CSCI 3308 Software Development Methods and Tools **Fall 2018**
Instructor: Alan Paradise
TAs: Ajay Kedia, Chelsea Chandler, Michael Schneider, Nikhil Sulegaon and Rohit Mehra
Material: Nikhil Sulegaon
Submission deadline: 4th November 2018 at 11:59pm

Lab 9 - Basic CRUD operations using Node.js

Instructions:-

- In this lab you will be create a full-blown web-app that integrates the frontend with a database using Node.js.
- The webapp we would be creating will look exactly like: https://csci3308-lab9.herokuapp.com/. It would be a good idea for all of you to use this as a reference while solving this lab!
- **Please read and follow all the instructions carefully while you solve this lab.** Most answers are in instructions. So keep an eye out for them.
- Since we will be playing with a lot of file, it is recommended that you use the non-terminal editors. For example - Sublime text, Atom, etc for this lab.

Prerequisites:-

1. PostgreSQL v.10.5

- Make sure you have PostgreSQL installed on your machine. **If not**, run the following commands(**ignore** the '$' symbol) in your terminal to install PostgreSQL.

➔ Ubuntu OS - VM :-

```
$ sudo apt-get update
$ sudo apt-get install -y postgresql postgresql-contrib
$ sudo service postgresql start
```

➔ Mac OS:-

Download and install PostgreSQL v.10.5 using the link: https://www.enterprisedb.com/downloads/postgres-postgresql-downloads.

2. Node.js and NPM (using NVM)

NPM: Node package manager is a tool that helps you to download and install Node.js libraries or packages from the internet!

NVM: Node version manager is an "active" Node.js version manager. You can have multiple versions of Node.js on the same machine and switch between them using nvm.

Installing NVM:

- Run the following commands in your terminal to install **nvm** - Node Version Manager.

➔ Ubuntu OS - VM :-

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential libssl-dev
$ curl -sL
https://raw.githubusercontent.com/creationix/nvm/v0.33.8/install.sh | bash
$ source ~/.profile
```

➔ Mac OS

```
$ touch ~/.bash_profile
$ curl -sL
https://raw.githubusercontent.com/creationix/nvm/v0.33.8/install.sh | bash
$ source ~/.bash_profile
```

Verifying installation of NVM:

Run the following command to check if nvm has been successfully installed!

```
$ nvm --version
```

This command should return the following output to indicate successful installation of nvm:-

```
$ nvm --version
0.33.8
```

Installing Node.js and NPM using NVM:

- Run the following command in your terminal to download and install Node.js (v.8.0) and NPM (v.5.0) using NVM.

```
$ nvm install 8.0.0
$ nvm use 8.0
```

Verifying installation of Node.js and NPM:

Run the following command to check if Node and NPM have been successfully installed!

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

These commands should return the following outputs to indicate successful installation of Node:-

```
$ node -v
v8.0.0
$ npm -v
5.0.0
```

Note*: Although we do not need multiple versions of Node.js to be installed on your machine, we still use NVM to install node because:*

1. *NVM installs both Node and NPM*
2. *NVM also makes sure to install the most compatible version of NPM for the version of Node installed on your machine.*

(Not all NPM versions are compatible with every version of Node)

PART 1: Setting up a Node project.

1. Create a folder on your machine; call it `lab9` and navigate into it.

```
$ mkdir lab9
$ cd lab9
```

2. Run the following command to initialize a Node project using NPM.

```
$ npm init --yes
```

Once you run through the `npm init` above, a `package.json` file will be generated and placed in the current directory.

For more information on what this *`package.json`* file is and what exactly does the *`npm init`* command do refer to the following link: https://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/ (Specifically the "**The Essential npm Commands" section!**)

CHECKPOINT: Your directory structure at this point should look like the following:-

```
lab9
  |-- package.json
```

PART 2: Checking your existing PostgreSQL connection.

1. Make sure you are able to connect to your Postgres database from the terminal. This will ensure that your Node.js app will be able to connect to your PostgreSQL database. Run the following command to do so :-

```
$ psql -U postgres -h localhost
```

This command will bring up a PostgreSQL terminal (different from your BASH terminal) which will look similar to the following; thereby ensuring that your Postgres database is setup!

```
$ psql -U postgres -h localhost
psql (9.4.15)
Type "help" for help.

postgres=#
```

2. Now make sure the database `lab6` exists in PostgreSQL by running the below command inside the PostgreSQL terminal.

```
\c lab6
```

This command will result in the following output if the database exists

```
postgres=# \c lab6
You are now connected to database "lab6" as user "postgres".
lab6=#
```

3. Ensure that a table named `store` exists with some data in it. Run the following command inside the PostgreSQL terminal.

```
SELECT * FROM store;
```

This command will result in the following output if the table exists

```
lab6=# SELECT * FROM store;
 id | sname  | qty | price
----+--------+-----+-------
  5 | orange |  50 |   0.2
  7 | Test   |  12 |     4
  4 | lemon  |  10 |   0.1
  8 | name   |  12 |    14
(4 rows)

lab6=#
```

4. If the database `lab6` **does not** exists, Run the following command in the PostgreSQL terminal to create the `lab6` database and `store` table.

```
CREATE DATABASE lab6;
\c lab6;

CREATE TABLE if not exists store
( id serial,
  sname varchar(40) not null,
  qty integer not null,
  price float not null, primary key (id));

INSERT INTO store (sname, qty, price)
VALUES ('apple', 10, 1), ('pear', 5, 2), ('banana', 10, 1.5), ('lemon',
100, 0.1), ('orange', 50, 0.2);
```

PART 3: Installing the required Node.js packages (libraries).

1. To build a webapp that connects an HTML frontend with your database, we would need to download and install certain Node.js packages(libraries) using NPM. Run the following commands in your BASH **terminal** to do so (***Note:*** *Ignore the '$' symbol while copying the commands*).

```
$ npm install body-parser -save
$ npm install cookie-parser -save
$ npm install method-override -save
$ npm install ejs -save
$ npm install express -save
$ npm install express-flash -save
$ npm install express-session -save
$ npm install express-validator -save
$ npm install pg-promise -save
```

➔ Running the above commands will create a directory called as `node_module` which contains all the packages and libraries. It will also create a file called as `package-lock.json`. The goal of the file is to keep track of the exact version of every package that is installed so that a product is 100% reproducible in the same way even if packages are updated by their maintainers.
➔ For more information on NPM and how it works, do watch the following video: https://www.youtube.com/watch?v=x03fjb2VlGY.

CHECKPOINT: Your directory structure at this point should look like the following:-

```
lab9
  |-- package.json
  |-- package-lock.json
  |-- node_modules
        |-- ...
```

PART 4: Setting up the core of the App.

1. Create a file in your project-directory (`lab9`); call it `server.js`. Copy the following code blocks into the file you just created. (*Note: each code block is followed by a brief description for your reference.*)

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

   ➔ Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications!
   ➔ `require` is a Node.js function that loads packages or libraries.

```
app.set('view engine', 'ejs');
```

   ➔ `ejs` is a templating engine used to template HTML files.
   ➔ HTML requires us to hardcode data into it. However, the data in our database is dynamic,

thus we would need HTML templates using which we generate HTML code.

```
var expressValidator = require('express-validator');
app.use(expressValidator());

var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({extended: true}));
app.use(bodyParser.json());

var methodOverride = require('method-override');
app.use(methodOverride(function (req, res) {
    if (req.body && typeof req.body === 'object' && '_method' in req.body) {
        var method = req.body._method;
        delete req.body._method;
        return method
    }
}));

var flash = require('express-flash');
var cookieParser = require('cookie-parser');
var session = require('express-session');
app.use(cookieParser('csci3308'));
app.use(session({
    secret: 'csci3308',
    resave: false,
    saveUninitialized: true,
    cookie: {maxAge: 60000}
}));
app.use(flash());
```

- ➔ The above is the boilerplate code that is required to set-up url-encoding, sessions, cookies and flash messages for you webapp.
- ➔ Please google for more information on the same or consult your respective TAs.

```
var index = require('./routes/index');
var store = require('./routes/store');
app.use('/', index);
app.use('/store', store);
```

- ➔ Since we want to make our App modular we will create different modules - `index` and `store`. (*Note: these module are created in* ***Part 6*** ***& 8*** *of this write-up*)
- ➔ Each of these module defines certain routes and its handler of our application. For more

information on what routing is, refer to the following link: https://expressjs.com/en/starter/basic-routing.html

```
var port = 4000;
app.listen(port, function () {
    console.log('Server running on http://localhost:' + port)
});
```

➔ This code snippet starts a server on your local machine on port number 4000.

Note: Your `server.js` file after copy pasting all the above code blocks/snippets will look like this gist - https://gist.github.com/nikhilsu/68063a07be8da8188ff85a82e3bf7619. Use this gist to copy-paste code if you run into problems while copying it for this document.

PART 5: Setting up the database connector for the App.

1. Create a file in your project-directory (`lab9`); call it `database.js`. Copy the following code snippets into the file you just created. (*Note: each code block is followed by a brief description for your reference.*)
   a. **Make sure to fill in your PostgreSQL password in the code snippet below.**

```
var pgp = require('pg-promise')();

const dbConfig = {
   host: 'localhost',
   port: 5432,
   database: 'lab6',
   user: 'postgres',
   password: '' // TODO: Fill in your PostgreSQL password here.
                // Use empty string if you did not set a password
};

var db = pgp(dbConfig);

module.exports = db;
```

➔ `pg-promise (pgp)` is a Node.js package/library which allows you to programmatically connect and run queries on a PostgreSQL database! More information about this library in this link: http://vitaly-t.github.io/pg-promise/index.html.

➔ `dbConfig` is an Object (key-value pairs) that houses information of the PostgreSQL database you want to connect to.

Note: Your `database.js` file after copy pasting all the above code blocks/snippets will look like this gist - https://gist.github.com/nikhilsu/ea57a94b943e0ef2857b4dc8960bbcc8. Use this gist to copy-paste code if you run into problems while copying it for this document.

PART 6: Setting up the 'index' route (Home-page of the app).

STEP a: Creating the backend to handle request for home-page

1. Create a **directory** inside your project-directory (`lab9`); call it `routes`. Inside the `routes` directory create a **file** named `index.js`.
2. Copy the following code into the `index.js` file.

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

app.get('/', function (request, response) {
   // render the views/index.ejs template file
   response.render('index', {title: 'Lab 9 - Integration using Node.js'})
});

module.exports = app;
```

- ➔ This code snippet defines the `index (/)` route and renders the index.ejs template when a request is made to the index route. ***Note:*** *The index.ejs template is created in Part 8 of this document.*
- ➔ For more information on the `render` function and HTML templating using `ejs` it is highly recommended that you read through this link: https://www.codementor.io/naeemshaikh27/node-with-express-and-ejs-du107lnk6

STEP b: Creating the frontend HTML template(ejs)

1. Create a **directory** inside your project-directory (`lab9`); call it `views`. Inside the `views` directory create a **file** named `index.ejs`.

2. Copy the following code into the `index.ejs` file.

```
<html>
<head>
    <!-- 'title' is the data that is passed from the index.js during the
response.render function call -->
```

```
    <title><%= title %></title>
</head>
<body>
<div>
    <a href="/">Home</a> |
    <a href="/store/add">Add item</a> |
    <a href="/store">All items</a>
</div>
<h1><%= title %></h1>
</body>
</html>
```

Note: Your `index.js` file after copy pasting all the above code blocks/snippets will look like this gist - https://gist.github.com/nikhilsu/4cc92e830ae09cc4659bdda875521ecc. The `index.ejs` file will look like - https://gist.github.com/nikhilsu/355ca374fdcf9ff84a35d2b8af5ee5c8.

CHECKPOINT: Your directory structure at this point should look like the following:-

```
lab9
  |-- package.json
  |-- package-lock.json
  |-- node_modules
  |     |-- …
  |-- server.js
  |-- database.js
  |-- routes *
  |     |-- index.js *
  |-- views *
        |-- index.ejs *
```

* new files or directories added during this Step.

PART 7: Creating layouts - headers and footers for the HTML templates.

1. Create a **directory** inside the `views` directory; call it `layouts`. Inside this `layouts` directory create a **file** named `header.ejs`.
2. Copy paste the following code snippet into `header.ejs`.

```
<html>
```

```
<head>
   <!-- The param which is passed from render-->
   <title><%= title %></title>
</head>
<body>
<div>
   <a href="/">Home</a> |
   <a href="/store/add">Add item</a> |
    <a href="/store">All items</a>
</div>
<h1><%= title %></h1>
```

- ➔ This file is the **common** HTML template for the **header** of each page of our App.
- ➔ Instead of repeating this piece of code in every HTML page of our App we extract it into partials which can then just be included into different views or pages!

3. Within the `layouts` directory, create a file name `footer.ejs` and copy-paste the following code snippet.

```
 </body>
</html>
```

- ➔ This file is another **common** HTML template for the **footer** of each page. Thus we extract it as a partial as well!

CHECKPOINT: Your directory structure at this point should look like the following:-

```
lab9
   |-- package.json
   |-- package-lock.json
   |-- node_modules
   |     |-- ...
   |-- server.js
   |-- database.js
   |-- routes
   |     |-- index.js
   |-- views
         |-- layouts *
         |     |-- header.ejs *
         |     |-- footer.ejs *
```

```
        |-- index.ejs
```
* new files or directories added during this Step.

PART 8: Setting up the 'store' module for CRUD operations.

Note: *Before you proceed with this part make sure you understand what an HTTP route is - make sure you have read through this blog (from* ***Part 4****).*

PART 8a: Displaying data in the store table on an HTML page.

STEP a: Backend to handle request for displaying data ('/store' route).

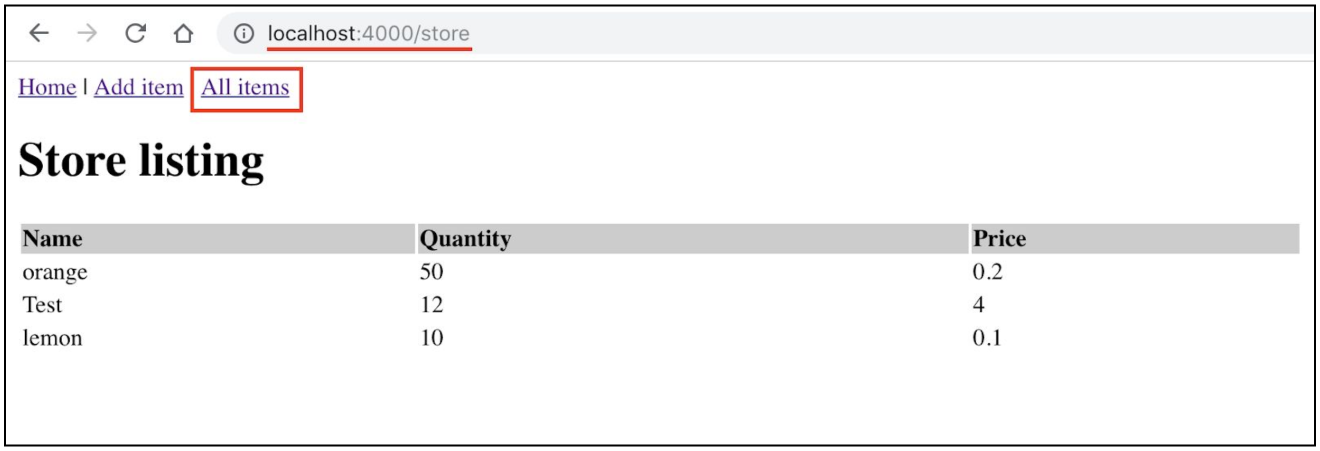
1. Inside the `routes` directory create another **file** named `store.js`. Copy paste all the following code snippets into `store.js` one after the other in the **same order** that they appear! ***NOTE*****:** *Some code snippets below require you to answer questions by completing a part of the code snippet. Look out for these questions!*

```
var express = require('express');
var db = require('../database');
var app = express();
module.exports = app;
```

➔ This code snippet loads/includes the `database` module that we just created in **Part 5**. We will be making use of this module to connect to the database to run SQL queries below.

QUESTION 1 :-

➔ The following code snippet define the function to be executed when a request has been made to the `/store` (http://localhost:4000/store) route.



Home | Add item | All items

Store listing

Name	Quantity	Price
orange	50	0.2
Test	12	4
lemon	10	0.1

➔ The `/store` route should render an HTML page (like the one shown in the screenshot) which lists all the items(rows) in the store(table in your database).

➔ Complete the following code snippet so that you initialize the `query` variable with an SQL query that returns all the items(rows and columns) in the store table.

- ***Note:*** *Do not worry about the frontend, i.e., how the HTML table is rendered at this point. The code snippet below only deals with the backend side of things, i.e, fetching data from the database. Populating the HTML template with data is done in* ***next step*** *of this document.*
- For more information on how to data is fetched from the database using the `pg-promise` package. Refer to this link: https://github.com/vitaly-t/pg-promise/wiki/Learn-by-Example#basics

➔ After you complete the code-snippet by adding the SQL query, copy-paste it into the `store.js` file in the `routes` directory.

```
app.get('/', function (request, response) {

    // TODO: Initialize the query variable with a SQL query
    // that returns all the rows and columns in the 'store' table
    var query = '';

    db.any(query)
      .then(function (rows) {
          // render views/store/list.ejs template file
          response.render('store/list', {
              title: 'Store listing',
              data: rows
          })
      })
      .catch(function (err) {
          // display error message in case an error
```

```
            request.flash('error', err);
            response.render('store/list', {
                title: 'Store listing',
                data: ''
            })
        })
});
```

STEP b: Creating the Frontend HTML template display data fetched above.

1. Inside the `views` directory create another **directory** named `store`.
2. Inside this `store` directory, create a file named `list.ejs` and copy paste the following code snippet. ***Note:*** *Read through the comments in the code to understand what each section is doing.*

```
<!--Including Header Partial-->
<%- include ../layouts/header.ejs %>

<!--Using if-check to see if an error occurred and displaying appropriate
message-->
<% if (messages.error) { %>
    <p style="color:red"><%- messages.error %></p>
<% } %>
<% if (messages.success) { %>
    <p style="color:blue"><%- messages.success %></p>
<% } %>

<script type="text/javascript">
    var err = <%- JSON.stringify(messages) %>;
    console.log(err);
</script>

<table border="0" width='60%'>
    <tr style='text-align:left; background-color:#CCC'>
        <th>Name</th>
        <th>Quantity</th>
        <th>Price</th>
    </tr>
    <% if (data) { %>
    <!-- Using a for-loop to loop over each row in the database.
    The 'data' variable is passed from the store.js during the
response.render function call -->
        <% data.forEach(function(item){ %>
            <tr>
                <td><%= item.sname %></td>
                <td><%= item.qty %></td>
                <td><%= item.price %></td>
            </tr>
        <% }) %>
    <% } %>
</table>

<!--Including Footer Partial-->
<%- include ../layouts/footer.ejs %>
```

Note: Your `routes/store.js` file after copy pasting all the above code blocks/snippets will look like this gist - https://gist.github.com/nikhilsu/9263e46ce1e090b4a9205e973d4f7095. The `views/store/list.ejs` file will look like - https://gist.github.com/nikhilsu/f15741a3bda46211045a6c68b6e6110b.

CHECKPOINT: Your directory structure at this point should look like the following:-

```
lab9
  |-- package.json
  |-- package-lock.json
  |-- node_modules
  |     |-- ...
  |-- server.js
  |-- database.js
  |-- routes
  |     |-- index.js
  |     |-- store.js *
  |-- views
        |-- layouts
        |     |-- header.ejs
        |     |-- footer.ejs
        |-- store *
        |     |-- list.ejs *
        |-- index.ejs
```
* new files or directories added during this Step.

TEST: You can run your code and test your app at this point.

1. Navigate into your project directory(`lab9`) and run the following command in your **terminal**.

```
$ node server.js
```

***Note:* You would need to restart your server by re-running the above command, if you make changes to you code.**

2. Navigate to the URL http://localhost:4000 on your browser to view your homepage.
3. Click the '*List all*' link in the header of the homepage to see all the items in our database!
    - ***Note:** The '**Add item**' link the header is not yet wired-up and hence it wouldn't work as of yet.*

PART 8b: Inserting data into the store table using an HTML form.

STEP a: Creating the Frontend HTML form (template) to take user input.

1. Inside the `views/store` directory, create a file named `add.ejs` and copy paste the following code snippet.

```
<!--Including Header partial-->
<%- include ../layouts/header.ejs %>

<% if (messages.error) { %>
    <p style="color:red"><%- messages.error %></p>
<% } %>
<% if (messages.success) { %>
    <p style="color:green"><%- messages.success %></p>
<% } %>
<!--Create a form to take input from user-->
<form action="/store/add" method="post" name="form1">
    <table width="25%" border="0">
        <tr>
            <td>Name</td>
            <td><input type="text" name="sname" id="name" value="<%= sname
%>"/></td>
        </tr>
        <tr>
```

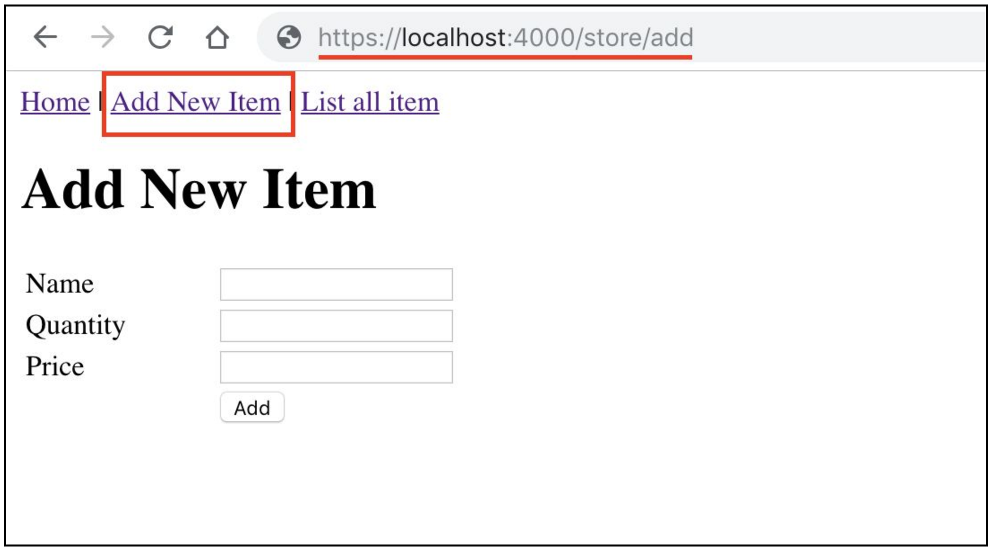
```
            <td>Quantity</td>
            <td><input type="text" name="qty" value="<%= qty %>"/></td>
        </tr>
        <tr>
            <td>Price</td>
            <td><input type="text" name="price" value="<%= price %>"/></td>
        </tr>
        <tr>
            <td></td>
            <td><input type="submit" name="Submit" value="Add"/></td>
        </tr>
    </table>
</form>

<!--Including Footer partial-->
<%- include ../layouts/footer.ejs %>
```

➔ The above HTML template generates a view shown in the screenshot below.

STEP b: Backend to handle request for the HTML form ('/store' route).

1. Now let us define a route that renders a form that takes in information of an item from the user (so that this item can later be added to our store!).


https://localhost:4000/store/add
Home | Add New Item | List all item

Add New Item

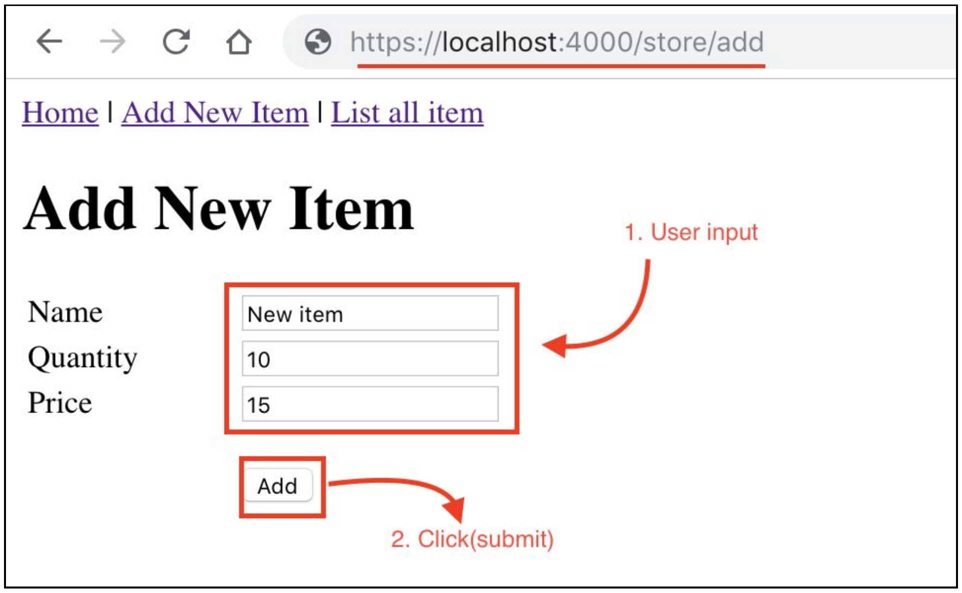
Name	
Quantity	
Price	
	Add

➔ The screenshot above shows the HTML page that should be rendered when a **GET** request is made to the `/store/add` route.
➔ Copy-paste the following code snippet at the end of the `store.js` file inside the `routes` directory.

```
app.get('/add', function (request, response) {
    // render views/store/add.ejs
    response.render('store/add', {
        title: 'Add New Item',
        sname: '',
        qty: '',
        price: ''
    })
});
```

STEP c: Backend to insert data into database AFTER user input.

1. Next we define a route handler which stores the data input by the user into our database.
2. The screenshot below shows the user interaction with the be HTML input form. Note that when the user submits the form by clicking the `Add` button, a **POST** request is made to the `/store/add` route of our Node App.



3. Copy paste the following code snippet at the end of the `route/store.js` file.
   a. ***Note****: Read through the comments in the code snippet below to understand what each section does.*

```
// Route to insert values. Notice that request method is POST here
app.post('/add', function (request, response) {
    // Validate user input - ensure non emptiness
    request.assert('sname', 'sname is required').notEmpty();
    request.assert('qty', 'Quantity is required').notEmpty();
    request.assert('price', 'Price is required').notEmpty();
    var errors = request.validationErrors();
    if (!errors) { // No validation errors
        var item = {
            // sanitize() is a function used to prevent Hackers from inserting
            // malicious code(as data) into our database. There by preventing
            // SQL-injection attacks.
            sname: request.sanitize('sname').escape().trim(),
            qty: request.sanitize('qty').escape().trim(),
            price: request.sanitize('price').escape().trim()
        };
        // Running SQL query to insert data into the store table
        db.none('INSERT INTO store(sname, qty, price) VALUES($1, $2, $3)',
[item.sname, item.qty, item.price])
            .then(function (result) {
                request.flash('success', 'Data added successfully!');
                // render views/store/add.ejs
                response.render('store/add', {
                    title: 'Add New Item',
                    sname: '',
                    qty: '',
                    price: ''
                })
            }).catch(function (err) {
            request.flash('error', err);
            // render views/store/add.ejs
            response.render('store/add', {
                title: 'Add New Item',
                sname: item.sname,
                qty: item.qty,
                price: item.price
            })
        })
    } else {
        var error_msg = errors.reduce((accumulator, current_error) =>
accumulator + '<br />' + current_error.msg, '');
        request.flash('error', error_msg);
        response.render('store/add', {
            title: 'Add New Item',
            sname: request.body.sname,
            qty: request.body.qty,
            price: request.body.price
        })
    }
});
```

Note: Your `routes/store.js` file after copy pasting all the above code blocks/snippets will now look like this gist - https://gist.github.com/nikhilsu/cb62349b75b202e112474f7344cbc2be. The `views/store/add.ejs` file will look like - https://gist.github.com/nikhilsu/10ba9a0b3ccde8cb888c6118391d966a.

CHECKPOINT: Your directory structure at this point should look like the following:-

```
lab9
  |-- package.json
  |-- package-lock.json
  |-- node_modules
  |     |-- …
  |-- server.js
  |-- database.js
  |-- routes
  |     |-- index.js
  |     |-- store.js
  |-- views
        |-- layouts
        |     |-- header.ejs
        |     |-- footer.ejs
        |-- store
        |     |-- list.ejs
        |     |-- add.ejs *
        |-- index.ejs
```

* new files or directories added during this Step.

TEST: You can re-run your code and test the 'Add item' feature at this point.

1. Hit '***Ctrl + c***' to stop the previously running Node server
2. Run the following command in your **terminal again**.

```
$ node server.js
```

Note: **You would need to restart your server by re-running the above command, if you**

make changes to you code.

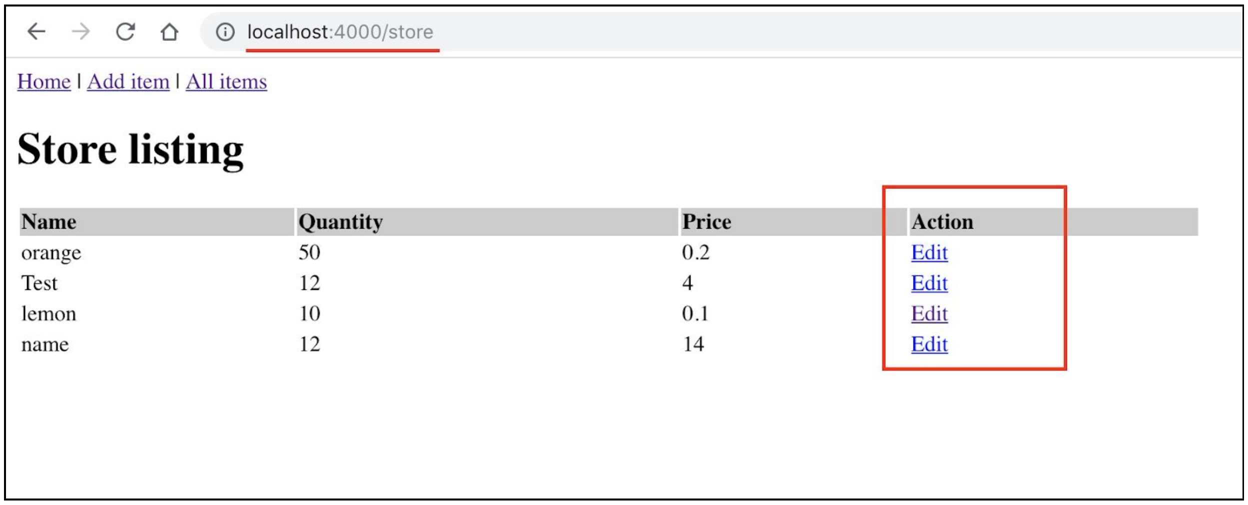
3. Navigate to the URL http://localhost:4000 on your browser to view your homepage.
4. Click the '*Add item*' link in the header. This directs you to the HTML form.
5. Input data and add item. You should see a "Data added successfully!" message appear.

PART 8c: Updating an item in the store table from an HTML page.

Note: Updating an item in our database has 4 steps to it:

STEP a: Creating link to the edit HTML form of each item.

1. In order to edit/update the details of an item we need an edit-form (like the HTML 'Add item' form) specific to each item.



localhost:4000/store

Home | Add item | All items

Store listing

Name	Quantity	Price	Action
orange	50	0.2	Edit
Test	12	4	Edit
lemon	10	0.1	Edit
name	12	14	Edit

2. Thus, we first need to edit the `views/store/list.ejs` file in such a way that it renders a UI like view shown in the screenshot above - create a new column in the HTML table and *'Edit'* links for each item.
3. Paste the code snippets below into the `views/store/list.ejs` file in such a way that a view like the one in the screenshot is rendered. ***NOTE: Paste the below snippets intelligently, don't just append it at the end of the file!!!***

```
<th>Action </th>
```

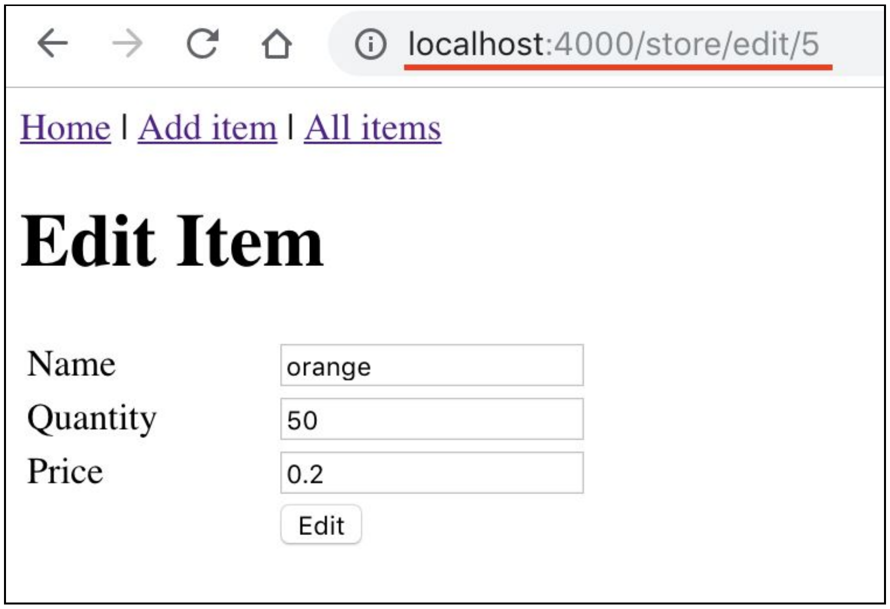
**Hint: This snippet should be a table header!*

```
<td>
    <div class="edit-delete" style="float:left">
        <a href='/store/edit/<%= item.id %>'>Edit</a> &nbsp;
    </div>
</td>
```

Hint: This creates **a link to the edit form of an item. We need one link for each item. Thus, this might within the for-loop!*

STEP b: Creating the update HTML form(template) of each item.

1. Now we need to create an edit/update form to which a user is redirected to when they click the *'Edit'* link that we created in the previous step!
2. This form is similar to the '*Add item*' form except that all textboxes are prefilled with the existing details of the item that we are editing.



3. The form will look like the view in the screenshot above.
4. Create a file named `edit.ejs` **inside** the `views/store` directory and copy-paste the following code-snippet into it.

```
<!-- Include Header Partial -->
<%- include ../layouts/header.ejs %>

<% if (messages.error) { %>
    <p style="color:red"><%- messages.error %></p>
<% } %>
<% if (messages.success) { %>
    <p style="color:green"><%- messages.success %></p>
<% } %>
<!-- The action attribute of the form refers to a item using its database id
-->
<form action="/store/edit/<%= id %>" method="post" name="form1">
    <table width="25%" border="0">
```

```
        <tr>
            <td>Name</td>
            <td><input type="text" name="sname" id="name" value="<%= sname
%>"/></td>
        </tr>
        <tr>
            <td>Quantity</td>
            <td><input type="text" name="qty" value="<%= qty %>"/></td>
        </tr>
        <tr>
            <td>Price</td>
            <td><input type="text" name="price" value="<%= price %>"/></td>
        </tr>
        <tr>
            <td></td>
            <td><input type="submit" name="Submit" value="Edit"/></td>
        </tr>
    </table>
    <input type="hidden" name="_method" value="PUT"/>
</form>

<!-- Include Footer Partial -->
<%- include ../layouts/footer.ejs %>
```

STEP c: Backend to handle request for the update HTML form.

1. Now let us define a route handler that renders an update form specific to an item given its **id** (i.e, the id of the item in the database).
2. Given the database id of an item, the route-handler must render an update form **prefilled** with the details of the item. In order to achieve this, the handler first queries the database for the details of this item by its id.

QUESTION 2 :-

3. Complete the following code snippet so that you initialize the `query` variable with an SQL query that fetches the details of an item given its database `id`.
   - *Examples in this link may come of help here!*
4. After you complete the code-snippet by adding the SQL query, copy-paste it into the end of `routes/store.js` file.

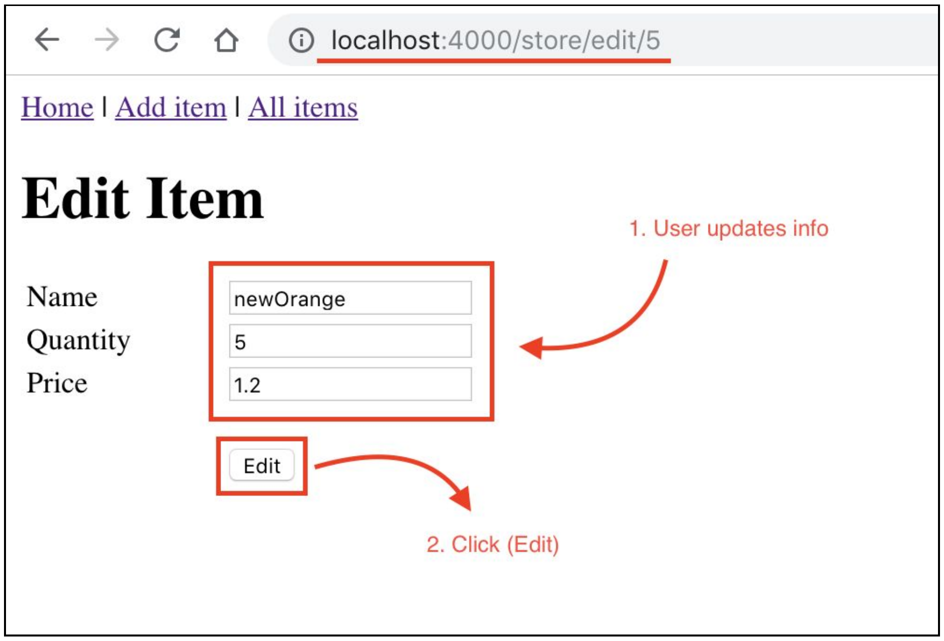
```
app.get('/edit/(:id)', function (request, response) {
    // Fetch the id of the item from the request.
    var itemId = request.params.id;

    // TODO: Initialize the query variable with a SQL query
    // that returns all columns of an item whose id = itemId in the
    // 'store' table
    var query = '';
    db.one(query)
        .then(function (row) {
            // if item not found
            if (row.length === 0) {
                request.flash('error', 'Item not found with id = ' +
request.params.id);
                response.redirect('/store')
            }
            else {
                response.render('store/edit', {
                    title: 'Edit Item',
                    id: row.id,
                    qty: row.qty,
                    price: row.price,
                    sname: row.sname
                })
            }
        })
        .catch(function (err) {
            request.flash('error', err);
            response.render('store/list', {
                title: 'Store listing',
                data: ''
            })
        })
});
```

STEP d: Backend to update item in the database AFTER user input.

1. Next we define a route handler which updates the details of the item in our database after user's (updated) input.
2. The screenshot below shows the user interaction with the be update HTML form. Note that when the user submits the form by clicking the `Edit` button, a **PUT** request is made to the

`/store/edit/{id}` route of our Node App.



QUESTION 3 :-

3. Complete the following code-snippet so that you initialize the [`updateQuery`]() variable with an SQL query that updates the details of an item given its database `id`.
    - *Examples in [this]() link may come of help here!*

4. After you complete the [code-snippet]() by adding the SQL query, copy-paste it into the end of the `route/store.js` file.

    a. ***Note**: Read through the comments in the code snippet below to understand what each section does.*

```
// Route to update values. Notice that request method is PUT here
app.put('/edit/(:id)', function (req, res) {
    // Validate user input - ensure non emptiness
    req.assert('sname', 'Name is required').notEmpty();
    req.assert('qty', 'Quantity is required').notEmpty();
    req.assert('price', 'Price is required').notEmpty();

    var errors = req.validationErrors();
    if (!errors) { // No validation errors
        var item = {
            // sanitize() is a function used to prevent Hackers from inserting
            // malicious code(as data) into our database. There by preventing
            // SQL-injection attacks.
            sname: req.sanitize('sname').escape().trim(),
            qty: req.sanitize('qty').escape().trim(),
            price: req.sanitize('price').escape().trim()
        };

        // Fetch the id of the item from the request.
        var itemId = req.params.id;

        // TODO: Initialize the updateQuery variable with a SQL query
        // that updates the details of an item given its id
        // in the 'store' table
        var updateQuery = “ ”;

        // Running SQL query to insert data into the store table
        db.none(updateQuery)
            .then(function (result) {
                req.flash('success', 'Data updated successfully!');
                res.redirect('/store');
            })
            .catch(function (err) {
                req.flash('error', err);
                res.render('store/edit', {
                    title: 'Edit Item',
                    id: req.params.id,
                    sname: req.body.sname,
                    qty: req.body.qty,
                    price: req.body.price
                })
            })
    }
    else {
        var error_msg = errors.reduce((accumulator, current_error) =>
accumulator + '<br />' + current_error.msg, '');
        req.flash('error', error_msg);
        res.render('store/edit', {
            title: 'Edit Item,',
            id: req.body.id,
            sname: req.body.sname,
            qty: req.body.qty,
            price: req.body.price
        })
    }
});
```

Note: Your `routes/store.js` file after copy pasting all the above code blocks/snippets will now look like this gist - https://gist.github.com/nikhilsu/0c6132ed51ed41c5f5cc7c715cb5c785. For obvious reasons this gist does not contain answers to the questions asked above; you still need to complete the code even if you copy-paste it from the gist.

CHECKPOINT: Your directory structure at this point should look like the following:-

```
lab9
  |-- package.json
  |-- package-lock.json
  |-- node_modules
  |     |-- …
  |-- server.js
  |-- database.js
  |-- routes
  |     |-- index.js
  |     |-- store.js
  |-- views
        |-- layouts
        |     |-- header.ejs
        |     |-- footer.ejs
        |-- store
        |     |-- list.ejs
        |     |-- add.ejs
        |     |-- edit.ejs *
        |-- index.ejs
```

* new files or directories added during this Step.

TEST: You can re-run your code and test the *'Edit'* feature at this point.

1. Hit '***Ctrl + c***' to stop the previously running Node server
2. Run the following command in your **terminal again**.

```
$ node server.js
```

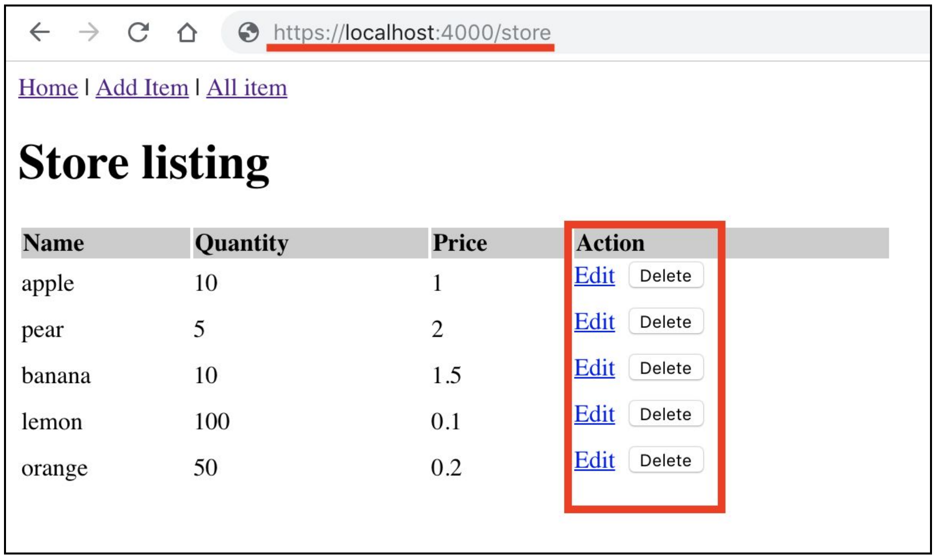
***Note:* You would need to restart your server by re-running the above command, if you make changes to you code.**

3. Navigate to the URL http://localhost:4000/store on your browser to view all items in your database.
4. Click the '*Edit*' link next to an item. This directs you to the item's update form, pre-populated with the current details of the item.
5. Update the details and hit the 'Edit' button. You should see a "Data updated successfully!" message appear.

PART 8d: Deleting an item in the store table from an HTML page.

STEP a: Creating a button to the delete an item.

1. In order to delete a specific item we first need a '*Delete*' button beside each item.



https://localhost:4000/store

Home | Add Item | All item

Store listing

Name	Quantity	Price	Action
apple	10	1	Edit Delete
pear	5	2	Edit Delete
banana	10	1.5	Edit Delete
lemon	100	0.1	Edit Delete
orange	50	0.2	Edit Delete

2. Thus, we first need to edit the `views/store/list.ejs` file in such a way that it renders a UI like view shown in the screenshot above - create a *'Delete'* button for each item.
3. Paste the code snippets below into the `views/store/list.ejs` file in such a way that a view like the one in the screenshot is rendered. ***NOTE: Paste the below snippets intelligently, don't just append it at the end of the file!!!***

```
<form action="/store/delete/<%= item.id %>" method="post" style="float:right">
    <input name="delete" onclick="return confirm('Are you sure you ' +
                                                 'want to delete?')"

           type="submit" value='Delete'>
    <input name="_method" type="hidden" value="DELETE">
```

```
</form>
```

**Hint: The above snippet renders a single button. Paste it so that it appears after the 'Edit' link (<a> tag).*

STEP b: Create backend to handle request to delete an item from DB.

1. Here we define a route handler which deletes an item from our database given its database `id`.

QUESTION 4 :-

2. Complete the following code-snippet so that you initialize the `deleteQuery` variable with an SQL query that deletes an item from the database given the item's database `id`.
3. After you complete the code-snippet by adding the SQL query, copy-paste it into the end of the `route/store.js` file.
    a. ***Note***: *Read through the comments in the code snippet below to understand what each section does.*

```
// Route to delete an item. Notice that request method is DELETE here
app.delete('/delete/(:id)', function (req, res) {
    // Fetch item id of the item to be deleted from the request.
    var itemId = req.params.id;

    // TODO: Initialize the deleteQuery variable with a SQL query
    // that deletes an item whose id = itemId in the
    // 'store' table
    var deleteQuery = '';
    db.none(deleteQuery)
        .then(function (result) {
                  req.flash('success', 'successfully deleted it');
                  res.redirect('/store');
        })
        .catch(function (err) {
                   req.flash('error', err);
                   res.redirect('/store')
        })
});
```

Note: Your `routes/store.js` file after copy pasting all the above code blocks/snippets will now look like this gist - https://gist.github.com/nikhilsu/a810c64806502f405845b02da60f3a62. For obvious reasons this gist does not contain answers to the questions asked above; you still need to complete the code even if you copy-paste it from the gist.

Credit: To get credit for this lab exercise, zip all the files in the project directory `lab9` and submit the same on to moodle. If you paired with someone in this lab, make sure to include your partner's name in the `server.js` file.