Lab 8 - REST and CRUD Tutorial (Node.js)

Material by Kyle Helmick in the Spring of 2017 for use in CSCI 3308

Objectives

- See how CRUD and REST methods can work together
- Please pair-program today!

Background

Nodejs is a popular JavaScript runtime. It uses Chrome's V8 JavaScript engine (what's built into chrome to handle JavaScript on webpages). It uses npm as its package manager (similar to apt-get for debian based linux systems).

Exercise

First we will have to set up our working environments. This includes installing nodejs, npm, and build-essential.

```
cURL -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
sudo apt-get install -y nodejs build-essential
```

We can make sure they're both installed by checking their versions.

```
node -v
npm -v
```

More Background: Express is a popular minimalist web framework for node.js. It is completely written in JavaScript and provides helpful function and methods for your web application! We're going to utilize their app generator to make our base app. To install it we use npm (our node package manager) and we will be installing it globally (-g).

```
npm install express-generator -g
```

Now that we have our app generator installed let's go ahead and make an app with our view engine set to pug (formerly known as jade).

```
express --view=pug express_mysql_tutorial
```

Let’s cd into our express_mysql_tutorial folder and check out a couple of files and folders.

app.js - the main configuration for your web app, this is where “routes” to different pages can be declared and if wanted other packages can be 'required' and used.

bin/www - this is our "server" file, it’s the main configuration for our websites server. You can change things like what port it’s on, etc...

package.json - this is another configuration for your app, it's kind of like meta data for the app, it holds all of the links to packages needed for the app to run, helpful information regarding the app, and commands to test it, one of them is npm start.

public/ - this is where you can store publicly available scripts, images, or stylesheets. Your users will be able to see files in this directory on their browsers so storing sensitive information here is not recommended.

routes/ - routes are like links for your web app. If you want a subpage like localhost:3000/birds, you can make a route in this folder to help direct to the correct page and subpages. Also routes can do work for you, our routes will access our MySQL database and send (route) information to our pug files.

views/ - this is where your page scripts are held. They are in pug (jade) and are how you display things, they can take and display variables passed from your routes.

Note that we might have a bunch of files but no actual server files or packages or anything, so we’ll type:

```
npm install
```

What this does is it checks our package.json file for any dependencies (packages) that are required for our server to run, and you’ll see there are several, and downloads the required files into a node_modules directory. This holds all of the methods that our packages need. We’re going to need one more package **saved** to really get this show going though and that’s mysql.

```
npm install mysql --save
```

Now that all of the setup (it’s a lot of work to get running but pays off in the long run) let’s see if it works!

```
npm start
```

app.js

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var index = require('./routes/index');
var users = require('./routes/users');
var update = require('./routes/update');
var del = require('./routes/delete');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', index);
app.use('/users', users);
app.use('/update', update);
app.use('/delete', del);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handler
app.use(function(err, req, res, next) {
```

In this step we're going to be setting up our routes for our website. We're adding two called update and delete.

index.js

```
var express = require('express');
var router = express.Router();
var mysql = require('mysql');
var bodyParser = require('body-parser');

var jsonParser = bodyParser.json();
/* GET home page. */
router.get('/', function(req, res, next) {

    var connection = mysql.createConnection( {
      host: 'localhost',
      user: 'root',
      password: 'passw0rd',
      database: 'Lab'
    });

    connection.connect( function(err) {
    if (err) {
      res.render('error', { error: err });
    }
    else {
      connection.query('SELECT * FROM store', function (mysql_err, mysql_res,
      mysql_fields) {
        connection.end();
        if (mysql_err) {
          res.render('error', { error: mysql_err });
        }
        else {
          res.render('index', {title: 'Express', mysql: mysql_res});
        }
      });
    }
  });
});

router.post('/', jsonParser, function(req, res) {

  var connection = mysql.createConnection( {
    host: 'localhost',
    user: 'root',
    password: 'passw0rd',
    database: 'Lab'
  });

  connection.connect( function(err) {
    if (err) {
      res.render('error', { error: err });
    }
    else {
      connection.query('INSERT INTO store SET ?', req.body, function
      (mysql_err, mysql_res, mysql_fields) {
        connection.end();
        if(mysql_err) {
          res.render('error', { error: mysql_err });
        }
        else {
          res.redirect('/');
        }
      });
    }
  });
});

module.exports = router;
```

Create and Display

In this step we make our functions for our possible HTTP methods. On our homepage we're only going to print (**R**ead of **CRUD**) and submit (**C**reate of **CRUD**).

First we make sure to require packages that will make this easy for us and also set our **bodyParser** to use json.

router.get

The landing page for our website, anytime you go to a html page the first call is most likely a get.

Inside we first make a connection with our database, go ahead and fill out the fields for this and play around with it! The docs for **mysqljs** are here.

Then we connect! After a quick error check we send our first query to our MySQL connection and then **end the connection.** It's important to end the connection because if you don't you might end up out of sync with servers and express/mysqljs will whine a lot. Either way, we have a quick error check again and if everything is good we render the index page (in our views folder) with our MySQL response in **mysql_res**.

router.post

This is also a function of our landing page as we will have the ability to add items to the store from the main page. The steps are similar to router.get but out query now is pass **req.body**. This is the information set with the put request from our HTML form. Then we redirect back to our homepage to show the new item instead of rendering a new page.

“But wait! How do we delete and update?”

Remember making those two routes in our app.js file? Let’s actually make those now. Inside of the routes folder make update.js and delete.js. We’ll work on **update.js** first (**U**pdate of **CRUD**).

update.js

```
var express = require('express');
var router = express.Router();
var mysql = require('mysql');
var bodyParser = require('body-parser');

var jsonParser = bodyParser.json();
/* GET home page. */
router.get('/', function(req, res, next) {

  var connection = mysql.createConnection( {
      host: 'localhost',
      user: 'root',
      password: 'passw0rd',
      database: 'Lab'
  });

  connection.connect( function(err) {
    if (err) {
      res.render('error', { error: err });
    }
    else {
      connection.query('UPDATE store SET id = ?, name = ?, qty = ?, price = ?
      WHERE id = ?', [req.query.id, req.query.name, req.query.qty,
      req.query.price, req.query.id], function (mysql_err, mysql_res,
      mysql_fields) {
        connection.end();
        if (mysql_err) {
          res.render('error', { error: mysql_err });
        }
        else {
          res.redirect('/');
        }
      });
    }
  });
});

module.exports = router;
```

This looks a lot like your get and post functions but since we are in the update route our URL path will look like **localhost:3000/update?id=5&name=pineapple&qty=100&price=1** due to our HTML form. Like usual we connect and make a query but this time thanks to express and mysqljs we can pass values from our URL straight to the query with **req.query.<>**. We end the connection and redirect to ‘/.’ Now this might be confusing because this router is getting ‘/’ and the index router gets ‘/’ so how do we know which we are redirecting to? Redirecting is in terms of URL so **localhost:300**0 is index’s ‘/’ while **localhost:3000/update** is update’s ‘/’.

Now we're working on delete.js (**D**elete of **CRUD**)

delete.js

```
var express = require('express');
var router = express.Router();
var mysql = require('mysql');
var bodyParser = require('body-parser');

var jsonParser = bodyParser.json();
/* GET home page. */
router.get('/:id', function(req, res, next) {

  var connection = mysql.createConnection( {
      host: 'localhost',
      user: 'root',
      password: 'passw0rd',
      database: 'Lab'
  });

  connection.connect( function(err) {
    if (err) {
      res.render('error', { error: err });
    }
    else {
      connection.query('DELETE FROM store WHERE id = ?', [req.params.id],
      function (mysql_err, mysql_res, mysql_fields) {
        connection.end();
        if (mysql_err) {
          res.render('error', { error: mysql_err });
        }
        else {
          res.redirect('/');
        }
      });
    }
  });
});

module.exports = router;

```

This one looks a lot like the update but now we have a different URL string to handle. **'/:id**' requests look like **localhost:3000/delete/54** for instance. With this functionality we can call delete on a specific item (by id) and delete it. Having **:id** also lets us pass the value into our get method as **req.params.id**. Just like last update we redirect to '/' being the base URL **localhost:3000.**

Let's see our finished product!

index.pug

```
extends layout

block content
  h1= title
  p Welcome to #{title}
  p Does this work?
  ul
    each val in mysql
      div
        p #{val.id} - #{val.name} - #{val.qty} - #{val.price} -
          a(href="/delete/"+val.id) DELETE

    form(name="postform", method="post")
      input(type="number", placeholder="id", name="id")
      input(type="text", placeholder="name", name="name")
      input(type="number", placeholder="qty", name="qty")
      input(type="float", placeholder="price", name="price")
      input(type="submit", value="SUBMIT")

    form(name="putform", action="/update", method="put")
      input(type="number", placeholder="id", name="id")
      input(type="text", placeholder="name", name="name")
      input(type="number", placeholder="qty", name="qty")
      input(type="float", placeholder="price", name="price")
      input(type="submit", value="UPDATE")

```

So this is pug (jade) and sure, you're wondering where all of **<** these and those **/>** are but don't worry, we're working with something a little smarter now. So from the top we **extend layout**; **layout.pug** is another file in the directory and it has simple presets like a title and a style sheet. We start our **block** of **content**, saying now we're done with declaring the header and the initial body "tag." We make **h1** equal to a variable called **title** (passed from **index.js**) and then start a paragraph under that. In the paragraph we also embed the variable but we have to tell pug that it's a variable and not just the word title with **#{variable_name}**.

Next we'll start an un-ordered list and inside you can read it as for **each val**ue **in mysql** (a variable that we passed from **index.js**) make a **div.** Inside of that div make a **p**aragraph with values id, name, qty and price. After the paragraph generate a link that directs to **/delete/val.id** (maybe something like **/delete/54**?). And well, that was our **R**ead of **CRUD** output!

The two forms that follow are **C**reate and **U**pdate of **CRUD**. They're given names (whatever you want really), our update is told to send to **/update** and then the **method** (whether it was **post** or **put**). The **input**s are fairly explanatory, they have a given **type** (make sure this matches with your table columns), a **placeholder** to help identify them on the page and their "variable" **name**.