# Lab 6: Node.js

**Due.** Wednesday, July 10th, 2019

In this lab, you will continue working on your website from the previous lab. You will integrate your PostgreSQL database and create a back-end server which connects the front-end (your webpages) to the back-end (your database). Again, the files for the website have been updated to work with Node.js. However, you need to update your PostgreSQL database.

## What To Do

In this first part, you will update your PostgreSQL database so that it can work with today's lab. You need to add a password to your database and update the football_players data.

**Part 1: Updating PostgreSQL database**

**Step 1.** During the PostgreSQL installation process, PostgreSQL created a user account named "postgres". You'll be using this user account. Launch your PostgreSQL server and verify the user account name.

**Step 2.** During this lab, we will set up the Node.js server to access the PostgreSQL database. In order to do that, the database has to have a password. Execute the following command to create a password:

```
postgres=# | \password
```

**Step 3.** Now, update the football_db database. First, switch to the database:

```
postgres=# | \c football_db;
```

Then, download the "sql_scripts.txt" file here. Copy and paste the contents of the file into your PostgreSQL terminal.

Exit the PostgreSQL terminal using:

```
postgres=# | \q
```

Ensure that the "postgres" service is running in the background.

**Part 2: Walk through of GET/POST and ejs**

For the second part, you will follow a walkthrough for setting up your website's home page. This will include handling a GET request, POST request, accessing the favorite_colors table in your PostgreSQL database, and working with EJS (Embedded JavaScript templates). In this walkthrough, you will learn all the things necessary to finish the remaining webpages in Part 3.

**Step 0.** Download the template files for the website here.

**Step 1.** First, you need to setup the routing to our home page (a way for us to access it) using a GET request. Add the following lines of code your server.js file below the comment `/*Add your other get/post request handlers below here: */` :

```
/*Add your other get/post request handlers below here: */

app.get('/home', function(req, res) {
    res.render('pages/home',{
        my_title:'Home Page',
        color: 'FF0000',
        color_msg: 'The Color Red'
    });
});
```

What does this code do?

At the top of the file, you will find the following two lines:

```
var express = require('express'); //Ensure our express framework
has been added
var app = express();
```

The first line tells node that we want to use a module called "express". In the simplest terms, modules are just other javascript files. We "require" the module so that we can use it in this file. The second line instantiates an object called "app". You can think of this as a call to a constructor method that is contained within the module "express".

**app.get** defines a GET request for a specific route, in this case the route is **'/home'**. This is not a relative path.

**res** is the object that represents the HTTP response that our server sends when it receives an HTTP request (like GET). It allows us to render a given page, in this case the page that is located at the relative path **'/pages/home'**.

In our curly braces, we can define any variables we wish to pass onto our website. These variables can then be referenced inside of the view (in our case the EJS files). In this case, we are passing the title of the page in a variable called "**my_title**", a hex color value in a variable called "**color**", etc.

**Step 2.** Load your home page. You will need to first start up the node server. Use the following commands:

```
node server.js
```

To view the homepage, go to localhost:3000/home. For this lab, the server will only be accessible locally so we use "localhost" as our IP address. We also have to specify the port number, 3000, followed by the route "/home".

Hrm .. the color didn't change. That's because we haven't updated our home.ejs file to use the passed variables.

**Step 3.** Update the home.ejs file. Where it says `<!-- TODO: First EJS Update -->`, add the following code:

```
<p> Color Message:
     <% if(color)
       {%>
           <%- color_msg %>
           <% var script_text =
           '<script>document.body.style.backgroundColor = "#' +
           color + '";</script>'; %>
           <%- script_text %>
       <% }%>
</p>
```

To embed our javascript code, we have to use <%%> symbols to let the server know that this block of code should be interpreted as server-side javascript not HTML.

But for <%- color_msg %> we have added a “minus” sign to the opening <%. This means that the HTML code inside this variable should be preserved.

Refresh your browser and see the changes.

**Step 4.** What if we want to pass data from our database to our website? Let’s change our code to handle a simple database request. We will have our home page list a button for each color option in our favorite_colors table.
Let’s start by replacing the GET request in our server.js file with the following code:

```
app.get('/home', function(req, res) {
    var query = 'select * from favorite_colors;';
    db.any(query)
        .then(function (rows) {
            res.render('pages/home',{
                 my_title: "Home Page",
                 data: rows,
                 color: '',
                 color_msg: ''
             })

        })
        .catch(function (err) {
            // display error message in case an error
            request.flash('error', err);
            response.render('pages/home', {
                title: 'Home Page',
                data: '',
                color: '',
                color_msg: ''
            })
        })
});
```

First, we define the query we want to execute against our database. Make sure you place a semicolon INSIDE the single quotes to ensure the query will execute.

The next line is **db.any(query)**. We have defined a database object named “db” at the top of our server.js (similarly to how we defined the app object) file so we can use here. The function “any” means that our query can return ANY number of rows.

The then/catch statements work like a try/catch block in other programming languages. Inside the **.then()** we will execute our code IF our query is successful (no errors), otherwise we will jump over to the **.catch()**.

Here is some magic. Inside our **.then()** we have created a function to hold onto the results from our PostgreSQL query. For this example, we have named the data “**rows**”. We didn’t explicitly assign anything to the variable “**rows**” yet our data is there. We do, however, have to explicitly pass “**rows**” to our front-end in our render statement (i.e., **data: rows**)

**Step 5.** Now we have to change our home.ejs file to make use of the new variables that we are passing along. Add the following code below `<!-- TODO: Second EJS Update -->` :

```
<%
if (data) {
   var buttons = '';
   data.forEach(function(item){
      buttons += '<button type="submit" name="color_selection"
      class="btn btn-block" style="background-color:#' +
      item.hex_value + '" value="' + item.hex_value + '">';
        if(item.name){
         buttons += item.name + '</button>';
      }
      else{
         buttons += item.hex_value + '</button>';
      }
    }); %>
    <%- buttons %>
<% } %>
```

See if you can read the code above and discern what it does. Then, restart the server. You can use ctrl + c to end a running process. Then you can re-run “node server.js” to restart the server. We always need to restart the server to see changes made to server.js, but we do not need to restart the server to see changes made to our EJS files.

The homepage now has a form which contains all of our potential color options but it still can’t change our background color.

**Step 6.** We need to add a new route to our server.js which can process our GET request’s **color_selection** parameter. Add the following code **below app.get(‘/home’ …)**. ADD THIS CODE; DO NOT REPLACE OUR EXISTING app.get(‘/home’ …).

```
app.get('/home/pick_color', function(req, res) {
    var color_choice = req.query.color_selection;
    var color_options =  'select * from favorite_colors;';
    var color_message = "select color_msg from favorite_colors
    where hex_value = '" + color_choice + "';";
    db.task('get-everything', task => {
        return task.batch([
            task.any(color_options),
            task.any(color_message)
        ]);
    })
    .then(info => {
        res.render('pages/home',{
                my_title: "Home Page",
                data: info[0],
                color: color_choice,
                color_msg: info[1][0].color_msg
             })
    })
    .catch(error => {
        // display error message in case an error
            request.flash('error', err);
            response.render('pages/home', {
```

```
                title: 'Home Page',
                data: '',
                color: '',
                color_msg: ''
            })
    });

});
```

Notice how the route that we added (**'/home/pick_color'**) matches the **action="..."** field from our form. This is how the server knows how to handle our form's GET request.

To access our GET request parameters, we can use "**req.query**" followed by the name of the parameter which is "**color_selection**". The **color_selection** parameter matches the name field from the buttons we created on our home page (i.e., **<button name="color_selection">** ).

Next, we define two queries.

Now, we have multiple queries to deal with! To handle them all, we have to use the database's task operation. This operation can handle batch processing and batch process our queries. Inside of **db.task()** we can list as many queries as needed. Just make sure to separate the queries with commas.

When working with multiple queries, we also have to deal with multiple results. The variable **info** is an array with all of the query results. To access an individual query's result we use **info[query_index].**

Since our second query only has 1 value returned, we can actually access it's field directly like this: color_msg: **info[1][0].color_msg**

**Step 7.** The last step to updating our home page is to include a form that lets a user add a new favorite color via a POST request. Add the following code below `<!-- TODO: Third EJS Update -->` :

```
<div class="form-group row">
    <label class="col-sm-2 col-form-label" for="color_hex">Color Hex
    Value:</label>
    <div class="col-sm-10">
```

```
    <input type="text" class="form-control" name="color_hex"
    id="color_hex"  placeholder="Enter Hex Value" maxlength="6">
   </div>
</div>
<div class="form-group row">
   <label class="col-sm-2 col-form-label"  for="color_name">Color
   Name:</label>
   <div class="col-sm-10">
    <input type="text" class="form-control" name="color_name"
    id="color_name" placeholder="Enter Color's Name">
   </div>
</div>
<div class="form-group row">
  <label  class="col-sm-2 col-form-label" for="color_message">Color
  Message:</label>
   <div class="col-sm-10">
     <input type="text" class="form-control" id="color_message"
     name="color_message" placeholder="Enter the Color's
     Description">
   </div>
</div>
<button type="submit" class="btn btn-primary
btn-block">Submit</button>
```

Then, you need to update server.js to add a route for a POST request. You'll be adding an app.post() for the route '/home/pick_color'. This is the same route as for the GET request, but this time we will process the form data and insert the data into the database.

Add the following code to your server.js file:

```
app.post('/home/pick_color', function(req, res) {
    var color_hex = req.body.color_hex;
    var color_name = req.body.color_name;
```

```
    var color_message = req.body.color_message;
    var insert_statement = "INSERT INTO favorite_colors(hex_value,
    name, color_msg) VALUES('" + color_hex + "','" + color_name +
    "','" + color_message +"') ON CONFLICT DO NOTHING;";

    var color_select = 'select * from favorite_colors;';
    db.task('get-everything', task => {
        return task.batch([
            task.any(insert_statement),
            task.any(color_select)
        ]);
    })
    .then(info => {
        res.render('pages/home',{
                my_title: "Home Page",
                data: info[1],
                color: color_hex,
                color_msg: color_message
             })
    })
    .catch(error => {
        // display error message in case an error
            request.flash('error', err);
            response.render('pages/home', {
                title: 'Home Page',
                data: '',
                color: '',
                color_msg: ''
            })
    });
});
```

Much of the syntax here is similar to a GET request. The key difference is in how we access our POST data vs. our GET data. To access our POST data we will need to use our “body-parser” module. This changes our notation from “req.query” (GET) to “**req.body**” (POST).

That’s all for Part 2.

**Part 3: Walk through of GET/POST and ejs**

For this last part, you will finish updating the website to handle GET requests for the football statistics page and the player information page. You will be updating the server.js file as well as the “player_info.ejs” and “team_stats.ejs” files.

**Step 1.** Create a GET request for the football statistics page with the route ‘/team_stats’. This route has no parameters, but will require 3 PostgreSQL queries:

1. Retrieve all of the football games in the Fall 2018 season.
2. Count the number of winning games in the Fall 2018 season.
3. Count the number of losing games in the Fall 2018 season.

The results of the queries will then be passed on to the team_stats view (a.k.a. ‘pages/team_stats’). This view will display all of the football games for the season, show who won each game, and show the total number of wins/loses for the season.

**Step 2.** Create a GET request for the player information page with the route ‘/player_info’. This route has no parameters and will require a single query to retrieve the id & names for all of the football players. It will pass the ids and names to the player_info view (a.k.a. ‘pages/player_info’). The view will use the ids and names to populate the select tag for a form.

**Step 3.** Create a GET request for the player information page with the route ‘/player_info/select_player’. This route has one parameter, the player_id, which corresponds with the football_player’s ids in the database. This route will handle three queries:

1. Retrieve the user ids and names of the football players (just like for route ‘/player_info’)
2. Retrieve the specific football player’s information with id = player_id
3. Retrieve the total number of football games the player (with id = player_id) has played in

The query results will be passed to the player_info view (a.k.a. ‘pages/player_info’). The ids and names will be used to populate the select tag for a form. The view should also display the selected football player’s information from the other two queries.

# What To Turn In

Submit a .zip of all the contents of your website on Canvas.