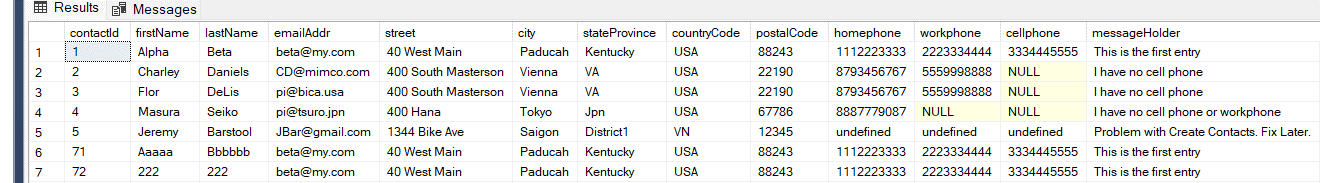
This objective of this lesson will be to connect to and retrieve single users in the database from a single http request. In order to select the correct database record, we will pass the ‘contact ID number’ as a parameter of the HTTP GET request. We will use Chrome Browser to test this function by keying in a ‘known’ contactId number.

My dataset in SQL Server includes the following:



We can select by contactId. We choose contactId=3, so our webpage url should read as follows:

http:/localhost:5000/3 Notice the parameter value (3) is used.

1. In dbOperations, create the interface function called ‘getThisContact’ so that we can retrieve that particular contact by contactId number: code for getThisContact is as follows:

const getThisContact = async(contactId) => {

    try {

       let pool = await sql.connect(config); // Log on to database

       let theseContacts = await pool.request()

           .input('input\_parameter', sql.Int, contactId)

           .query( "SELECT \* FROM Contacts where contactId =  @input\_parameter");

           return theseContacts;

    }  // end try block

    catch(error) {

        console.log("DATABASE Query ERROR IN     getThisContact  Error Posted Next Line");

        console.log(error);

    } // end catch

}  // end getThisContact

// Export getThisContact so that ‘allroutes’ is aware of its existence

module.exports = { getContacts, getThisContact}

1. In ‘allRoutes.js’ the dbOperations file is already imported, so we do not need to import additional modules. Write the route to access the ‘contacts’ table by ‘contactid’ and call the database interface function getThisContact from ‘dbOperations’.

router.route('/contacts/:contactId').get((request, response)=> {

  dboperations.getThisContact(request.params.contactId).then(result => {

    //  response.send(result);

      response.json(result);

  })

})

1. The complete ‘allroutes.js’ and ‘dbOperations.js’ files are listed below. Save all files and test with an ID number that you see in your specific SQL-SERVER table. Regarding the json information which is displays on the Chrome Browser: see the Special Note on the OUTPUT that is listed below.

=========== =========== =========== =========== =========== =========== =========

Complete files:

// file:  dboperations.js

const config = require('./dbconfig'); // access database configuration

const sql = require('mssql');

const getContacts = async() => {

     try {

        let pool = await sql.connect(config); // Log on to database

        let theseContacts = await pool.request().query(

            "SELECT \* FROM Contacts"

        );

        return theseContacts;

     }  // end try block

     catch(error) {

         console.log("DATABASE CONNECTION ERROR IN     getThisContact  Error Posted Next Line");

         console.log(error);

     } // end catch

}  // end getContacts

const getThisContact = async(contactId) => {

    try {

       let pool = await sql.connect(config); // Log on to database

       let theseContacts = await pool.request()

           .input('input\_parameter', sql.Int, contactId)

           .query( "SELECT \* FROM Contacts where contactId =  @input\_parameter");

           return theseContacts;

    }  // end try block

    catch(error) {

        console.log("DATABASE Query ERROR IN     getThisContact  Error Posted Next Line");

        console.log(error);

    } // end catch

}  // end getThisContact

module.exports = { getContacts, getThisContact}

// file allroutes.js

const express = require('express');

const router = express.Router();

// imports to implement api's

var bodyParser = require('express');

var cors = require('cors');

const app = express();  // to create an object of express

const dboperations = require('../dbfiles/dboperations')

app.use(bodyParser.urlencoded({extended:true}));

app.use(bodyParser.json());

app.use(cors());

router.use((request, response, next) => {

  console.log('middleware');   // for authentication software etc

  next();

})

router.route('/contacts').get((request, response)=> {

  dboperations.getContacts().then(result => {

       console.log(result);

       response.json(result);

   })

})

router.route('/contacts/:contactId').get((request, response)=> {

  dboperations.getThisContact(request.params.contactId).then(result => {

     response.send(result);

   //   response.json(result);

  })

})

router.route('/user').get((request, response)=> {

        console.log(" console    This is your first routed api call");

        response.send(" This is your first routed api call");

    })

    router.route('/test').get((request, response)=> {

      console.log(" console    This is a call to test");

      response.send(" This is a test");

  })

module.exports = router;    // export all of the routes

=========== =========== =========== =========== =========== =========== =========  
Special Note on the OUTPUT

Why do I receive 2 responses for each HTTP request coming in to nodemon.

Example:

Using Chrome web browser was used to generate Get request (Get the info in the database for a certain id.)

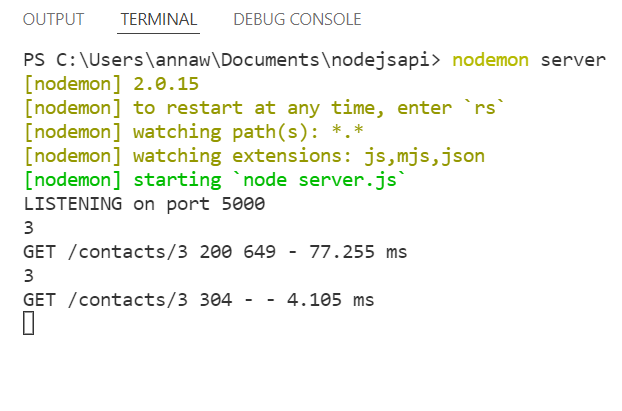
<http://localhost:5000/3>

was sent,

The console shows that, as a result of the single request, two request messages were sent to the back end server.

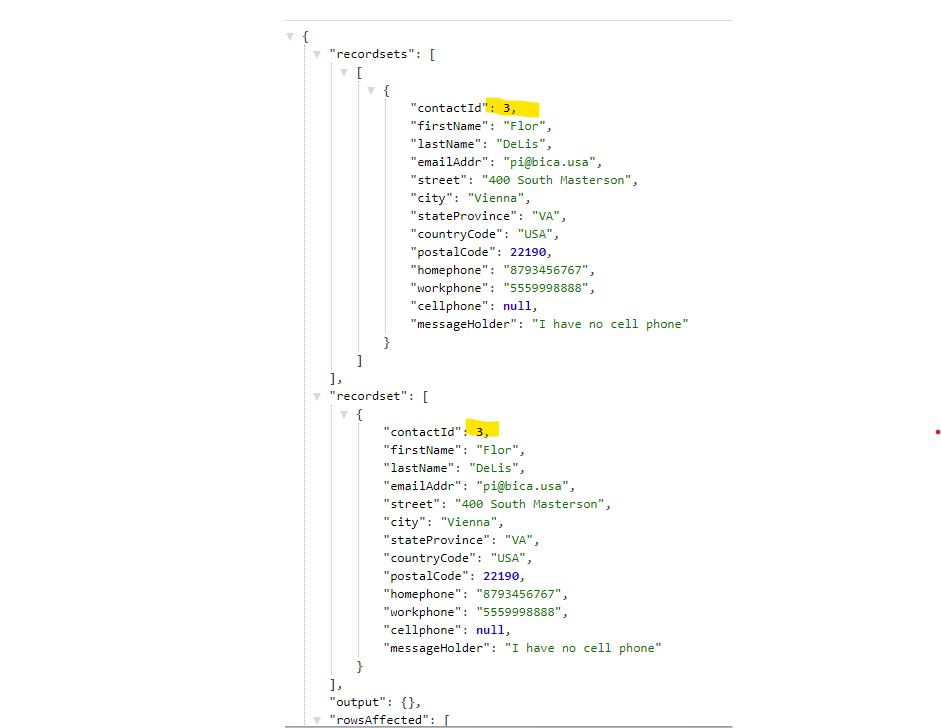
The first request (receiving a status code of 200) is called a CORS preflight message in which the backend software checks that all is well with the request so the message is s.

The second request (receiving a status code of 30) sends back the complete response (data including the data from the SQL Server)



The data received by the Chrome Browser is as follows

:



CORS: the *Cross Origin Resource Sharing* is used to allow node to connect to Express middleware. CORS will allow access to clients from different systems. Consequently, the first response sent to the browser should be a code 200: “I have checked your ‘header’ and everything is appropriate. I will now process the HTTP data, at which time the HTTP response will then now transmit the data in the second reply with the data that you requested.

Example:

