Guide: Connecting MongoDB with React.js for Seamless Full Stack Development

I'd be happy to walk you through the process of connecting MongoDB with a React.js application step by step. In this example, we'll assume...

3 min read · Aug 19

23

João Juliasz de Morais

Ace Your React Interview: Understanding Key React Concepts

While browsing LinkedIn, I came across a post by Sai Somanaboina, in which he posed 19 questions typically asked during a React interview...

4 min read · Nov 26

12