Learning Docs For Node js (Express + Mysql)

Creating a CRUD (Create, Read, Update, Delete) application using Express.js and MySQL involves setting up a backend server with Express.js to handle HTTP requests and a MySQL database to store and manage data. Below is a basic example of how you can create a simple CRUD application:

Setup Project:

First, create a new directory for your project and initialize a new Node.js project:

```
mkdir express-mysql-crud
cd express-mysql-crud
npm init -y
```

Install required dependencies:

```
npm install express mysql body-parser
```

Create MySQL Database:

Create a MySQL database and a table to store your data. For example:

```
CREATE DATABASE mydatabase;

USE mydatabase;

CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(255) NOT NULL,
   email VARCHAR(255) NOT NULL
);
```

Create Express.js Server:

Create an `app.js` file for your Express.js server:

```
const express = require('express');
const bodyParser = require('body-parser');
const mysql = require('mysql');

const app = express();
const port = 3000;

// Create MySQL connection
const db = mysql.createConnection({
  host: 'localhost',
  user: 'your_mysql_username',
  password: 'your_mysql_password',
  database: 'mydatabase'
});

// Connect to MySQL
db.connect((err) => {
  if (err) {
    throw err;
  }
  console.log('Connected to MySQL database');
});

// Parse JSON bodies
app.use(bodyParser.json());

// Create a new user
app.post('/users', (req, res) => {
  const { name, email } = req.body;
  const sql = 'INSERT INTO users (name, email) VALUES (?, ?)';
  db.query(sql, [name, email], (err, result) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.status(201).send('User added successfully');
  });
});

// Get all users
app.get('/users', (req, res) => {
  const sql = 'SELECT * FROM users';
  db.query(sql, (err, result) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.json(result);
  });
});
```

```
// Update a user
app.put('/users/:id', (req, res) => {
  const { name, email } = req.body;
  const userId = req.params.id;
  const sql = 'UPDATE users SET name = ?, email = ? WHERE id = ?';
  db.query(sql, [name, email, userId], (err, result) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.send('User updated successfully');
  });
});

// Delete a user
app.delete('/users/:id', (req, res) => {
  const userId = req.params.id;
  const sql = 'DELETE FROM users WHERE id = ?';
  db.query(sql, [userId], (err, result) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.send('User deleted successfully');
  });
});

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

Run Your Application:

```
node app.js
```

Now your CRUD application using Express.js and MySQL should be up and running. You can test it using tools like Postman or by creating a simple frontend interface.

Authentication in an Express.js CRUD application

Implementing authentication in an Express.js CRUD application involves adding middleware to protect routes that require authentication. Below is an example of how you can implement basic authentication using Express.js middleware:

Setup Dependencies:

Ensure you have `bcrypt` and `jsonwebtoken` installed:

```
npm install express mysql body-parser bcrypt jsonwebtoken dotenv
```

Node and npm version

```
express-mysql-crud % node -v
v20.12.0
express-mysql-crud % npm -v
10.5.0
```

Package.json file like

```
{
 "name": "express-mysql-crud",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "keywords": [],
 "author": "",
 "license": "ISC",
 "dependencies": {
   "bcrypt": "^5.1.1",
```

```
    "body-parser": "^1.20.2",
    "dotenv": "^16.4.5",
    "express": "^4.19.2",
    "jsonwebtoken": "^9.0.2",
    "mysql": "^2.18.1"
  },
  "devDependencies": {
    "nodemon": "^3.1.0"
  }
}
```

.env file should be like

```
JWT_SECRET="d4e79c638e6b80d5b0fbfc3ab80cd9f08674c0fe91dcd75e6743bb1d1f1461ddb5c4f8db96471cb5251276087ec7faf07760c8d159ed3af7a0bb688df8b39701"
JWT_REFRESH_SECRET="49e27134ecc747120306b019768472fe0d45c5c4076b0d89d37f6a6d059d2bb5a8b1e24042d34039d3986ce45c04b1ca5e83adf017b2a690545d7c1c0b3288f3"
```

IN .env file to generate JWT secret we may use below steps

```
JWT_SECRET
```

express-mysql-crud % node
Welcome to Node.js v20.12.0.
Type ".help" for more information.
> **require('crypto').randomBytes(64).toString('hex')**
'd4e79c638e6b80d5b0fbfc3ab80cd9f08674c0fe91dcd75e6743bb1d1f1461ddb5c4f8db96471cb5251276087ec7faf07760c8d159ed3af7a0bb688df8b39701'
> require('crypto').randomBytes(64).toString('hex')
'49e27134ecc747120306b019768472fe0d45c5c4076b0d89d37f6a6d059d2bb5a8b1e24042d34039d3986ce45c04b1ca5e83adf017b2a690545d7c1c0b3288f3'

app.js file should be like below

```
const express = require('express');
const bodyParser = require('body-parser');
const mysql = require('mysql');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
```

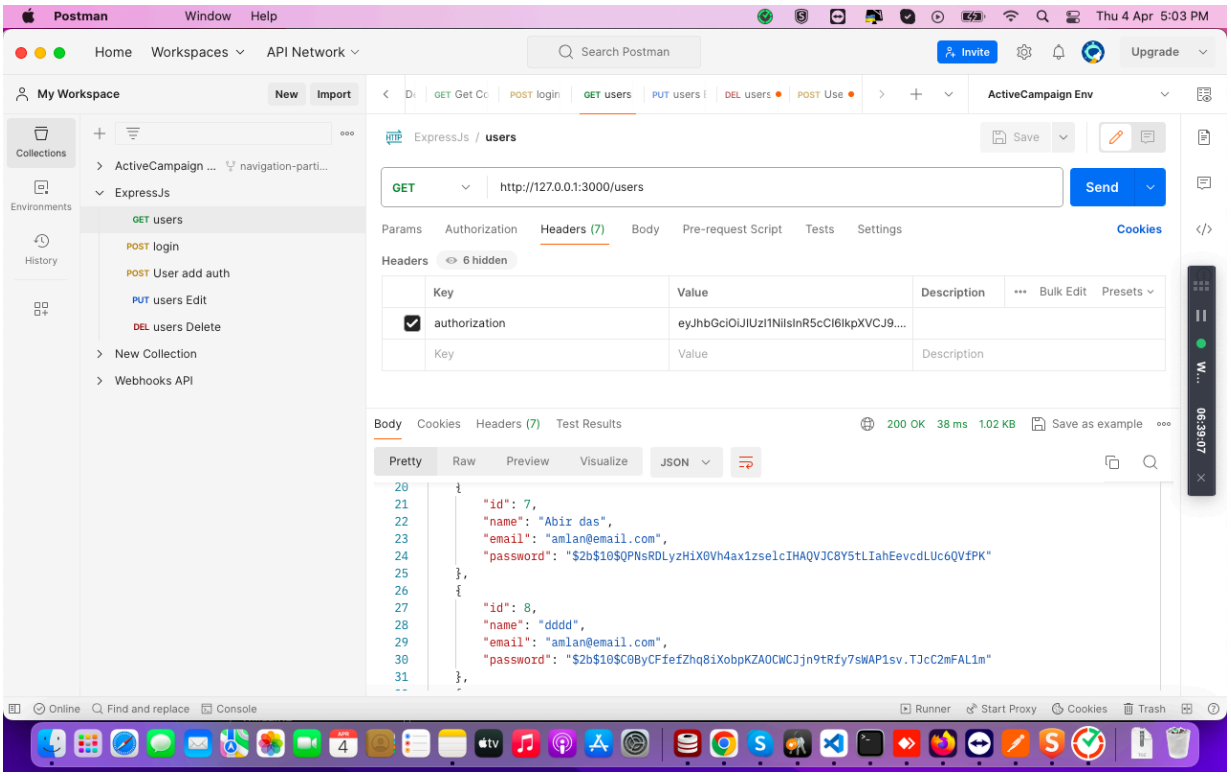
```
require('dotenv').config();

const app = express();
const port = 3000;

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

// Connect to MySQL
db.connect((err) => {
 if (err) {
   throw err;
 }
 console.log('Connected to MySQL database');
});

// Parse JSON bodies
app.use(bodyParser.json());

// Authentication middleware
const authenticateJWT = (req, res, next) => {
 const token = req.headers.authorization;
 if (!token) {
     return res.status(401).json({ error: 'Unauthorized: Missing token' });
 }
 jwt.verify(token, process.env.JWT_SECRET, (err, decoded) => {
     if (err) {
         console.error('JWT verification error:', err);
         return res.status(403).json({ error: 'Forbidden: Invalid token' });
     }
     req.email = decoded.email;
     next();
 });
};

// Login route
```

```
app.post('/login', (req, res) => {
   const { email, password } = req.body;
   const sql = 'SELECT * FROM users WHERE email = ?';
   db.query(sql, [email], async (err, result) => {
     if (err) {
       res.status(500).json({ error: err.message });
       return;
     }
     if (result.length === 0) {
       res.status(401).json({ error: 'Invalid username or password' });
       return;
     }
     const user = result[0];
     try {
       if (await bcrypt.compare(password, user.password)) {
         const accessToken = jwt.sign({ email: user.email },
process.env.JWT_SECRET, { expiresIn: '1h' });
         res.json({ accessToken });
       } else {
         res.status(401).json({ error: 'Invalid username or password' });
       }
     } catch {
       res.status(500).send();
     }
   });
 });

// Create a new user
app.post('/users', authenticateJWT, (req, res) => {
   // Extract user data from request body
   const { name, email, password } = req.body;
    // Validate user data
   if (!name || !email || !password) {
     return res.status(400).json({ error: "Name, email, and password are required"
});
   }
    // Hash the password
   bcrypt.hash(password, 10, (err, hashedPassword) => {
     if (err) {
       return res.status(500).json({ error: "Failed to hash password" });
     }
```

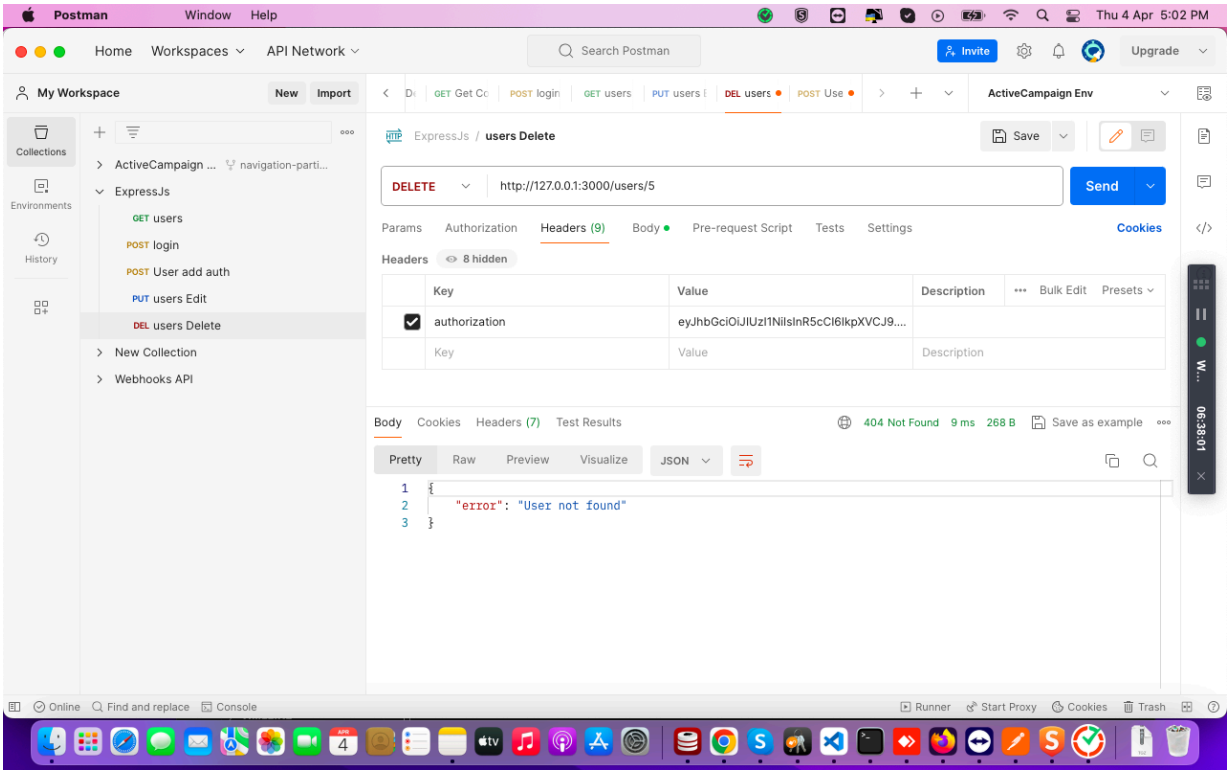
```
      // Insert user into the database
      const sql = 'INSERT INTO users (name, email, password) VALUES (?, ?, ?)';
      db.query(sql, [name, email, hashedPassword], (err, result) => {
        if (err) {
          return res.status(500).json({ error: err.message });
        }
        res.status(201).json({ message: 'User created successfully' });
      });
    });
  });

// Get all users
app.get('/users', authenticateJWT, (req, res) => {
  const sql = 'SELECT * FROM users';
  db.query(sql, (err, result) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.json(result);
  });
});

// Update a user
app.put('/users/:id', authenticateJWT, (req, res) => {
  const { name, email } = req.body;
  const userId = req.params.id;
  const sql = 'UPDATE users SET name = ?, email = ? WHERE id = ?';
  db.query(sql, [name, email, userId], (err, result) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.send('User updated successfully');
  });
});

// Delete a user
app.delete('/users/:id', authenticateJWT, (req, res) => {
  const userId = req.params.id;
  const sql = 'DELETE FROM users WHERE id = ?';
```

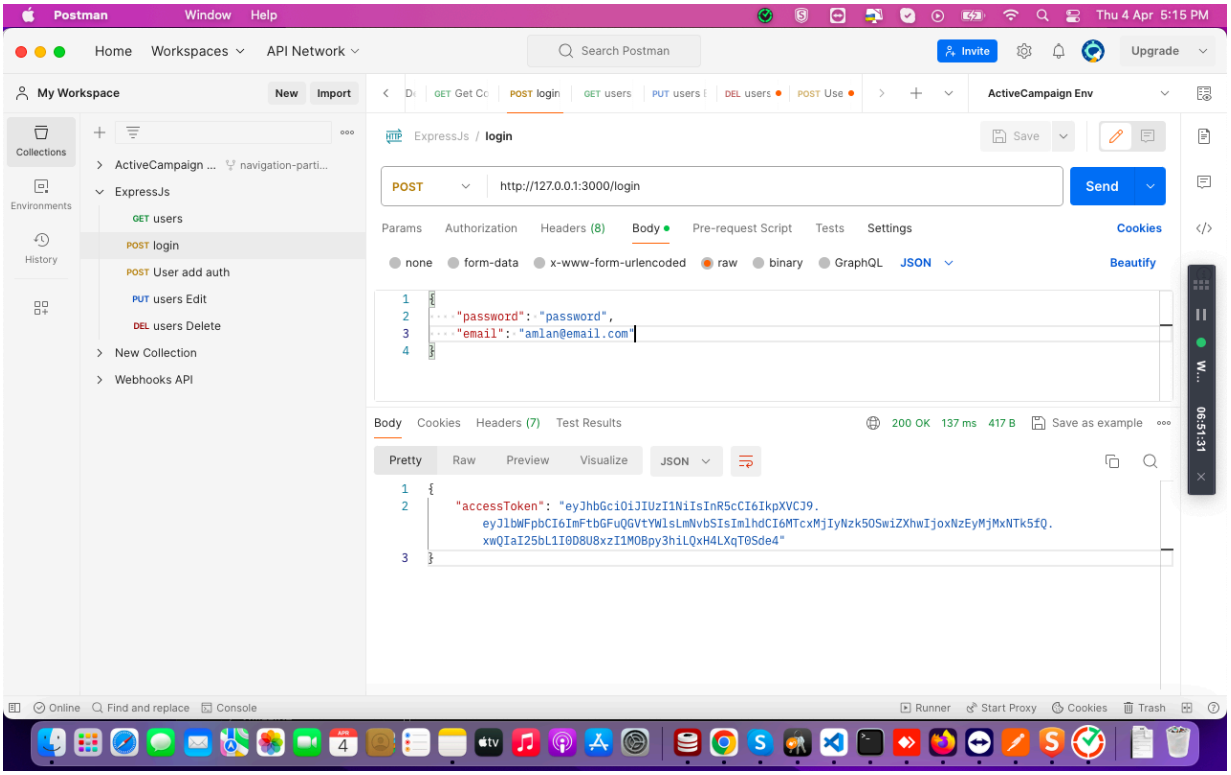
Learning Docs For Node js (Express + Mysql)

```
 db.query(sql, [userId], (err, result) => {
   if (err) {
     res.status(500).json({ error: err.message });
     return;
   }
   if (result.affectedRows === 0) {
     // No rows were affected, indicating that the user ID was not found
     res.status(404).json({ error: 'User not found' });
     return;
   }
   res.send('User deleted successfully');
 });
});


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



Learning Docs For Node js (Express + Mysql)

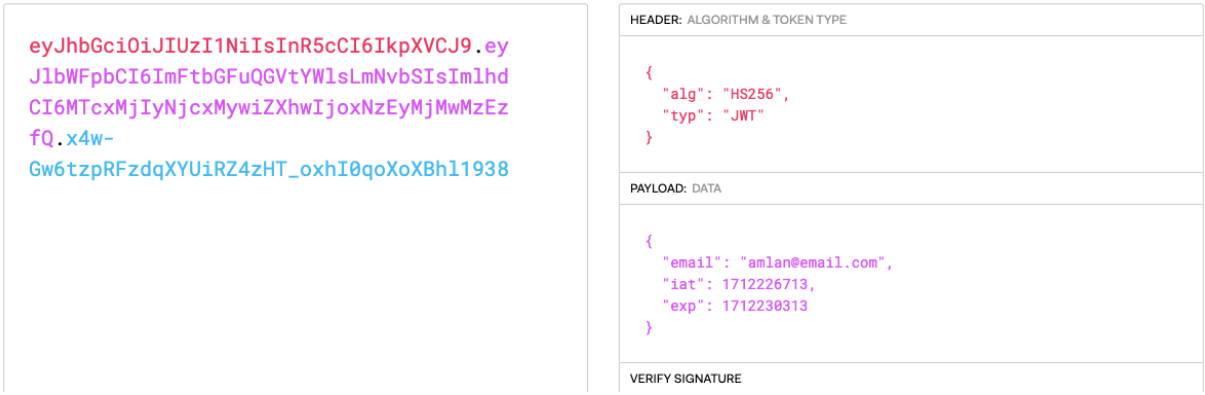




https://jwt.io/

JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.

JWT.IO allows you to decode, verify and generate JWT.



Details Explanation:

Source: https://www.scaler.com/topics/expressjs-tutorial/MYSQL_with_express_js/

Setting up Express.js for our RestAPI

To set up Express.js for your REST API development, the below steps should be followed:

- Install Node.js and npm Download and install the npm and node.js packages.
- Create a New Project Directory: Create a new directory for the project and navigate to it using the command line or terminal.

- Initialise a Node.js Project: Initialise a new Node.js project by running the following command in the project directory:

```
npm init -y
```

The above command will create a package.json file that will keep track of your project's dependencies and configurations.

- Install Express.js: Now, Install Express.js as a dependency by running the following command:

```
npm install express
```

- Create an Express.js Server File: Create the new JavaScript file (e.g., server.js) in your project directory. This file will serve as the entry point for the Express.js application.

Now inside server.js, require the Express module and create an instance for the Express application then add basic server configuration and routing as needed.

Below is an example to do so:

```
const express = require('express');
const app = express();
const port = 3000;

// Define the API routes
app.get('/', (req, res) => {
  res.send('Hello World!');
});

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

- Start the Express.js Server: In the command line or terminal, run the following command to start the Express.js server:

```
node server.js
```

Now in the console of the IDE the message Server is running on port 3000 will be displayed which ensures that the server is running.

- Test the REST API: Open up the web browser or use tools like Postman to test the REST API then access the defined routes and verify that the expected responses are received.

RestAPI Project Structure

When we create a REST API project using MySQL with Express js, it's important to establish a well-organised project structure that promotes scalability, maintainability and code reusability. While there isn't a strict standard structure, here's a commonly used project structure for building REST APIs with Express.js:

```
- project-root/
  |- src/
     |- controllers/
        |- userController.js
        |- ...
     |- models/
        |- userModel.js
        |- ...
     |- routes/
        |- userRoutes.js
        |- ...
     |- middlewares/
        |- authMiddleware.js
        |- ...
     |- services/
        |- userService.js
        |- ...
     |- utils/
        |- validation.js
        |- ...
     |- app.js
  |- package.json
  |- .env
```

Let's take a closer look at each directory and file within the project structure:

1. src/: This directory contains the main source code of the application.
2. controllers/: This directory holds the controller functions that handle the logic for each route or endpoint. Each controller file typically corresponds to a specific resource or entity in the API.
3. models/: This directory contains the database models or schemas that define the structure and behaviour of the data. Each model file typically represents a specific entity or collection in the database.
4. routes/: This directory houses the route definitions for the API endpoints. Each route file defines the API routes, associates them with the appropriate controller functions, and specifies the HTTP methods and URL paths.
5. middlewares/: This directory holds custom middleware functions that can be applied to specific routes or globally to handle request processing tasks such as authentication, validation, logging, error handling, etc.
6. services/: This directory contains additional services or modules that encapsulate reusable business logic, data manipulation, or integrations with external systems. Services can be used by multiple controllers or middleware.
7. utils/: This directory includes utility functions or modules that provide commonly used functionalities such as data validation, error handling, date formatting, encryption, etc.
8. app.js: This file serves as the entry point of your application. It initialises and configures the Express.js application, sets up middleware, connects to the database, and defines any global application-level settings.
9. package.json: This file keeps track of the project's dependencies, scripts, and other metadata. It is generated when running npm init and gets updated when we install new packages.
10. .env: This file is used for storing environment-specific configuration variables such as database connection details, API keys, or other sensitive information. It is not committed to version control and should be kept secure.

Let’s build a database which can be used for the get route, post route, put route and delete route. Below are the steps by which we will create the sample database:

```
Create a MYSQL Database:
CREATE DATABASE db1;

Create a Users table:
USE db1;
CREATE TABLE users
(
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255)
```

```
);
```

This creates a table named users with auto-incrementing primary key column “id” and a name column of type VARCHAR to store user names.

GET Route

A GET route in Express.js is used to handle HTTP GET requests sent to a specific URL or endpoint. It is commonly used to retrieve or fetch data from the server.

Here's an example of how to define a GET route in Express.js:

```
const express = require('express');
const mysql = require('mysql');
const app = express();

// Create a MySQL connection
const connection = mysql.createConnection ({
    host: 'localhost',
    user: 'coder',
    password: 'coder,
    database: 'db1' // Use the name of the database you created
});

// Connect to the MySQL database
connection.connect((err) => {
    if (err) {
        console.error('Error connecting to the database: ' +
err.stack);
        return;
  }
    console.log('Connected to the database as ID ' +
connection.threadId);
});

// Define a GET route
app.get('/api/users', (req, res) => {

    // Fetch users from the database
  connection.query('SELECT * FROM users', (error, results) => {
      if (error) {
```

```
        console.error('Error fetching users from the
database: ' + error.stack);
        return res.status(500).json({ error: 'Failed to fetch
users' });
    }

    // Send the fetched data as a response
     res.json(results);
  });
});

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

In this example, when we access the /api/users endpoint in your browser or send a GET request to it, the server will fetch the users from the MySQL users table and send them as a JSON response.

POST Route

A POST route in Express.js is used to handle HTTP POST requests sent to a specific URL or endpoint. It is commonly used to submit or create new data on the server. Here's an example of how to define a POST route in Express.js:

```
const express = require('express');
const mysql = require('mysql');
const app = express();

// Create a MySQL connection
const connection = mysql.createConnection ({
    host: 'localhost',
    user: coder,
    password: 'coder',
    database: db1// Use the name of the database you created
});

// Connect to the MySQL database
connection.connect((err) => {
```

```
    if (err) {
        console.error('Error connecting to the database: ' +
err.stack);
        return;
    }
    console.log('Connected to the database as ID ' +
connection.threadId);
});

// Middleware to parse JSON in the request body
app.use(express.json());

// Define a POST route to add a new user
app.post('/api/users', (req, res) => {
    const { name } = req.body;

    if (!name) {
        return res.status(400).json({ error: 'Name is required'
});
    }

  // Insert the new user into the database
  connection.query('INSERT INTO users (name) VALUES (?)',
[name], (error, results) => {
      if (error) {
          console.error('Error inserting user into the
database: ' + error.stack);
          return res.status(500).json({ error: 'Failed to
insert user' });
      }

    // Send a success response
    res.json({ message: 'User inserted successfully' });
  });
});

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

In this example, we have added a POST route at /api/users. When we send a POST request to this endpoint with a JSON payload containing the name field, it will insert the new user with the provided name into the MySQL database.

PUT Route

A PUT route in Express.js is used to handle HTTP PUT requests sent to a specific URL or endpoint. It is commonly used to update existing data on the server. Here's an example of how to define a PUT route in Express.js:

```
const express = require('express');
const mysql = require('mysql');
const app = express();

// Create a MySQL connection
const connection = mysql.createConnection ({
    host: 'localhost',
    user: coder',
    password: 'coder',
    database: db1// Use the name of the database you created
});

// Connect to the MySQL database
connection.connect((err) => {
    if (err) {
        console.error('Error connecting to the database: ' +
err.stack);
        return;
    }
    console.log('Connected to the database as ID ' +
connection.threadId);
});

// Middleware to parse JSON in the request body
app.use(express.json());

// Define a PUT route to update a user
app.put('/api/users/:id', (req, res) => {
    const userId = req.params.id;
    const { name } = req.body;
```

```
    if (!name) {
        return res.status(400).json({ error: 'Name is required' });
    }

  // Update the user in the database
  connection.query('UPDATE users SET name = ? WHERE id = ?', [name, userId], (error, results) => {
      if (error) {
          console.error('Error updating user in the database: ' + error.stack);
          return res.status(500).json({ error: 'Failed to update user' });
      }

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

    // Send a success response
    res.json({ message: 'User updated successfully' });
  });
});

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

In the example, we have added a PUT route at /api/users/:id. When we send a PUT request to this endpoint with a JSON payload containing the name field, it will update the user with the provided id in a MySQL database.

DELETE Route

A DELETE route in Express.js is used to handle HTTP DELETE requests sent to a specific URL or endpoint. It is commonly used to delete existing data on the server. Here's an example of how to define a DELETE route in Express.js:

```
const express = require('express');
const mysql = require('mysql');
const app = express();

// Create a MySQL connection
const connection = mysql.createConnection ({
    host: 'localhost',
    user: `coder',
    password: 'coder',
    database: 'db1' // Use the name of the database you created
});

// Connect to the MySQL database
connection.connect((err) => {
    if (err) {
        console.error('Error connecting to the database: ' +
err.stack);
        return;
    }

    console.log('Connected to the database as ID ' +
connection.threadId);
});

// Define a DELETE route to delete a user
app.delete('/api/users/:id', (req, res) => {
    const userId = req.params.id;

  // Delete the user from the database
  connection.query('DELETE FROM users WHERE id = ?', [userId],
(error, results) => {
      if (error) {
          console.error('Error deleting user from the database:
' + error.stack);
          return res.status(500).json({ error: 'Failed to
delete user' });
      }

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

```
        }

    // Send a success response
    res.json({ message: 'User deleted successfully' });
  });
});

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

In the example, we have added a DELETE route at /api/users/:id. When you send a DELETE request to this endpoint with the specific id parameter in a URL, it will delete the corresponding user from the MySQL database.

Testing our APIs

We can test all our API routes. To do so we will create a new route, we can do it with either Postman or any other HTTP client:

```
curl -i -X POST -H 'Accept: app/json' \
    -H 'Content-type: app/json' http://localhost:3000/routes\
    --data '{"name":"Coder", "released_year": 2023,
"githut_rank": 23, "pypl_rank": 10, "tiobe_rank": 15}'
```

The code will generate the output as:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: app/json; charset=utf-8
Content-Length: 55
ETag: W/"37-3mETlnRrtfrms6wlAjdgAXKq9GE"
Date: Mon, 19 MAY 2023 11:20:07 GMT
Connection: keep-alive

{"message":"Routes"}
```

Similarly, we can perform other routing operations like delete, post, put.

Further Considerations

When testing the APIs, several additional considerations can help improve the effectiveness and reliability of your tests.

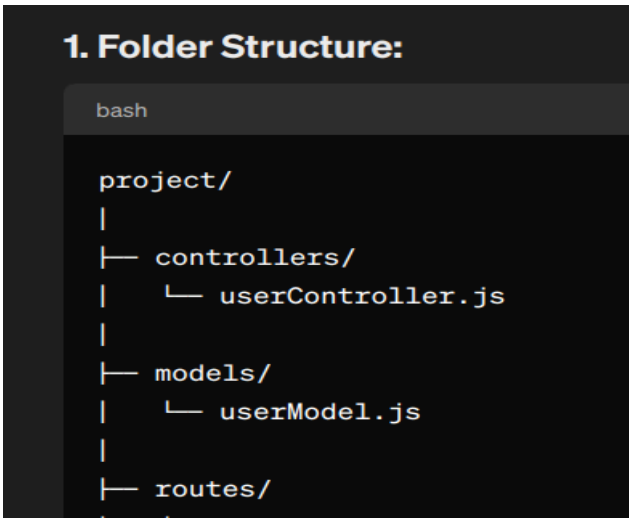
Here are a few considerations to keep in mind:

- Test Coverage: Aim for comprehensive test coverage by testing various scenarios, edge cases, and input validations. Consider different HTTP methods (GET, POST, PUT, DELETE), authentication and authorization scenarios, error handling, and boundary values for input fields.
- Environment Configuration: Use different environments for testing, such as development, staging, and production-like environments, to mimic real-world scenarios. This helps uncover environment-specific issues and ensures your APIs work correctly in different settings.
- Data Seeding: Use data seeding techniques to set up consistent test data in your testing environment. This ensures predictable results and avoids dependency on external data sources. Tools like Faker.js or factory functions can help generate realistic test data.
- Mocking External Dependencies: When testing API endpoints that rely on external services or dependencies, consider using mocking techniques to isolate and control the behaviour of those dependencies. This helps create repeatable and reliable tests without relying on the availability or state of external systems.
- Asynchronous Testing: If the API involves asynchronous operations, such as database queries or external API calls, make sure your tests handle those operations correctly. Use techniques like promises, async/await, or testing frameworks built-in mechanisms (e.g., Mocha's done() function or Jest's async/await support) to handle asynchronous code and ensure proper testing.

NODE JS : https://nodejs.org/docs/latest/api/
Additional websites :W3schools
Express : Express.js
Mongoose : Mongoose

TOPICS IN NODE JS :

1. INTRODUCTION AND SETUP
2. INSTALLATION



Learning Docs For Node js (Express + Mysql)

3. Basic Concepts :W3schools
    1. Node.js Modules
    2. Node.js HTTP Modules
    3. Node.js File System
    4. Node.js URL Module
    5. Node.js NPM
    6. Node.js Events
    7. Node.js Upload Files
    8. Node.js Email

TOPICS IN EXPRESS JS : Express.js

1. INSTALLATION AND SETUP
2. PROJECT STRUCTURE
3. Basic Hello World Example
4. Routing (GET, POST, PUT, DELETE, etc)
5. MiddleWare
6. Error Handling
7. Packages :
    1. Nodemon ,
    2. Cors,
    3. Dotenv,
    4. Body parser,
    5. Express Validator,
    6. Multer.

TOPICS IN MONGOOSE : Mongoose

1. INSTALLATION AND SETUP
2. PROJECT STRUCTURE
3. Basic Topics
    1. Schemas,
    2. Models,
    3. Do a CRUD Operation (express + mongoose) ,
    4. Querying Data ,
4. MiddleWare
5. Packages :
    1. Bcrypt,
    2. JsonWebToken,
6. Indexes
7. Aggregation Framework

Authentication Middleware:

1. Passport.js https://www.passportjs.org/packages/passport-local/

INSTALLATION GUIDE :

TO INSTALL REACT JS:

Before installing React, ensure you have Node.js and npm installed on your system.

Once Node.js and npm are installed, you can create a new React project using the create-react-app tool. Open your terminal and run the following command:

Learning Docs For Node js (Express + Mysql)

npx create-react-app my-react-app

Once the project is created, navigate into the project directory:
cd my-react-app

To Start the development server:
npm start

TO INSTALL NODEJS on linux & Mac OS :
https://nodejs.org/en/download/package-manager

TO INSTALL Express JS :
If you're starting a new project, create a new directory for your project and navigate into it in your terminal. You can create a new Node.js project by running:
npm init -y

This command will create a package.json file in your project directory with default settings.
Then Install Express.js Run the following command in your terminal:
npm install express

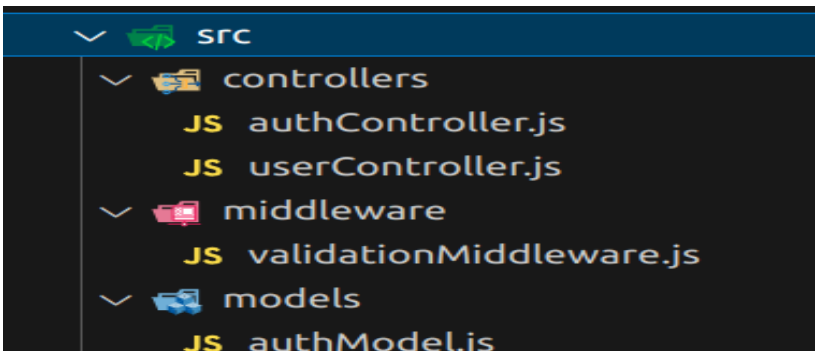
TO INSTALL MONGODB Compass (GUI):
Click the link and choose your platform, either Ubuntu or macOS.
https://www.mongodb.com/try/download/compass

Once you have downloaded MongoDB and run the application, click on "**Connect**". If it **doesn't connect**, try running the following command in the terminal:
sudo systemctl start mongod.service

TO INSTALL POSTMAN Application:
https://www.postman.com/downloads/

CRUD OPERATION GUIDE:

In Backend (Express + Mongoose):



Learning Docs For Node js (Express + Mysql)

Folder Structure

Step 1: Create Schema - Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.
Step 2: Create Model - A Model is a class that's your primary tool for interacting with MongoDB. An instance of a Model is called a Document.
Simply Model is to create a Document (table) in a Database.

This is a `userModel` File

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
  email: String,
  address: String,
  image:String,
});

const UserModel = mongoose.model("crud-operation", userSchema);
module.exports = {UserModel};
```

Step 3: Define Routes For (GET ,CREATE ,UPDATE & DELETE)
Create a Index File For Routes To Access All the routes

```
function routes(app){
   app.use("/api/user" , require("./userRoutes"))
   app.use("/api/" , require("./authRoutes"))
}
module.exports = routes
```

This is a `userRoutes` File

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

router.get("/crud", userController.getAllUsers);
```

So , the Get Route will be http://localhost:5000/api/user/crud , For Like Try to Create

For POST , PUT , DELETE

Step 4: Create Controller For (GET ,CREATE ,UPDATE & DELETE)

```
const { UserModel } = require('../models/userModel');
exports.getAllUsers = async (req,res) => {
try {
// Step 4: Query the database for all user details
const users = await UserModel.find({});
const totalData = await UserModel.countDocuments();
// Step 5: Send the retrieved user details as a response
res.status(200).json({data: users, totalData:totalData, message: "Data
GET Success"});
   } catch (error) {
// Handle errors
console.error('Error fetching users:', error);
res.status(500).json({ error: 'Internal server error' });
  }
}
```

Step 5 : Install the Postman App,To check Whether Route is Working or Not .