**Alankrit Bhardwaj**

**Nurukurthi Prakash**

**MDITS ASSINGMENT**

For this assignment we have successfully tried to demonstrate how to build CRUD APIS using Node.js, express.js with MySql. And authentication was implemented with JWT.

**Graphical user interface, diagram

Description automatically generated**

The given picture describes how the connection between client and database is made through REST API. The client (**postman** in our case) uses the rest API (**Node.js**) to pass a query to database (**MySQl**) which in turn send the data to the REST API interface and then back to client.

Let’s see how to use Node.js to create CRUD APIs-

To create the complete the project we need to install certain libraries like:

1) **bcrypt-** It is used to encrypt the password.

Text

Description automatically generated

2) **jsonwebtoken(JWT)-** To generate a token so that user can be authenticated

Text

Description automatically generated

Writing the code for NODE JS:

Before going to the coding part let’s understand some of the basic terminology:

a) **Services** - The services contain the database queries and returning objects or throwing errors.

**b) Controllers** - The controllers handles all the logic behind validating request parameters, query, Sending Responses with correct codes.

c) **Routes** - The API routes maps to the Controllers.

Let’s start with connection between mysql and Node.js

Step1: Create a file with database.js.

Step2: Write the following code to connect with the mysql database present on your device-

Text

Description automatically generated

Let’s make the service file for our application:

The following lines of code will insert data inside our database(**test**) inside table **registration**.

Text

Description automatically generated

The **module.exports** is a special object which is included in every JavaScript file in the Node.js application by default. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to **module.exports** will be exposed as a module.

**Pool.query** will allow you to execute a basic single query when you need to execute from a client that would be accessed from the pool of client threads. So pool.query could be used to directly run the query rather than acquiring a client and then running the query with that client. It takes in 3 parameter-

a) String query which needs to be run on the mysql server.

b) An array of value which needs to be passed through the query.

c) It contains a function to return an error or and result from the database.

After creating all the example api the file will look like this-

const pool=require("../../config/database");

module.exports={

create:(data,callBack)=>{

pool.query(

'insert into registration(id,firstName,lastName,gender,email,password,number) values(?,?,?,?,?,?,?)',

[data.id,

data.firstName,

data.lastName,

data.gender,

data.email,

data.password,

data.number],

(error,results,fields)=>{

if(error){

return callBack(error);

}

return callBack(null,results);

}

);

},

getUsers:callBack=>{

pool.query(

'select id,firstName,lastName,gender,email,password,number from registration',

[],

(error,results,fields)=>{

if(error){

return callBack(error);

}

return callBack(null,results);

}

);

},

getUsersbyUserId:(id,callBack)=>{

pool.query(

'select id,firstName,lastName,gender,email,number from registration where id=?',

[id],

(error,results,fields)=>{

if(error){

return callBack(error);

}

return callBack(null,results[0]);

}

);

},

updateUserById:(data,callBack)=>{

pool.query(

'update registration set firstName=?,lastName=?,gender=?,email=?,password=?,number=? where id=?',

[

data.firstName,

data.lastName,

data.gender,

data.email,

data.password,

data.number,data.id],

(error,results,fields)=>{

if(error){

return callBack(error);

}

return callBack(null,results);

}

);

},

deleteByUserId:(data,callBack)=>{

pool.query(

'delete from registration where id=?',

[data.id],

(error,results,fields)=>{

if(error){

return callBack(error);

}

return callBack(null,results);

}

);

},

getUserByUserEmail: (email, callBack) => {

pool.query(

`select \* from registration where email = ?`,

[email],

(error, results, fields) => {

if (error) {

callBack(error);

}

return callBack(null, results[0]);

}

);

}

}

Now let’s create the controller class which will run a function on the data received from the database through service class-

Text

Description automatically generated

Here, we write the code which will be run when a certain endpoint is called through the routers. We use the **create** method which we created inside the service and pass the body of data (JSON) send by the client application.

After Creating all the functions our service.js class will look like this-

createUser: (req,res)=>{

const body=req.body;

const salt=genSaltSync(10);

//body.password=hashSync(body.password,salt);

create(body,(err,results)=>{

if(err){

console.log(err);

return res.status(500).json({

success:0,

message:"Database connection error"

});

}

return res.status(200).json(

{

success:1,

data: results

}

);

});

},

getUsersbyUserId: (req,res)=>{

const id=req.params.id;

getUsersbyUserId(id,(err,results)=>{

if(err){

console.log(err);

return;

}

if(!results){

return res.json(

{

success:0,

message:"Record not Found"

}

);

}

return res.json({

success:1,

data:results

});

});

},

getUsers: (req,res)=>{

getUsers((err,results)=>{

if(err){

console.log(err);

return;

}

return res.json(

{

success:1,

data: results

}

);

});

},

updateUserById: (req,res)=>{

const body=req.body;

const salt=genSaltSync(10);

// body.password=hashSync(body.password,salt);

updateUserById(body,(err,results)=>{

if(err){

console.log(err);

return ;

}

return res.json(

{

success:1,

data: "Updated Successfully"

}

);

});

},

deleteByUserId: (req,res)=>{

const body=req.body;

deleteByUserId(body,(err,results)=>{

if(err){

console.log(err);

return ;

}

if(!results){

return res.json(

{

success:0,

data: "Record not Found"

}

);

}

return res.json(

{

success:1,

data: "User Deleted Successfully"

}

);

});

}

Let’s create the router class for the given data:

const{createUser,getUsers,getUsersbyUserId,deleteByUserId,updateUserById,login} =require("./user.controller");

const router=require("express").Router();

const {checkToken}=require("../../auth/token\_validation");

router.post("/",createUser);

router.get('/',getUsers);

router.get('/:id',getUsersbyUserId);

router.patch('/:id',updateUserById);

router.delete('/',deleteByUserId)

module.exports=router;

We create the endpoint which needs to be called while accessing the apis which we created.

JWT

JSON Web Token is a proposed Internet standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims. The tokens are signed either using a private secret or a public/private key.

Diagram

Description automatically generated

For example, a server could generate a token that has the claim "logged in as administrator" and provide that to a client. The client could then use that token to prove that it is logged in as admin. The tokens can be signed by one party's private key (usually the server's) so that party can subsequently verify the token is legitimate. If the other party, by some suitable and trustworthy means, is in possession of the corresponding public key, they too are able to verify the token's legitimacy. The tokens are designed to be compact,URL-safe, and usable especially in a web-browser single-sign-on (SSO) context. JWT claims can typically be used to pass identity of authenticated users between an identity provider and a service provider, or any other type of claims as required by business processes

For implementing jwt we create another controller class- token\_validation, also a new function in controller and service class-

Service.js class

Text

Description automatically generated

This is to get the user data from the database with the help of email. So now, we can use this query to get the data and pass it to callBack function which will be called in controller class.

Controller.js

login: (req, res) => {

const body = req.body;

getUserByUserEmail(body.email, (err, results) => {

if (err) {

console.log(err);

}

if (!results) {

return res.json({

success: 0,

data: "Invalid email or password"

});

}

console.log(results);

var result;

if(body.password===results.password){

result=true;

}else{

result=false;

}

console.log(body.password+" "+ results.password+" "+result);

if (result) {

results.password = undefined;

const jsontoken = sign({ result: results }, "qwe1234", {

expiresIn: "1h"

});

return res.json({

success: 1,

message: "login successfully",

token: jsontoken

});

} else {

return res.json({

success: 0,

data: "Invalid email or password"

});

}

});

}

Here we take in the password and compare it to the one in database by decrypting them and send the appropriate result.

To verify the token which would be send to us after login we need to create another class of token\_verification:

const {verify}=require("jsonwebtoken");

module.exports={

checkToken: (req, res, next) => {

let token = req.get("authorization");

if (token) {

// Remove Bearer from string

token = token.slice(7);

verify(token, "qwe1234", (err, decoded) => {

if (err) {

return res.json({

success: 0,

message: "Invalid Token..."

});

} else {

req.decoded = decoded;

next();

}

});

} else {

return res.json({

success: 0,

message: "Access Denied! Unauthorized User"

});

}

}

}

Now, let’s update our router routes:

Text

Description automatically generated

All the path other than is only accessible if we provide the authentication token to the server and will only work if the user is authenticated.

Output Screen shot:

The following output takes in the email and password and authenticate them and will send us the token which needs to be send for every other authentication.

A screenshot of a computer

Description automatically generated

Getting the user by its id:

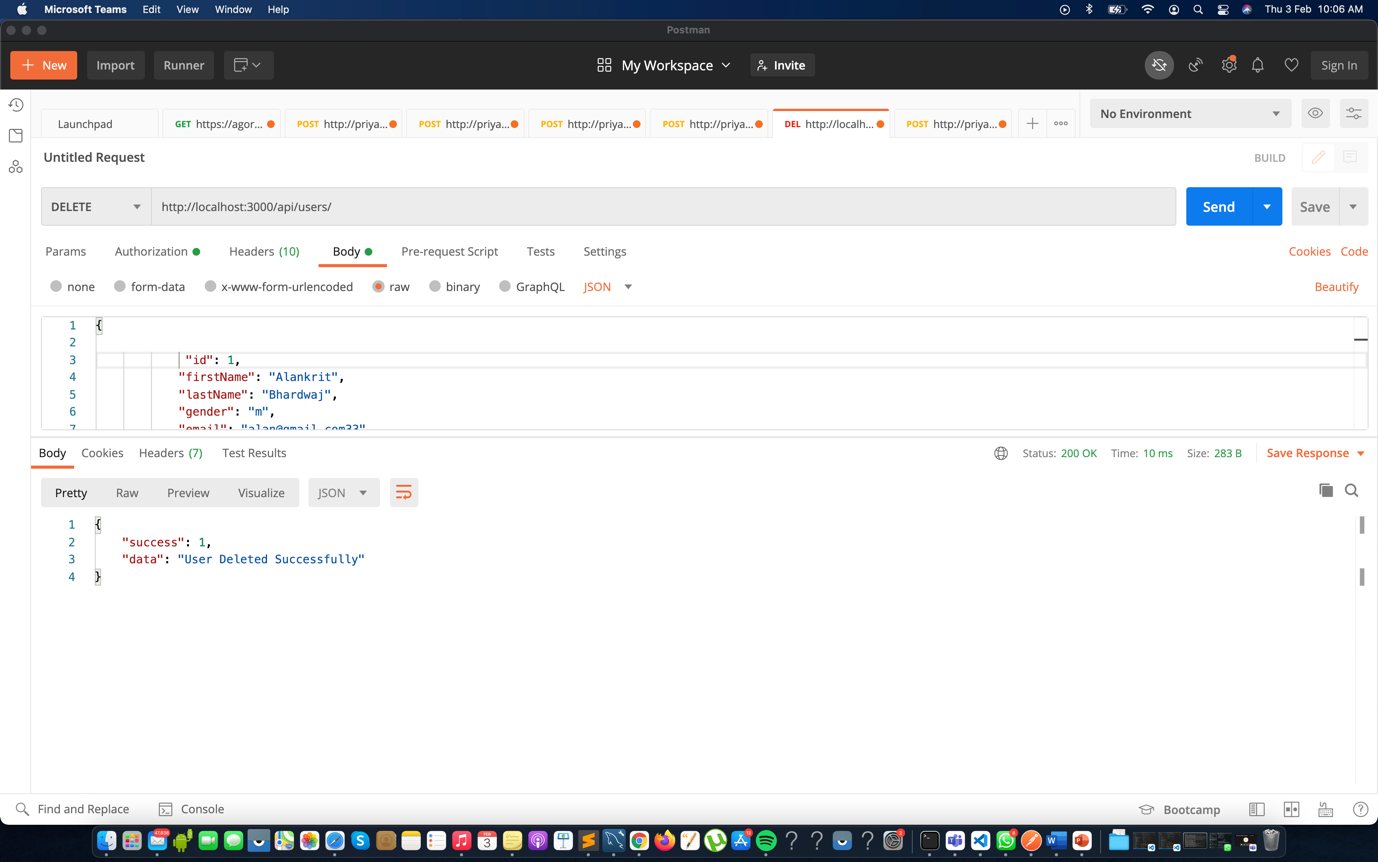
Graphical user interface, text, application, email

Description automatically generated

Updating the user details by its id:Graphical user interface, text, application, email

Description automatically generated

Deleting a user by its id:



Creating a new user:

Graphical user interface, text, application, email

Description automatically generated