Chapter 2

RESTFUL Web Services with Node js: Creating our first RESTFUL Web Service

RESTful APIs

Based on the above table, we are going to provide the following RESTful APIs.

S/N	URL	HTTP Method	POST Body	Result
1	user/<num>	GET	empty	Retrieve data of user with id <num>
2	user	GET	empty	Retrieve data of all users
3	user	POST	JSON String of user details	Insert new user record
4	user/<num>	POST	JSON String of updated user details	Update user with id <num>
5	user/<num>	DELETE	empty	Delete user with id <num>

We will be designing our web services based on the Model-View-Controller(MVC) Design. Our web services will function in the Controller layer, and call the model layer js files for data extraction and modification in the database latyer.

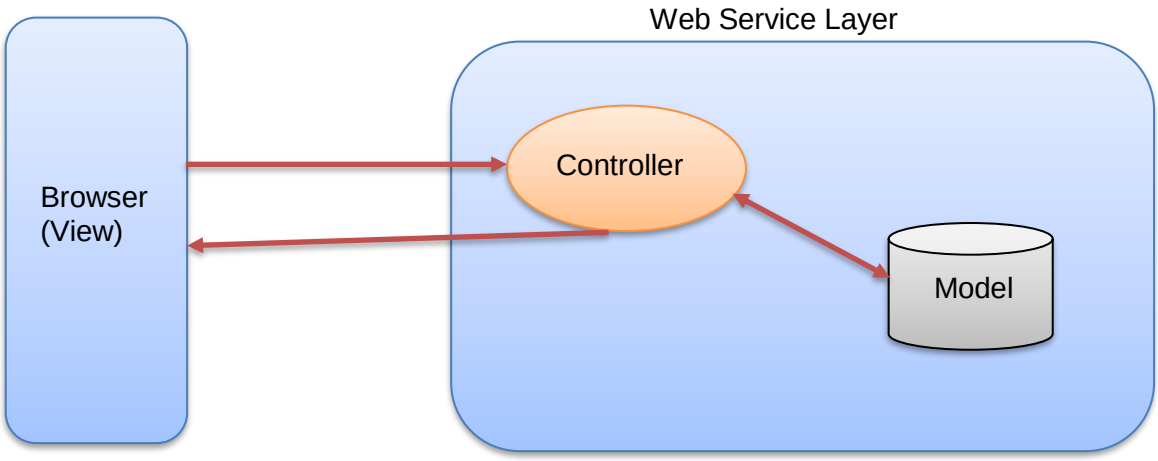
What is MVC ?

MVC is a simple architecture where all components are separated into three classes:

Model - Classes that contain data that will be shown to the user.

View - Components that will display the model to the user.

Controller - Components that will handle any interaction with the user.



The simple scenario of processing an MVC request is described in the following steps:

1. The user enters in the browser some URL that is sent to the server, e.g., http://localhost/user

2. The user request is analyzed by the framework in order to determine what controller (web service method) should be called.

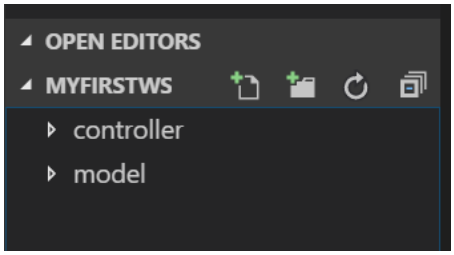
3. The Controller takes the parameters that the user has sent, calls the model to fetch some data, and loads the model object (JSON data) that should be displayed.

4. The Controller passes the model object back to the view.

5. The View gets data from the model, puts it into the HTML template, and displays the response to the user browser.

Creating our GET Method for retrieving user data

Let's setup our application directory and download the necessary libraries. Navigate to your hard drive and create a new directory called myFirstWS. In Visual Studio Code, open the directory and create 2 more separate directories controller and model. We will structure the code using the Model-View-Controller(MVC) design pattern. As web services is meant to be consumed by an external view layer, we will only create controller and model layer for this project.



Our webservices will be fetching or updating data from the mysql database. In order to accomplish that, we need to download libraries for mysql. We will also download the library body-parser as we need to process POST data.

Open the run from the integrated command terminal to setup necessary packages for your project:

npm init
npm install mysql --save
npm install body-parser --save
npm install express –save
npm install cors --save

We will next design the model layer which focuses on processing of data to/from database.

Let us first create write code to create a connection from our web app to the mysql database.

Defining and creating the database connection in databaseConfig.js:

We will make use of the createConnection method from mysql library api to create a connection to the database.

```
var mysql = require('mysql');

var conn = mysql.createConnection({

            host: "localhost",

            user: "root",

            password: "root",

            database: "batam"

});
```

Load mysql library

Define the connection settings for mysql database

For the connection settings, we need to specify the host ip, database user account and password, and the database schema to connect to.

As the database connection and its settings will be used frequently by different js modules, we will define the codes in a module.

Create the below file databaseConfig.js in the model folder:

```
var mysql = require('mysql');

var dbconnect = {

    getConnection: function () {

        var conn = mysql.createConnection({

            host: "localhost",

            user: "root",

            password: "root",

            database: "batam"

        });

        return conn;

    }

};

module.exports = dbconnect
```

Creating functions for database access:

Now that we have created the database configuration settings to obtain the connection settings, we will proceed to design our database calls to access data in the db1 schema. This js module will be handling all database operations to the **user** table and used by the restful web service.

(**Note**: For security purposes, password should not be stored in plain text and retrieved over to the client browser in real life. You can install libraries like bcrypt. For simplicity, in this exercise, we will retrieve all data relating to the user in the web service get method)

For this section, we will be creating and calling asynchronous functions which allow us to pass in a callback function to handle results from the asynchronous function. Node.js, being an asynchronous platform, doesn't wait for things like file I/O to finish and code will just continue to execute. Once the task completes, a callback function is passed in by caller to handle the result if need be. Various functions in the mysql module library is asynchronous, and we have to write functions to handle the callback.

We will be creating an asynchronous function called getUser that will return a callback function once the data is returned from the database.

First we will be importing the databaseConfig.js module:

```
var db = require('./databaseConfig.js');
```

With the configuration imported, we can proceed to connect to MySQL and write functions to query the database table and retrieve the results. The function we are creating will be called getUser and it takes in a parameter userid representing the user's id and returns a callback function to the caller containing the error(if any) and the results. We will create a variable userDB representing the object with the DB functions running database operations to the user table.

The object and its corresponding function will look like the below code:

```
var userDB = {

    getUser: function (userid, callback) {

                  …..

    }

}
```

Inside the getUser function, we will get the connection configuration settings for the mysql database. Afterwards, we proceed to connect to the mysql database and the db1 schema by calling the connect function. The connect function is an asynchronous function so we will provide a callback function to handle the error if any.

```
var conn = db.getConnection();
conn.connect(function (err) {
    if (err) {
        console.log(err);
        return callback(err,null);
    }
    else {
        …..
    }
});
```

Callback function to handle results from connection

Error from connection detected

Connection successful, proceed to do the query

Querying the database

We will issue a query to the database table to retrieve details of a user with a particular user id provided by the caller. To prevent SQL injection, we need to escape the user supplied values. Our query will be 'SELECT * FROM user WHERE userid = ?' with ? representing the user input value that has to be escaped to prevent SQL injection. After which we will be calling the query function to fetch the results from the database.

```
console.log("Connected!");
var sql = 'SELECT * FROM user WHERE userid = ?';

conn.query(sql, [userid], function (err, result) {
    conn.end();

    if (err) {
        console.log(err);
        return callback(err,null);
    } else {
        console.log(result);
        return callback(null, result);
    }
});
```

Array of values to replace the ? placeholder

Error detected we return a callback function with an error and null results

Result retrieved successfully and we return a callback with null error and a result.

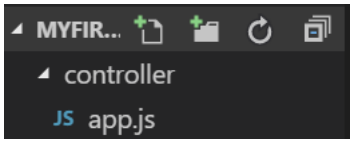
Full Source Code for user.js:

```
var db = require('./databaseConfig.js');

var userDB = {

    getUser: function (userid, callback) {

        var conn = db.getConnection();

        conn.connect(function (err) {

            if (err) {

                console.log(err);

                return callback(err,null);

            }

            else {

                console.log("Connected!");

                var sql = 'SELECT * FROM user WHERE userid = ?';

                conn.query(sql, [userid], function (err, result) {

                    conn.end();

                    if (err) {

                        console.log(err);

                        return callback(err,null);

                    } else {

                        return callback(null, result);

                    }

                });

            }

        });

    }
```

```
}

module.exports = userDB
```

Defining the routing in the controller layer

At the controller layer, we will create a new router app.js to define the application routing.



The web service we are creating has a get method and takes in the userid provided by the caller to retrieve details of the user matching the user id.

```
var express = require('express');
var cors = require('cors');

var app = express();
app.use(cors());

var user = require('../model/user.js');

app.get('/api/user/:userid', function (req, res) {
    …


});
```

Load express library

Load cors library

Create an instance of express

Use cors for cross origin support

Callback function to handle the request and response

Path of web service. :userid refers to an input value by caller that will be mapped to userid var

The req object in the callback function refers to the http Request object and you can retrieve request query params, body, headers and cookies from it.

The res object refers to the http Response object, which is the response sent back to the client browser. Note that once res.send() or res.redirect() or res.render() is called, you can't do it again, else there will be an error.

Full Source code for app.js:

```
var express = require('express');
var app = express();
var user = require('../model/user.js');

app.get('/api/user/:userid', function (req, res) {
    var id = req.params.userid;

    user.getUser(id, function (err, result) {
        if (!err) {
            res.send(result);
        }else{
            res.status(500).send("Some error");
        }
    });

});

module.exports = app
```

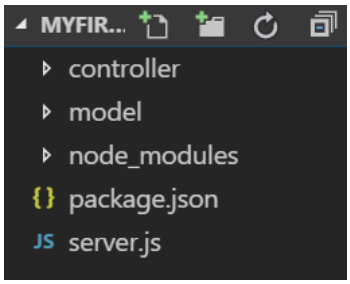
Gets caller supplied parameter called userid defined as part of the url

Calls the getUser method we defined in user.js previously to query the database

Send to browser the result if no error. Else indicate an error status code to browser

Creating our main server in the root folder

Finally at the root folder, we create server.js to listen at port 8080




```
var app = require('./controller/app.js');

var server = app.listen(8081, function () {

    var port = server.address().port;

    console.log('Web App Hosted at http://localhost:%s',port);

});
```

Finally, we run node server.js on the terminal.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS D:\batam\myFirstWS> node server.js
Web App Hosted at http://localhost:8081
```

To test the web service, we can run http://localhost:8081/api/user/1.

localhost:8081/user/1

[{"userid":1,"username":"Samsudin","email":"samsudin@bmail.com","role":"user","password":"abc123"}]

That's it, we have our first web service!
Exercise 1:

Now that you have written your first webservice method, try implementing the second get webservice to retrieve all users from the user table.

You have to implement the following:

1) In user.js add another method and implement your code to do the connection and retrieval of results:

```
   getUsers: function (callback) {

      var conn = db.getConnection();

      //implement the database query and return result if successful
    }
```

2) In app,js, add another get method to call getUsers:

```
app.get('/api/user', function (req, res) {

    user.getUsers( function (err, result) {
        if (!err) {
            res.send(result);
        }
        else{
             console.log(result);

             res.status(500).send("Some error");
        }
    });

});
```

3) Sample Output:

localhost:8081/api/user

[{"userid":1,"username":"Samsudin","email":"samsudin@bmail.com","role":"user","password":"abc123"},
{"userid":2,"username":"hidayat","email":"hidayat@imail.com","role":"admin","password":"a12112x"}]

Creating POST methods for Inserting and Retrieving User Data

In user.js, we will add another asynchronous function to handle the creation of a new user in the database.

The procedure is generally the same, except we don't have data records that are returned. We will however extract the number of rows that were affected by the SQL statement by calling result.affectedRows.

```
addUser: function (username, email, role, password, callback) {

        var conn = db.getConnection();
        conn.connect(function (err) {
            if (err) {
                console.log(err);
                return callback(err,null);
            }
            else {
                console.log("Connected!");

                var sql = 'Insert into
user(username,email,role,password) values(?,?,?,?)';

                conn.query(sql, [username, email, role, password],
function (err, result) {
                    conn.end();

                    if (err) {
                        console.log(err);
                        return callback(err,null);

                    } else {

                        console.log(result.affectedRows);

                        return callback(null,result.affectedRows);

                    }
                });

            }

        });

}
```

Insert sql statement with 4 target fields to insert

Return number of record(s) inserted

In app.js, we will add a new route for the new post method for inserting the new record in the database table.

To handle HTTP POST request in Express.js, we need to use a middleware module called body-parser. The body-parser can extract the entire body portion of an incoming request stream and allow access of this data by using req.body.

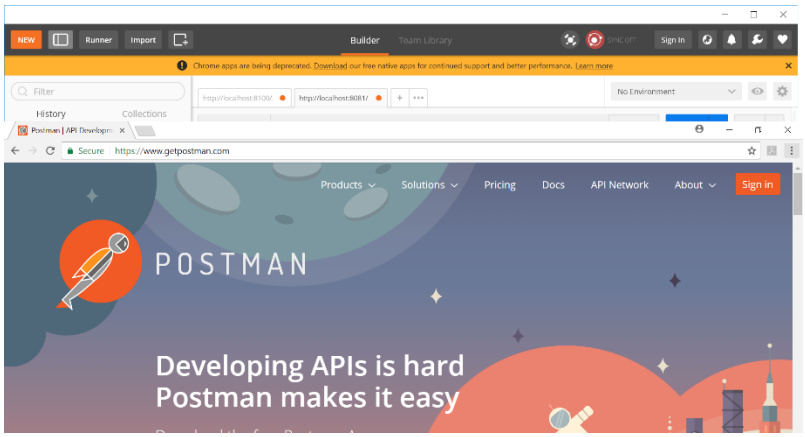
This body-parser module can be used to parse JSON, string and URL encoded data submitted through HTTP POST.

```
var bodyParser = require('body-parser');
var urlencodedParser = bodyParser.urlencoded({ extended: false });

app.post('/api/user', urlencodedParser, function (req, res) {

          var username = req.body.username;
          var email = req.body.email;
          var role = req.body.role;
          var password = req.body.password;

          user.addUser(username, email, role, password, function (err,
      result) {
              if (!err) {
                  console.log(result);
                  res.send(result + ' record inserted');
              } else{
                  res.send(err.statusCode);

              }
          });

});
```

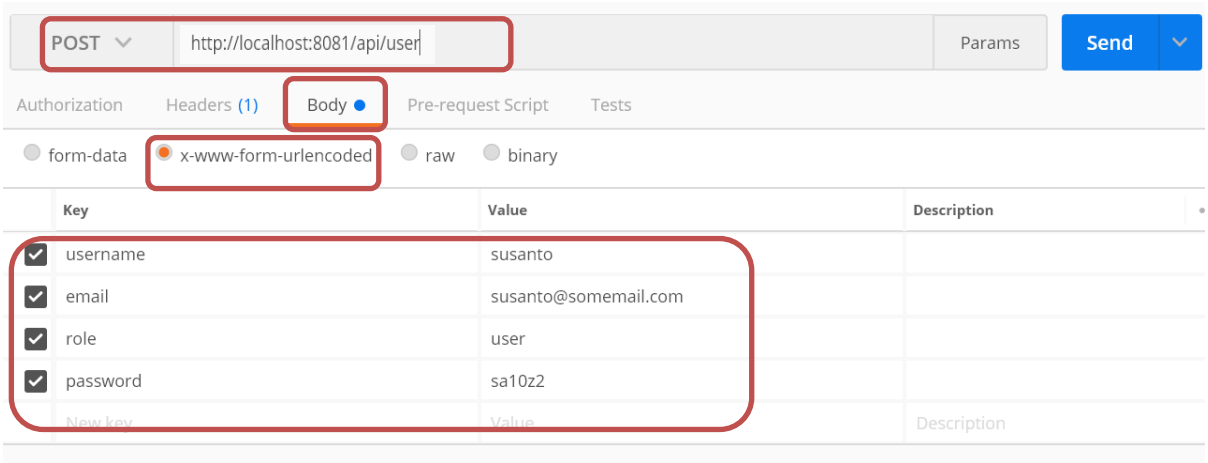
Usage of body-parser to parse HTTP POST data

Retrieve the POST data fields representing the 4 columns of data from user table

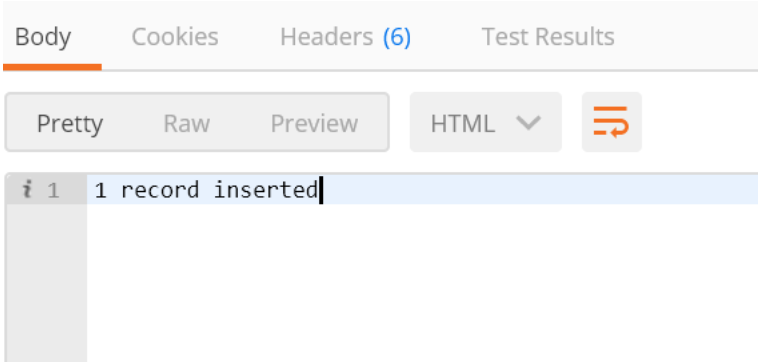
Finally, to test the POST webservice, we run server.js and test the webservice with a chrome plugin POSTMAN downloadable from https://www.getpostman.com/.



Startup POSTMAN and test the webservice by keying in the below fields and following the various selections:



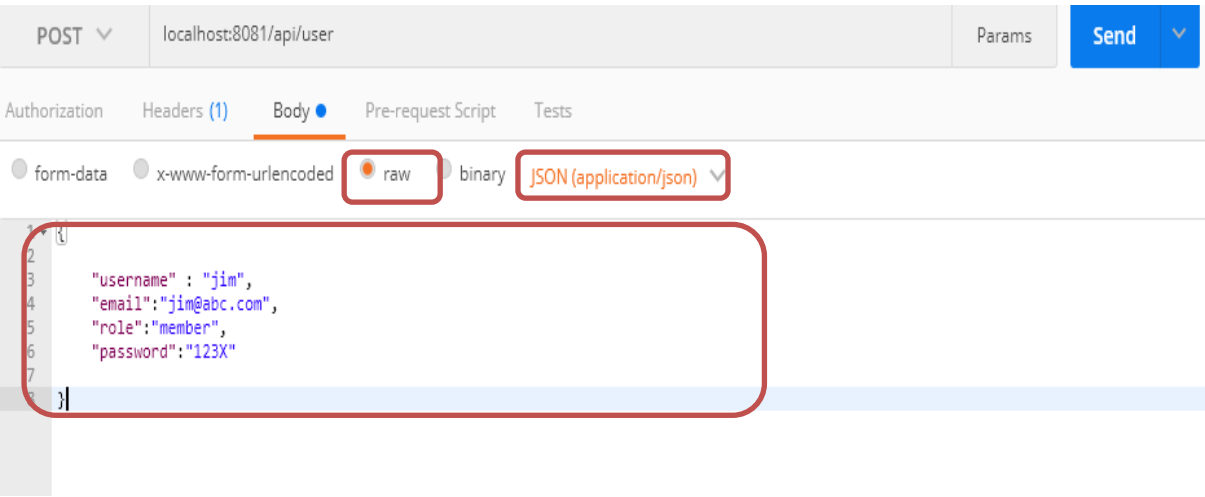
After clicking on send, you should get the below result:



What if the input data to web service is in json format?

Handling Json data with bodyParser.json:

```
var jsonParser = bodyParser.json();

app.post('/api/user', urlencodedParser,jsonParser, function (req, res) {

…
}
```



Exercise 2:

As an exercise, try implementing the post webservice to update the email and password of a user in the user table.

You have to implement the following:

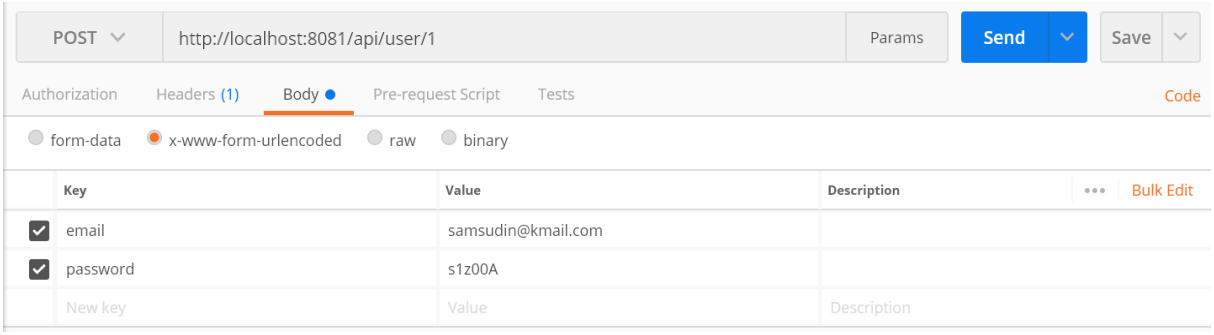
1) In user.js add another method and implement your code to do the connection and updating of record based on supplied userid in url:

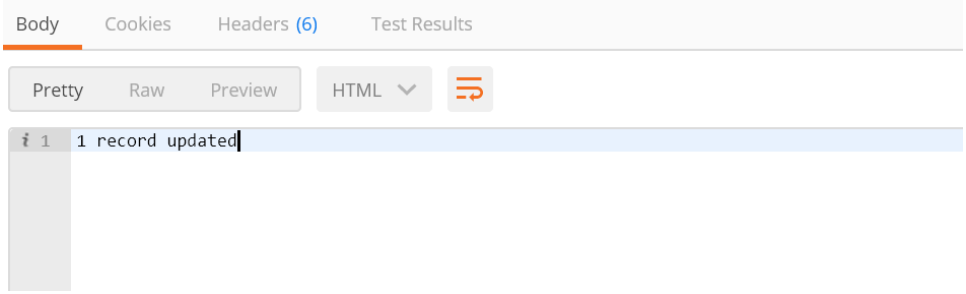
```
  updateUser: function (email,password,userid, callback) {

      var conn = db.getConnection();

      //The sql should be similar to var sql = 'Update user set
      email=?,password=? //where userid=?';
      //your code

    }
```

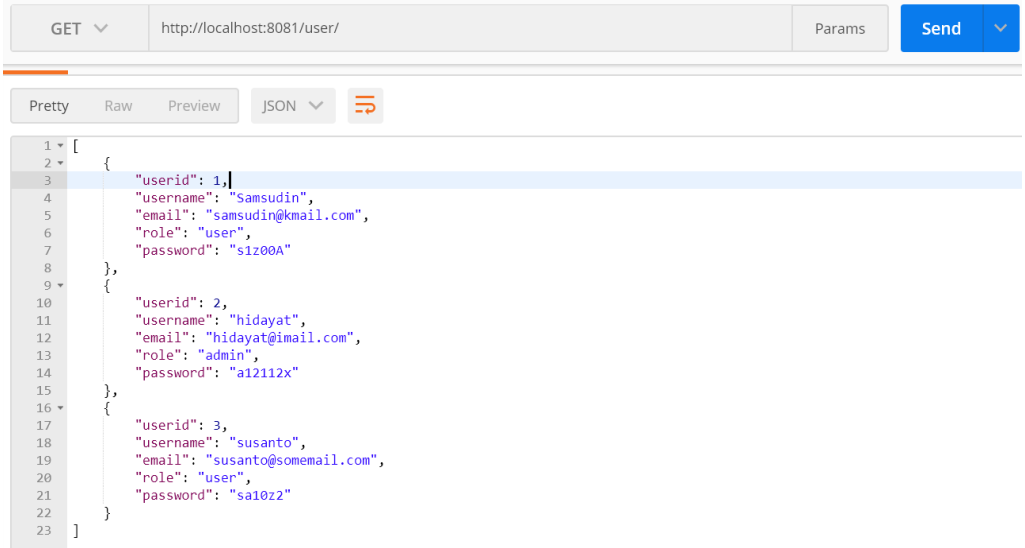
2) In app,js, add another post method to retrieve the parameters and call updateUsers:

```
app.post('/api/user/:userid', urlencodedParser, jsonParser, function
(req, res) {

    //implement your code


});
```

3) Sample Test and Output:







Creating DELETE method for Deleting User Data

Finally, we will create the 5th web service for deletion of data. In user.js, we will add another asynchronous function to handle the deletion of a user in the database.

Like the post method, we will extract the number of rows that were affected by the SQL statement by calling result.affectedRows to confirm the deletion of the record.

```
deleteUser: function (userid, callback) {

      var conn = db.getConnection();
      conn.connect(function (err) {
          if (err) {
              console.log(err);
              return callback(err,null);
          }
          else {

              console.log("Connected!");

              var sql = 'Delete from user where userid=?';

              conn.query(sql, [userid], function (err, result) {
                  conn.end();

                  if (err) {
                      console.log(err);
                      return callback(err,null);

                  } else {

                      return callback(null,result.affectedRows);

                  }
              });

          }

      });

  }
```

SQL to do the deletion of the record

In app.js, we will add a new route for the new delete method for deleting a record in the user database table. Retrieval of the userid is similar to that as in get as it is provided as part of the url.

```
app.delete('/api/user/:userid', function (req, res) {

    var userid = req.params.userid;

    user.deleteUser(userid, function (err, result) {
        if (!err) {

            res.send(result + ' record deleted');
        }else{
            console.log(err);

            res.status(500).send("Some error");

        }
    });

});
```