An Introduction to Building a RESTful API with Node.js & MySQL

Welcome! This handbook is your friendly guide to understanding and building a simple yet powerful backend API. We'll be walking through a project that connects a Node.js server to a MySQL database, showing you how to manage data in a real-world application.
Don't worry if you're new to backend development. We'll break down every concept, from setting up your project to creating and managing data, step-by-step. By the end of this guide, you will have a solid foundation for building your own API projects.

Overview & Prerequisites

What is This Project?

At its core, this project is a **RESTful API**. Think of an API (Application Programming Interface) as a set of rules that allows different applications to talk to each other. In our case, our API will let a frontend application (like a website or a mobile app) communicate with a database.

- **Node.js & Express:** We're using Node.js to run our server and Express, a popular framework, to make building the server and defining our API routes much easier.
- **MySQL:** This is our database system. It's where all our data will be stored in a structured way, like in tables with rows and columns.
- **RESTful Design:** This is a set of conventions for how our API should be structured. It uses standard HTTP methods like GET, POST, PATCH, and DELETE to perform actions on our data.
- **Faker Library:** This is a neat tool that helps us generate a lot of realistic, fake data (like user names and emails) quickly so we have something to work with.

Tools You'll Need to Install

Before we start, make sure you have the following software installed on your computer.

1. **Node.js and npm:** Node.js is the runtime environment that executes JavaScript on the server. npm (Node Package Manager) comes with Node.js and is used to install all the libraries we need for our project.
2. **A Code Editor:** You'll need an editor to write your code. Popular choices include Visual Studio Code, Sublime Text, or Atom.
3. **MySQL Server:** This is the database server that will host our data. You can download and install it from the official MySQL website. You will also need a way to interact with it, such as MySQL Workbench or a command-line interface.

Project Setup

The first step in any Node.js project is to get our project folder and all the necessary tools set up correctly.

Step 1: Create Your Project Folder

Choose a location on your computer and create a new, empty folder for your project. You can name it something like my-api-project.

Step 2: Initialize the Project with npm

Open your terminal or command prompt, navigate to your new folder, and run the following command:

```
npm init -y
```

This command initializes a new Node.js project. The -y flag answers "yes" to all the default questions, which creates a package.json file for you. This file will keep track of all the details about your project, including the libraries we're about to install.

Installing Dependencies

Now we need to install the libraries that our project relies on. These are often called "dependencies" or "packages." We'll install them all with a single command.

Step 3: Install Core Libraries

Run the following command in your terminal:

```
npm install express cors @faker-js/faker mysql2
```

Let's break down why we need each of these:

- **express:** This is the web framework that makes it easy to handle server requests and build our API endpoints. It saves us from writing a lot of complicated code from scratch.
- **cors:** This stands for "Cross-Origin Resource Sharing." It's a security feature that allows our API to be accessed by web pages from different domains. We need this so a frontend (like a React app) can talk to our backend.
- **@faker-js/faker:** This library is used to generate a large amount of realistic but fake data, like usernames, emails, and passwords, for testing purposes.
- **mysql2:** This is a client library that allows our Node.js application to connect to and interact with our MySQL database. It sends our SQL queries to the database and gets the results back.

After running the command, you will see a new folder called node_modules and a file called package-lock.json in your project directory. These are managed by npm and contain all the code for the libraries you just installed.

The server.js File

Now that our project is set up, it's time to start writing the code. All the backend logic for this project is contained within a single file, which we'll call server.js.
First, create a new file named server.js in your project folder. Then, copy and paste the following code into it:

```
// server.js
const express = require("express");
```

```
const cors = require("cors");
const app = express();
const { faker } = require("@faker-js/faker");
const mysql = require("mysql2");
const PORT = process.env.PORT || 5000;

app.use(cors());
app.use(express.json());

// ... (rest of the code will go here)

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

Code Walkthrough - Imports and Middleware

Let's go through the server.js file section by section to understand what's happening.

Imports

The first five lines are all about importing the libraries we installed earlier. The require() function is how Node.js loads external modules. We're loading express, cors, @faker-js/faker, and mysql2.

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

We also define the PORT on which our server will run. It will use a port specified in the environment (process.env.PORT), or default to 5000 if no environment variable is set.

```
const PORT = process.env.PORT || 5000;
```

Middleware

The next two lines are essential for setting up our server.

```
app.use(cors());
app.use(express.json());
```

These lines set up **middleware**. Middleware functions are like steps in a pipeline. Every time a request comes into our server, it passes through these functions before it reaches our API routes.

- **app.use(cors())**: This line enables CORS for all our routes. It handles the necessary HTTP headers so that a web browser from another domain doesn't block requests to our API.

- **app.use(express.json())**: This is a powerful piece of middleware from Express. It automatically parses any incoming request body that has a Content-Type of application/json (which is very common for APIs). It takes the JSON data and makes it available to us as a JavaScript object on the req.body property.

Code Walkthrough - Faker and Database Connection

Generating Fake User Data

The next section of code uses the Faker library to create a set of fake users.

```
//Get users From Faker
const getRandomUser = () => {
  return [
    faker.string.uuid(),
    faker.internet.username(),
    faker.internet.email(),
    faker.internet.password(),
  ];
};

let users = [];

for (let i = 0; i < 35; i++) {
  users.push(getRandomUser());
}
```

- **getRandomUser()**: This function uses the faker library to generate a universally unique ID (uuid), a username, an email, and a password. It returns these four values in an array.
- **users.push(getRandomUser())**: The for loop runs 35 times, calling getRandomUser() each time and adding the newly generated user to the users array. This gives us a total of 35 fake users to work with.

The MySQL Connection

This section sets up the connection to our database.

```
//DATABASE
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  database: "fuser_db",
  password: "*****",
});
```

The mysql2 library's createConnection() method takes an object with configuration details.

- **host: "localhost"**: This tells the program that the MySQL server is running on the same machine as our Node.js server.
- **user: "root"**: This is the username for logging into the database. root is the default for MySQL installations.

- **database: "fuser_db"**: This specifies the name of the database we want to use. We will create this database in the next section.
- **password: "*****"**: **Important!** This is where you will put the password you set up for your MySQL root user. In a real-world application, you would never hardcode a password like this. Instead, you would use environment variables. We'll touch on this in the "Best Practices" section.

The code also includes a commented-out section for inserting data. This is what you would use once to populate the database with the fake data we generated, but you don't need to run it every time.

Database Preparation

Before our API can do anything, we need to create the database and a table for our users.

Step 4: Create the Database

Open your MySQL client (like MySQL Workbench or the command line) and run the following command to create the database.

```
CREATE DATABASE fuser_db;
```

This creates an empty database named fuser_db.

Step 5: Create the Users Table

Next, we need to create a table inside our new database to store the user information. This table will be called fusers. Make sure you are using the fuser_db database, and then run the following SQL command.

```
USE fuser_db;

CREATE TABLE fusers (
    id VARCHAR(255) PRIMARY KEY,
    username VARCHAR(255),
    email VARCHAR(255),
    password VARCHAR(255)
);
```

Let's break down the table structure:

- **id VARCHAR(255) PRIMARY KEY**: This is our unique identifier for each user. We're using VARCHAR to store the UUID generated by Faker. PRIMARY KEY means each id must be unique and cannot be empty.
- **username VARCHAR(255)**: Stores the user's username.
- **email VARCHAR(255)**: Stores the user's email.
- **password VARCHAR(255)**: Stores the user's password. **Note:** In a real app, you would always store a hashed version of the password, never the plain text.

API Routes & REST Design

Now we get to the core of the API: the routes. Each route is a specific URL that our server

"listens" for. When a request comes in for one of these URLs, our server runs a specific function to handle it.

What are RESTful Endpoints?

A RESTful API uses **HTTP verbs** (like GET, POST, PATCH, DELETE) to define the action being performed on a resource. In our case, the resource is a "user."

- **GET**: Used to **read** or retrieve data. It's safe and shouldn't change anything on the server.
- **POST**: Used to **create** a new resource.
- **PATCH**: Used to **update** an existing resource.
- **DELETE**: Used to **delete** a resource.

Our API will have several of these routes. Each route is defined using app.METHOD(path, handler).

The GET Routes

1. Get the Total User Count (GET /api/)

This route handles a request to the root of our API and returns the total number of users in our database.

```
app.get("/api/", (req, res) => {
  let q = `SELECT COUNT(*) FROM fusers`;
  connection.query(q, (err, result) => {
    try {
      if (err) throw err;
      res.send(result[0]['COUNT(*)']);
    } catch (error) {
      console.log(error);
    }
  });
});
```

- **app.get("/api/", ...)**: This defines a GET route at the /api/ path.
- **SELECT COUNT(*) FROM fusers**: This is an SQL query that counts the total number of rows (users) in the fusers table.
- **res.send(result[0]['COUNT(*)'])**: If the query is successful, we send back the count to the client. The result from the database is an array of objects, so we access the first element and then the 'COUNT(*)' property.

2. Get All Users (GET /api/users)

This route fetches and returns a list of all users from the database.

```
app.get("/api/users", (req, res) => {
  let q = `SELECT * FROM fusers ORDER BY username ASC`;
  connection.query(q, (err, result) => {
    try {
      if (err) throw err;
      res.send(result);
```

```
    } catch (error) {
      console.log(error);
    }
  });
});
```

- **SELECT * FROM fusers ORDER BY username ASC**: This SQL query selects all columns (*) from the fusers table and sorts them alphabetically by username.
- **res.send(result)**: The query result, which is an array of user objects, is sent back to the client.

3. Get a Single User (GET /api/user/:id)

This route retrieves a single user based on their unique ID. The :id part of the path is a placeholder for a variable that changes for each request.

```
app.get('/api/user/:id', (req, res) => {
  const { id } = req.params;
  let q = `SELECT id, username, email FROM fusers WHERE id = '${id}'`;
  connection.query(q, (err, result) => {
    try {
      if (err) throw err;
      res.send(result);
    } catch (error) {
      console.log(error);
    }
  });
});
```

- **req.params**: Express automatically puts the value from the :id placeholder into an object called req.params. We use **destructuring** (const { id } = req.params;) to get the id value.
- **... WHERE id = '${id}'**: This is a powerful part of the SQL query that filters the results to only show the user whose id matches the one in the request URL.

The PATCH Route

Update a User (PATCH /api/user/:id/edit)

This is our first route that modifies data. It allows a user's username to be updated, but only if they provide the correct password.

```
app.patch('/api/user/:id/edit', (req, res) => {
  const { id } = req.params;
  const { username: newUsername, password: newPw } = req.body;

  // 1. Find the user
  let q = `SELECT * FROM fusers WHERE id = '${id}'`;
  connection.query(q, (err, result) => {
    if (err) {
      console.error('Database error:', err);
```

```
      return;
    }

    // 2. Check the password
    if (newPw != result[0]['password']) {
      console.log("Update Failed");
      res.status(404);
    } else {
      // 3. Update the user
      let q2 = `UPDATE fusers SET username = '${newUsername}' WHERE
id='${id}'`;
      connection.query(q2, (err, result) => {
        if (err) {
          console.error(err);
        } else {
          console.log("Updated Successfully");
          res.send("Updated Successfully");
        }
      });
    }
  });
});
```

This route performs a few steps:

1. It gets the id from the URL and the new username and password from the request body (req.body).
2. It first queries the database to get the existing user.
3. It compares the password provided in the request body with the password in the database.
4. If they match, it runs a new SQL query to UPDATE the username for that user.

Important Note: The password check in this code is **not secure**. It compares a plain text password to a plain text password, which is a major security vulnerability. In a real application, we would use a hashing algorithm to securely store and check passwords.

The DELETE Route

Delete a User (DELETE /api/user/:id/delete)

This route handles deleting a user from the database based on their ID.

```
app.delete('/api/user/:id/delete', (req, res) => {
  const { id } = req.params;
  let q = `DELETE FROM fusers WHERE id='${id}'`;
  connection.query(q, (err, result) => {
    if (err) {
      console.log("FAiled Delete");
      console.error(err);
    } else {
```

```
      console.log("Success Delete");
      res.send("DELETED");
    }
  });
});
```

- **DELETE FROM fusers WHERE id='${id}'**: This SQL command finds the row in the fusers table where the id matches the one from the request and removes it.

Error Handling & Best Practices

Catching Errors

You may have noticed the try...catch blocks and if (err) checks throughout the code. This is a basic way of handling errors. When we perform a database query, the connection.query() method gives us an err object.

- if (err) throw err;: This checks if there was a database error. If there was, it throws the error, which is then caught by the catch block.
- console.log(error);: In our catch block, we simply log the error to the console so we can see what went wrong.

Suggestions for Improvement

This project is a great starting point, but in a production environment, there are a few critical improvements to make.

- **SQL Injection Prevention:** Our current code uses **template literals** (...WHERE id = '${id}') to build our SQL queries. This is a big security risk! A malicious user could inject their own SQL code into the URL parameter and run unauthorized queries. The solution is to use **prepared statements** (also called parameterized queries) which separate the SQL code from the data. The mysql2 library fully supports this.
- **Secure Password Handling:** As mentioned before, storing passwords as plain text is unsafe. You should use a hashing library like **bcrypt** to hash and salt passwords before saving them. This makes them unreadable and secures them even if your database is compromised.
- **Environment Variables:** Hardcoding the database password and other sensitive information is bad practice. You should use a .env file and a library like dotenv to store these values, keeping them out of your code and out of source control.

Running the Server

Now let's get your API up and running!

Step 6: Start the Server

Go back to your terminal, make sure you are in the root directory of your project (where server.js is), and run the following command:

```
node server.js
```

You should see the message Listening at PORT: 5000 in your terminal. This means your server is running and ready to receive requests. It will keep running until you press Ctrl + C in the terminal to stop it.

Connecting a React Frontend

A powerful API is only half the story; we need a way for users to interact with it! This section explains how to set up a basic React frontend to consume your API.

Step 7: Create a React App

First, make sure you have create-react-app or a similar tool installed, and then run the following in a **new terminal** (in a separate folder from your backend):

```
npx create-react-app my-frontend-app
cd my-frontend-app
npm install react-router-dom axios
```

Step 8: The Frontend Code

The following files make up a simple React application that uses react-router-dom to handle navigation and axios to make requests to your API.

src/main.jsx

This file is the entry point for your React application. It sets up the BrowserRouter which allows react-router-dom to manage the URLs.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import {BrowserRouter} from 'react-router-dom'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <BrowserRouter>
    <App />
    </BrowserRouter>
  </StrictMode>,
)
```

src/App.jsx

This is the main component of your application. It defines the different routes using the <Routes> and <Route> components.

```
import {Routes, Route} from 'react-router-dom'
import './App.css'
```

```
import Home from './pages/Home';
import Users from './pages/Users';
import ShowUser from './pages/ShowUser';
import EditUser from './pages/EditUser';

function App() {

  return (
    <>
      <Routes>
        <Route path="/" element={<Home/>} />
        <Route path="users" element={<Users/>}/>
        <Route path="showuser" element={<ShowUser/>}/>
        <Route path="edituser" element={<EditUser/>}/>
      </Routes>
    </>
  );
}

export default App
```

src/pages/Home.jsx

This component displays the total number of users by calling the /api/ endpoint. It uses axios and the useState and useEffect hooks to fetch the data when the component loads.

```
import React, { useState } from 'react';
import './home.css';
import { useEffect } from 'react';
import axios from 'axios';
import {useNavigate} from 'react-router-dom'
export default function Home() {
  const [users, setUsers] = useState(null);
  const navigate = useNavigate();
  useEffect(()=>{
    const getUserLength = async ()=>{
      const response = await axios.get("/api/");
     setUsers(response.data);
    }
    getUserLength();
  },[])

  return (
    <div className="home-page">
      <h1 className="title">Welcome!</h1>
      <p className="subtitle">Current number of users:
<strong>{users}</strong></p>
      <button className="get-users-btn" onClick={()=>
```

```
navigate('users')}>
        Show Users
      </button>
    </div>
  );
}
```

src/pages/Users.jsx

This page fetches and displays a list of all users from the /api/users endpoint. Each user is a clickable card that navigates to a ShowUser page.

```
import React from "react";
import "./users.css";
import { useEffect } from "react";
import { useState } from "react";
import axios from "axios";
import { useNavigate } from "react-router-dom";
export default function Users() {
  const [users, setUsers] = useState([]);
  const navigate = useNavigate();

  useEffect(() => {
    const getAllUsers = async () => {
      const response = await axios.get("/api/users");
      setUsers(response.data);
    };

    getAllUsers();
  }, []);

  return (
    <div className="users-page">
      {users.map((user) => (
        <div className="user-card">
          <h1 className="username">Username: {user.username}</h1>
          <p className="email">Email: {user.email}</p>
          <p className="uid">UID: {user.id}</p>
          <button className="showuser-btn"
onClick={()=>navigate('/showuser', {state:{id: user.id}}) }>Show
User</button>
        </div>
      ))}
    </div>
  );
}
```

src/pages/ShowUser.jsx

This component fetches a single user's data and provides buttons to either edit the user's information or delete the user.

```
import React, { useEffect } from 'react';
import './showUser.css';
import { useLocation, useNavigate } from 'react-router-dom';
import { useState } from 'react';
import axios from 'axios';

export default function ShowUser() {
  const navigate = useNavigate();
  const location = useLocation();
  const userId = location.state.id;

  const[user, setUser] = useState([]);

  useEffect(()=>{
    const getUser =  async ()=>{
      const response = await axios.get(`/api/user/${userId}`)
      setUser(response.data[0])
    }
    getUser()
  },[])

  const handleDelete = async () =>{
    const response = await axios.delete(`/api/user/${userId}/delete`)
    navigate('/users', {state: {id: userId}})
  }

  return (
    <div className="showuser-page">
      <div className="user-card">
        <h1 className="user-id">ID: {userId}</h1>
        <h2 className="username">{user.username}</h2>
        <p className="email">{user.email}</p>

        <div className="button-group">
          <button className="edit-btn" onClick={ ()=>
navigate('/edituser', {state:{id: user.id}})}>Edit Username</button>
          <button className="delete-btn" onClick={handleDelete}>Delete
User</button>
        </div>
      </div>
    </div>
  );
}
```

src/pages/EditUser.jsx

This component provides a form to update a user's username. It fetches the user's current data to pre-populate the form and uses the PATCH method to send the update request.

```
import React, { useState } from "react";
import "./editUser.css";
import { useLocation, useNavigate } from "react-router-dom";
import { useEffect } from "react";
import axios from 'axios';
export default function EditUser() {
    const navigate = useNavigate();
  const location = useLocation();
  const userId = location.state.id;

  const [email,setEmail] = useState('');
  const [password, setPassword] = useState('')
  const [username, setUsername] = useState('')

  useEffect(() => {
    const getUser = async () => {
      const response = await axios.get(`/api/user/${userId}`);
      setEmail(response.data[0]['email'])
     setUsername(response.data[0]['username'])

    };
    getUser();
  }, []);

  const handleUpdate = async () =>{
    const response = await axios.patch(`/api/user/${userId}/edit`,
{username, password})
    navigate('/showUser', {state: {id: userId}})
  }

  return (
    <div className="edituser-page">
      <div className="edituser-card">
        <h3 >Email: {email}</h3>


        <div className="form-group">
          <label htmlFor="username">Username:</label>
          <input
            id="username"
            type="text"
           value={username}
```

```
            onChange={(e) => setUsername(e.target.value)}
             placeholder="Enter new username"
           />
         </div>

         <div className="form-group">
           <label htmlFor="password">Password</label>
           <input
             id="password"
             type="password"
             value={password}
             onChange={(e) => setPassword(e.target.value)}
             placeholder="Enter password"
           />
         </div>

         <button className="update-btn"
onClick={handleUpdate}>Update</button>
       </div>
     </div>
   );
}
```

Key Takeaways

You've come a long way! Let's recap the core concepts you've learned.

- **Express Routes:** You now know how to define different URL paths and handle HTTP requests (like GET, PATCH, and DELETE) with Express.
- **Middleware:** You understand how app.use() can apply functions like cors() and express.json() to every incoming request.
- **Database Integration:** You've seen how Node.js and the mysql2 library can be used to connect to a MySQL database and run SQL queries to read, update, and delete data.
- **RESTful Design:** You've applied the principles of REST to build a clean and predictable API.

This project is a powerful starting point for understanding backend development. The next step is to start building!