Web Programming
Lab 3 report

Student full name: Raj Dinesh Jagasia

Student HWU username: rj2011

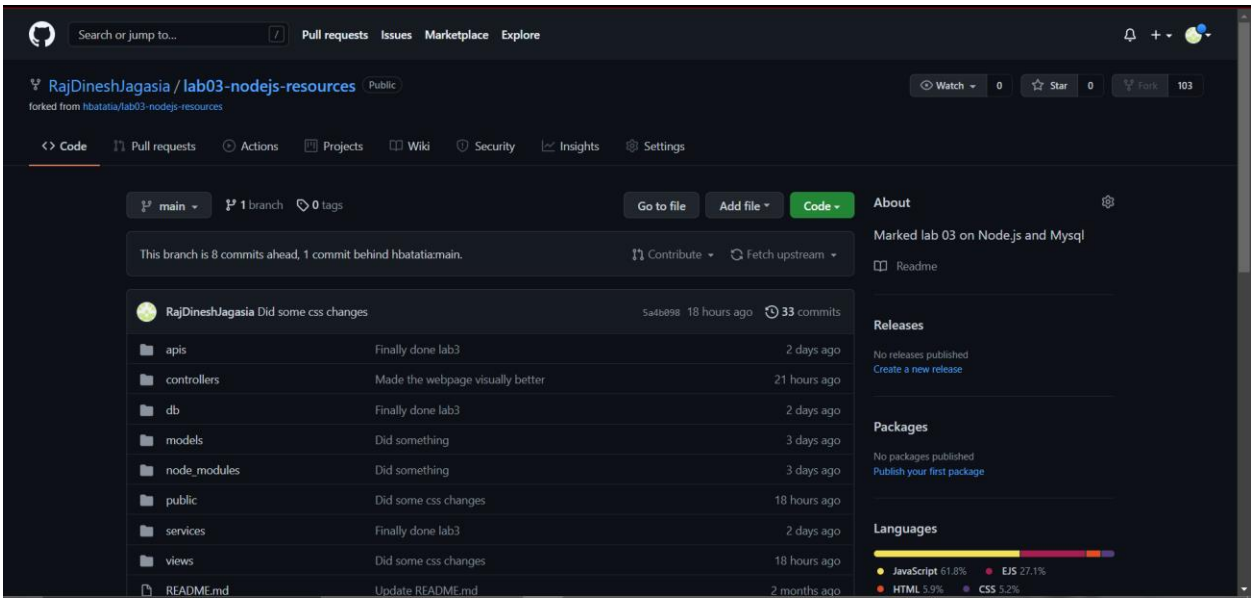
Student's GitHub URL of the Lab: https://github.com/RajDineshJagasia/lab03-nodejs-resources

Demonstrated to Lab helper: _____

Mode of demonstration: Face to Face

Date of demonstration: 30th Nov 2021

Time of demonstration: _____



Part 1: Create web server

Explain the structure of package.json

The package.json file is kind of manifest for my project. It is used to store the metadata related to the project as well as to store the list of dependency packages. The package.json file is used to give information to npm that allows it to identify the project as well as handle all the project's dependencies. In the scripts section there are the scripts which contain the shortcut to run a command. The dependencies section contains the following modules.
Dependencies-> bcryptjs, ejs, express, express session, MySQL and nodemon

Explain how to use npm to install packages.

Npm is a package management tool that is shipped with node.js. It is a command-line utility for interacting with said repository that aids in package installation, version management, and dependency management.
The syntax to install an npm module is → **'npm install <module name>'**
(Stores it in node_module folder, Dependencies defined in package.json)

What are express and nodemon modules?

Express module allows us to create a web server. (**npm install express)**
Nodeman module is a way to automate restaring the server after every change (**npm install nodemon)**

Explain how the HTTP server is created in your app.js

```
const express = require('express');
const app = express();
const port = process.argv[2] || process.env.PORT || 3000;
const server = app.listen(port, () => {
 console.log(`Cart app listening at http://localhost:${port}`);
});
```

We first load the express module and create an app.
HTTP server is created by creating by creating a constant port and defining the port number and server. Then we make the server listen on port 3000.

Part 2: Static web pages

Explain why do we need the static middleware? How does it work? Show example from the code.

In order to avoid adding an app.get(…) statement for each file to send to client, we will use the static module to automatically sends any static file (html, css, images, JavaScript…). Accessing static files are very useful when you want to put your static content accessible to the server for usage. To serve static files such as images, CSS files, and JavaScript files, etc we use the built-in middleware in node.js i.e. express.static. Setting up static middleware: You need to create a folder and add a file.

```
app.use(express.static('public'));
app.set('view engine', 'ejs');
```

```
app.get('/', (req, res) => {
 res.render('index');
});
//pass requests to the router middleware
const router = require('./apis/routes');
app.use(router);
```

Middleware enables a system or application to communicate with each other. It is important because it makes synergy and integration across the applications possible. For the server to pass the HTTP requests to the APIs we need to add the router with static middleware.

Part 3: Using ejs

Explain briefly ejs. Explain the structure of the ejs files (header, footer, index) and their dependence.

EJS (Embedded Javascript) is a template system. You define HTML pages in the EJS syntax and you specify where various data will go in the page. Then, your app combines data with the template and "renders" a complete HTML page where EJS takes your data and inserts it into the web page according to how you've defined the template.

To avoid rewriting the same content multiple times we can split the HTML into different files and use EJS to include them.

header.ejs → contains the HTML content until the content div. It has the opening tags

footer.ejs →contains the footer div. It has the closing tags

index.ejs → contains the header.ejs, contains the content div, and includes footer.ejs

```
<%- include('header'); -%>
 <div class="content"><H1 ID="welcome">WELCOME to Webstore</H1><br>
<%- include('footer'); -%>
```

The include directive is surrounded by specific tag symbols (<%- … -%>).

Show code excerpts from app.js that allow setting ejs and sending the static HTML content from index to the browser.

```
app.get('/contacts', (req, res) => {
    res.render('contacts');
   });
app.get('/catalogue', (req, res) => {
    res.render('catalogue');
   });
```

Explain how your contacts.ejs works.

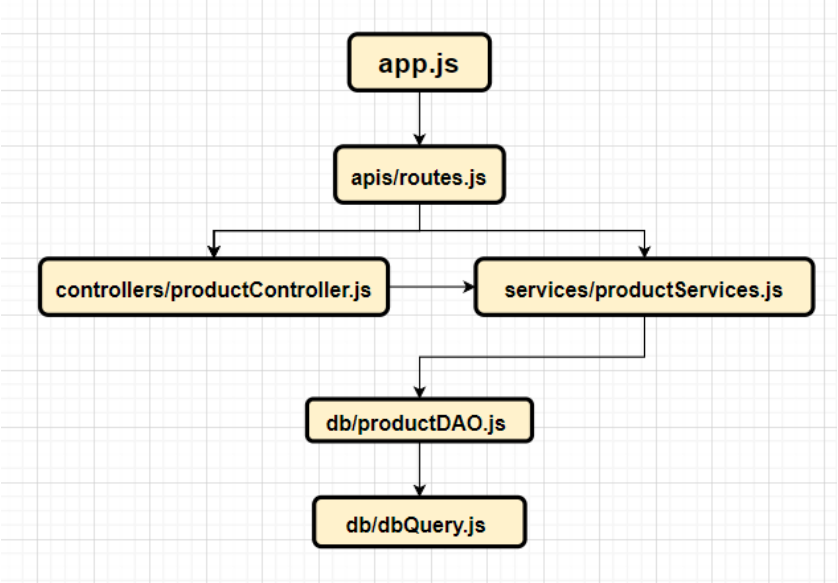
The contacts.ejs contains the name, address and the phone number of the store. It also includes the header and footer of the web page.

We use app.get to render contacts.

Part 4: Displaying the catalogue

Draw a simple diagram to explain the dependence of the following files

- *app.js*
- *routes/apis.js*
- *controllers/productController.js*
- *services/productServices.js*
- *db/productDAO.js*
- *db/dbQuery.js*



Briefly explain this structure and the role of each file

app.js -> It contains all the JavaScript code necessary for the connection of the server, creating the web server and app.get method for rendering the ejs files.

```
const router = require('./apis/routes');
app.use(router);
```

This helps to pass the requests to the router middleware. We *routes/apis.js* in this file

apis/routes.js -> It defines all the routes for which the server provides a response. It uses a router from express. It contains the routes for dynamic processing of products and clients.

```
const router = express.Router();
```

We call *services/productServices.js* and *controllers/productController.js* in routes.js also for displaying catalogue.

```
const catalogServices = require('../services/productServices');
const productController = require('../controllers/productController');
```

controllers/productController.js -> This is responsible for processing HTTP requests related to products. It has getCatalogue, getProductByID and getProductsByCategory in it. The control layer is responsible for handling the HTTP request to extract parameters (possibly check their validity), and sending back the response to the client. I our case, the module productController.js is responsible for processing HTTP requests related to products

```
    const catalogServices = require('../services/productServices');
```

We call *services/productServices.js* in this file.

services/productServices.js -> It is a higher-level service that searches for the products using different criteria.

```
const productDAO = require('../db/productDAO');
```

We call *db/productDAO.js* in this file.

db/productDAO.js -> This implements the queries related to products in the application. This is where we write code in mysql syntax. And then we use database(*db/dbQuery.js)* to get results.

```
const database = require('./dbQuery');
```

We call *db/dbQuery.js* in this file.

db/dbQuery.js -> This is responsible for creating a connection to the database server, submit any SQL query and get the results. These are returned through the callback mechanism.

```
var pool = mysql.createPool({
    connectionLimit: 100,
    host: "localhost",
    user: "root",
    password: "",
    database: "sales",
    debug: true
});
```

What is the text displayed on the browser when the "catalog" request is successful?

Running this application and clicking on Catalogue will bring the list of products as a JSON array

Part 5: Ejs for dynamic content

Explain how catalogue.ejs works

catalogue.ejs is used to display the table of products. In it there is a for loop to iterate through all the products and then grab the name, reference number, category and the price. It is all outputted in the form of a table, each row is a product. In the reference number a hyperlink is added to take it to the details of that article.

Using code excerpts (the function calls), show the flow of execution starting the user click on Catalog (in the web page) and ending with the actual table of products showing on the web pages.

When the user clicks catalog first a get method is called on the api of catalog [1]. Then it goes to routes.js[2] then productController.js [3] and calls the getcatalogue method and in that file it uses the productServices.js file [4] and calls the searchService function from productDAO.js[5]. Then it calls dbQuery.js[6] and then it will go to our database. From there it renders the product as rows and sends the data to the ejs file which finally displays it on the HTML page.

header.ejs

```
[1] → <a href="/api/catalog">Catalog</a>
```

apis/routes.js

```
[2] → router.get('/api/catalog', productController.getCatalogue);
```

controllers/productController.js

```
[3] → const getCatalogue = (request, response) => {
    const catalogServices = require('../services/productServices');
    catalogServices.searchService(function(err, rows) {
    response.render('catalogue', { products: rows });
    });
};
```

services/productServices.js

```
[4] const productDAO = require('../db/productDAO');
const searchService = function(callback) {
    productDAO.findAll(function(err, rows) { ...
```

db/productDAO.js

```
[5] → const database = require('./dbQuery');
function findAll(callback) {
    const selectProducts = "SELECT * from article; ";
    database.getResult(selectProducts, function(err, rows) ...
```

db/dbQuery.js

```
[5] → function getResult(query, callback) {
    executeQuery(query, function(err, rows) {
function executeQuery(query, callback) {
    pool.getConnection(function(err, connection) {
```

pool is attached above

In the apis.js there is get method created to get the catalog file and render it. Then once the button is clicked the ejs file is rendered and the table of products is displayed on the webpage. response.render is used to send the data from the controller to the ejs file.

Explain how you implemented the request for displaying a single article. In particularly, explain how you created article.ejs.

When the client clicks on the reference number in the catalog, a get method is called which calls and renders the article.ejs file. The article.ejs file contains the details of the products in a table format.

```
router.get('/api/article/:id', (req, res) => {
    console.log(req.params.id)
    catalogServices.searchIDService(req.params.id, function(err, rows) {
        res.render('article', { product: rows });
```

Part 6: Login

Explain your implementation of the login feature. List all files required for this functionality and explain their role.

A login.ejs is created with a form with input boxes for the username and password. The clientDAO.js contains the functions for finding username, create account and client. In client services there is a function called loginServices which handles the login request. It first checks for the username if it exists in the database then if it does it checks for the password and matches it with the hash. In clientController.js there is a function called loginControl which controls the login requests. The files used for login are login.ejs, clientController.js and clientDAO.js

I have also added afterLogin and failLogin files. afterLogin is for additional feature and failLogin is when the Login fails, we give the user the option to register.

Explain how the parameters (login and password) are sent by the client and processed by the server.

The login username and password are provided by the client through the input boxes in the login.ejs page.

```
    let username = request.body.username;
    let password = request.body.password;
```

we use request.body.username to get the username and password and then process it by clientServices.js by checking all the parameters and verifying the login. Then a post method is called on the login api to save the details.

Explain the password processing in the server side.

We installed a module bycryptjs for this. The password is processed on the server side by first using bcryptjs to encrypt the password and generate a hash. Then it checks for the similarity of this hash and the hash of the correct password, if it matches then the login is successful.

Show how you used ejs to send the login result to the client.

If the credentials are incorrect or don't match, I will render my failLogin.ejs file.
If they are correct and they match I will render render my afterLogin.ejs file.

clientController.js

```
    if (!username || !password) {
        response.render('failLogin', { username: username });
    } else {
        if (request.session && request.session.user) {
            response.render('afterLogin', { username: username }); ...
```

Part 7: Additional features

Display a single client:

Explain the flow of execution and the files you added to implement the feature that allows displaying the details (all attributes) of a single client giving their id.

I added the below in afterLogin.ejs

```
        <h1><p align= 'center'>
            Hello,
            <a href="/api/login/<%= username %>">
                 <%= username %>
            </a></p>
           <br> Click on your username to view your details.
        </h1>
```

When the user clicks on his username, he will be shown his details which he entered while registering.

For this clientDetails.ejs was created and all the necessary attributes was given in in it so that it can be displayed.

Then in the clientController.js there is a function called get client which gets the information of the client with the given client Id.

In the clientservices.js there is a function which searches the username of the client with the id. Then it displays the data in on the webpage.

clientController.js

```
    clientServices.searchUsernameService(username, function(err, rows) {
        num_client = rows[0].num_client
        clientServices.searchNumclientService(num_client, function(err, rows) {
            console.log(rows[0])
            response.render('clientDetails', {
```

The End