REST API CRUD with NodeJS,Express and MySQL

1. #.env

- A .env file is a file that contains environment variables. Environment variables are a way to store data that can be accessed by different parts of a program or application. They are often used to store sensitive data, such as passwords or API keys.
- The .env file is a plain text file, and it is typically placed in the root directory of a project. The file is not checked into version control, so it is not visible to other users.

2. #index.js

- The code you provided is a simple Express application that defines a route for /api/v1/posts and uses the postsRouter module to handle requests to that route.

- The first line of code, const express = require("express"), imports the Express module. The next line, const app = express(), creates a new Express application.

- The third line, require('dotenv').config(), loads the environment variables from the .env file.

- The fourth line, const postsRouter = require('./routes/posts.router'), imports the postsRouter module.

- The fifth line, app.use(express.urlencoded({extended:false})), configures Express to parse URL-encoded form data.

- The sixth line, app.use(express.json()), configures Express to parse JSON data.

- The seventh line, app.use("/api/v1/posts",postsRouter), maps the /api/v1/posts route to the postsRouter module.

- The eighth line, const PORT = process.env.PORT || 3000, defines the port that the application will listen on. The process.env.PORT environment variable is used to get the port from the environment. If the environment variable is not set, the default port of 3000 is used.

- The ninth line, app.listen(PORT, () => { console.log("Server running.....") }, starts the Express application and listens on the specified port.

- This code is a simple example of how to create a basic Express application.

3. #controller\postscontroller.js

- The code snippet you provided is the beginning of a controller function for a posts API. The pool variable is a connection pool to the database, and the postsController object will contain functions for performing CRUD operations on posts.

- The code you provided is a Node.js module that exports a controller for managing posts in a database. The controller has the following methods:

    - getAll(): Gets all posts from the database.
    - getById(): Gets a post by its ID.
    - create(): Creates a new post in the database.

    - update(): Updates an existing post in the database.
    - delete(): Deletes a post from the database.
- The controller uses the pool variable to connect to the database. The pool variable is a connection pool that manages a pool of database connections. This allows the controller to handle multiple requests without having to open and close a new database connection for each request.

- The controller uses the query() method to execute SQL queries against the database. The query() method returns an array of rows, where each row is an object representing a post in the database.

- The controller also uses the res.json() method to send a JSON response to the client. The JSON response contains the data returned by the query() method.

- Overall, the code you provided is a well-written and efficient way to manage posts in a database. The controller is easy to use and can be easily extended to add new features.

- Here are some additional comments on the code:

    - The getAll() method uses the SELECT * FROM posts statement to get all posts from the database. This is a simple and efficient way to get all posts.
    - The getById() method uses the SELECT * FROM posts WHERE id = ? statement to get a post by its ID. This is a safe and efficient way to get a post by its ID.
    - The create() method uses the insert into posts (title, content) values (?, ?) statement to create a new post in the database. This is a safe and efficient way to create a new post.
    - The update() method uses the UPDATE posts SET title = ?, content = ? WHERE id = ? statement to update an existing post in the database. This is a safe and efficient way to update an existing post.

- The delete() method uses the DELETE FROM posts WHERE id = ? statement to delete a post from the database. This is a safe and efficient way to delete a post.

4. #database\index.js

- The code you provided creates a MySQL connection pool and exports it as a promise.

- The first line of code, const mysql = require('mysql2'), imports the mysql2 module. The mysql2 module is a popular MySQL client for Node.js.

- The second line of code, const pool = mysql.createPool({...}), creates a MySQL connection pool. The createPool method takes an object as its argument, which specifies the configuration options for the pool. In this case, the configuration options specify the host, user, password, and database name for the pool.

- The third line of code, module.exports = pool.promise(), exports the pool as a promise. This means that the pool can be used as a promise in other parts of the application.

- Here is an explanation of the configuration options for the createPool method:

    - host: The hostname of the MySQL server.
    - user: The username for the MySQL database.
    - password: The password for the MySQL database.
    - database: The name of the MySQL database.
    - port: The port number of the MySQL server.
    - connectionLimit: The maximum number of connections that can be in the pool at the same time.

- waitForConnections: Whether or not to wait for connections to be available before returning a promise.

- The createPool method returns a promise that resolves to a connection pool. The connection pool can be used to execute queries against the MySQL database.

5. #routes\posts.router.js

- The code you provided is a router for handling requests to the /posts endpoint. The router uses the postsController module to handle the different types of requests.

- The first line of code, const express = require("express"), imports the Express module. The next line, const router = express.Router(), creates a new Express router.

- The third line, const postsController = require("../controller/postscontroller"), imports the postsController module. The postsController module contains the functions that are used to handle the different types of requests to the /posts endpoint.

- The next four lines of code, router.get("/", postsController.getAll), router.get("/:id", postsController.getById), router.post("/",postsController.create), router.put("/:id", postsController.update), router.delete("/:id" ,postsController.delete), define the routes for the router. The routes map the different HTTP methods to the functions in the postsController module.

- The last line of code, module.exports = router, exports the router. This means that the router can be used in other parts of the application.

- Here is an explanation of the routes in the router:

  - router.get("/"): This route handles GET requests to the /posts endpoint. The postsController.getAll function is used to handle these requests.
  - router.get("/:id"): This route handles GET requests to the /posts/:id endpoint. The postsController.getById function is used to handle these requests.
  - router.post("/"): This route handles POST requests to the /posts endpoint. The postsController.create function is used to handle these requests.
  - router.put("/:id"): This route handles PUT requests to the /posts/:id endpoint. The postsController.update function is used to handle these requests.
  - router.delete("/:id"): This route handles DELETE requests to the /posts/:id endpoint. The postsController.delete function is used to handle these requests.