COMP3322A Modern Technologies on World Wide Web

Lab 6: RESTful Web Service Using Express.js, MongoDB and Pug

Introduction

In this lab exercise, we will use Node.js to implement a RESTful Web service and Pug to generate the HTML content for accessing the Web service. In particular, we will use the Express.js web framework based on Node.js, together with the Pug template engine and MongoDB. The Web service allows retrieving, adding, updating and deleting contacts from a MongoDB database. The HTML page provides an interface for displaying contact information, adding, updating and deleting contacts.

Set Up Runtime Environment and MongoDB

Follow the instructions (Steps 1 to 3) in **setup_nodejs_runtime_and_examples_1.docx** to set up Node.js runtime environment and create an Express project named "**lab6**".

We will need a MongoDB database to store contacts information. We create the database and install dependency modules as follows.

Step 1: In the "lab6" project directory, create a new directory "data". This directory will be used to store database files.

```
cd lab6
mkdir data
```

Step 2: **Launch the 2nd terminal** (besides the one you use for running NPM commands), and switch to the directory where MongoDB is installed. Start MongoDB server using the "data" directory of "lab6" project as the database location, as follows: (replace "**YourPath**" by the actual path on your computer that leads to "lab6" directory)

```
./bin/mongod --dbpath YourPath/lab6/data
```

After starting the database server successfully, you should see some prompt in the terminal like "…2019-11-02T11:14:10.896+0800 I NETWORK [initandlisten] waiting for connections on port 27017". This means that the database server is up running now and listening on the default port 27017. **Then leave this terminal open and do not close it during your entire lab practice session,** in order to allow connections to the database from your Express app.

Step 3: **Launch the 3rd terminal**, switch to the directory where mongodb is installed, and execute the following commands:

```
./bin/mongo
use lab6
db.contactList.insert({'name':'Jim', 'email':'jim@gmail.com', 'tel':'1234567'})
```

The "use lab6" command creates a database named "lab6". The next command followed by "use lab6" inserts a new record into the "contactList" collection of the database.You can insert more records into the database collection to facilitate testing of your program.

Step 4: Now switch back to the terminal where you run NPM commands. Go to the "lab6" project directory, and install the dependencies asfollows:

```
cd lab6
npm install
```

In addition, install Monk and Pug template engine as follows in the project directory:

```
npm install monk
npm install pug
```

Rename the suffix of all template files in the "./views" folder from "jade" to "pug".

Step 5: Now open the generated **app.js** with an editor and check out its content. Replace app.set('view engine', 'jade'); in **app.js** by app.set('view engine', 'pug');.

You can see a number of middlewares have been included to handle the incoming requests, among which we will need express.json(),express.urlencoded(), and express.static() in this lab, and you can remove the code related to cookieParser.

This line of code "module.exports = app;" at the end of **app.js** exports this *app* as the default module that the Express app will run once started; then you can start the Express app in the terminal by typing the following command in the "lab6" directory:

```
npm start
```

After running "npm start", the web server is started and listens at the default port 3000. You can launch a web browser and visit the web page at http://l27.0.0.1:3000 or http://localhost:3000.

Lab Exercise 1: Create the Web Page Using Pug

We next modify the Pug templates in the "./views" directory of "lab6", in order to render the homepage of our Express app.

Step 1: Replace **index.pug** with the index.pug which we provide in **lab6_materials.zip**. Open it using a text editor. Please refer to https://pugjs.org/api/getting-started.html to understand the code in the file.

Step 2: Open **layout.pug** using a text editor and modify it to contain the following content:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
    script(src='https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js')
    script(src='/javascripts/externalJS.js')
```

The first line of code in **index.pug** indicates that **index.pug** extends **layout.pug**. By modifying **layout.pug** as above, the web page rendered links to:

(1) a **style.css** file under ./public/stylesheets for styling (use the style.css file we provide to you in **lab6_materials.zip** to replace the default style.css file under ./public/stylesheets);

(2) the jQuery library on Google server;

(3) an **externalJS.js** file under ./public/javascripts containing client-side JavaScript (which we will create under that directory in Lab Exercise 2).

Note that the ./public directory has been declared to hold static files which can be directly retrieved by a client browser, using the line of code "app.use(express.static(path.join(__dirname, 'public')));" in **app.js** (note that there are two underscores "_" before dirname in the code). In this way, the rendered web page can directly load files under the ./public directory.

Step 3: Open **index.js** under the directory ./routes, and replace "Express" in the line "res.render('index', { title: 'Express' });" by "Lab 6".

Step 4: Now let's check out the web page rendered using the new Pug files. In the terminal, type "**npm start**" to start the Express app (**you should always use control+C to kill an already running app before you start the app again after making modifications**). Check out the rendered page again at http://localhost:3000 on your browser. You should see a page like Fig. 1. Especially, your GET request for "/" is handled on the server side by the router **index.js**, which renders the HTML page using **index.pug** and **layout.pug**. Further, on the terminal where you have run "npm start", you can see prompts like the following, showing that the browser has issued two more GET requests (after the GET request for "/") to the server side to retrieve the client-side javascript file and styling sheet file, respectively.

```
GET / 200 1132.058 ms - 1463
GET /javascripts/externalJS.js 404 48.687 ms - 1491
GET /stylesheets/style.css 200 2.237 ms – 2906
```

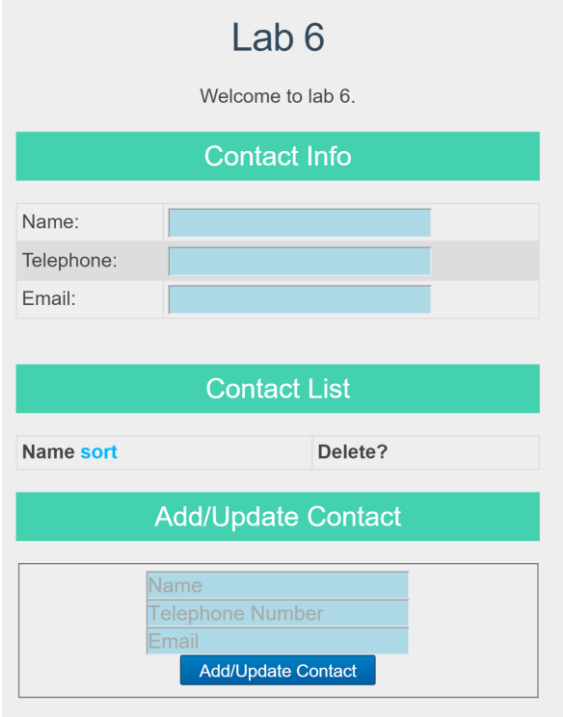

Fig. 1

Lab Exercise 2: List Contacts

We next modify our Express app to connect to the database, retrieve and display the contact list.

Step 1: Open **app.js** and add the following lines ***below*** "var logger = require('morgan');". By doing so, we establish a connection with the database "lab6" that we created.

```
// Database
var monk = require('monk');
var db = monk('localhost:27017/lab6');
```

Then we need to enable subsequent router modules to access the database. To achieve this, add the following code ***before*** the line of "app.use('/', indexRouter);".

```
// Make our db accessible to routers
app.use(function(req,res,next){
   req.db = db;
   next();
});
```

By assigning the **db** object to **req.db**, subsequent router modules can use **req.db** to communicate with the database.

Step 2: Now open **users.js** in the directory ./routes and modify the file such that it contains the following content:

```
var express = require('express');
var router = express.Router();

/*
* GET contactList.
*/
```

```
router.get('/contactList', function(req, res) {
        var db = req.db;
        var collection = db.get('contactList');
        collection.find({},{},function(err,docs){
                if (err === null)
                        res.json(docs);
                else res.send({msg: err});
        });
});

module.exports = router;
```

The middleware in this **users.js** controls how the server responds to the HTTP GET requests for "http://localhost:3000/users/contactList". The middleware will first retrieve the database connection. Then it will retrieve the 'contactList' collection, encode everything in this collection as a JSON message and send it back to the client.

Step 3: Restart your Express app with "npm start" in your first terminal. Test if your server-side code works by browsing http://localhost:3000/users/contactList on your browser. The browser should display a JSON response text like this:

[{"_id":"582e741ac1f51204644fb50e","name":"Jim","email":"jim@gmail.com","tel":"1234567"}]

The "_id" attribute was added by the database server into each contact document that we inserted earlier, which is used to uniquely identify the record in a collection. When a contact record is retrieved from the database, this "_id" attribute and its value are also included.

Step 4: Now we add client-side code for displaying the contact list. Recall that in Step 2 of Lab Exercise 1, we link the rendered HTML page to **externalJS.js**. Create an **externalJS.js** file under the directory ./public/javascripts. Put the following jQuery code into **externalJS.js**:

```
// contact data array for filling in info box
var contactListData = [];

// disable the contact info
$("#contactInfoName").prop("disabled", true);
$("#contactInfoTel").prop("disabled", true);
$("#contactInfoEmail").prop("disabled", true);

// DOM Ready =============================================================
$(document).ready(function() {

   // Populate the contact list on initial page load
   populateContactList();

});

// Functions =============================================================

// Fill contact list with actual data.
function populateContactList() {
```

```
  // Empty content string
  var tableContent = '';

  // jQuery AJAX call for JSON
  $.getJSON( '/users/contactList', function( data ) {
    contactListData = data;

    // For each item in our JSON, add a table row and cells to the content string
    $.each(data, function(){
      tableContent += '<tr>';
      tableContent += '<td><a href="#" class="linkShowContact" rel="' + this.name + '">' +
this.name + '</a></td>';
      tableContent += '<td><a href="#" class="linkDeleteContact" rel="' + this._id +
'">delete</a></td>';
      tableContent += '</tr>';
    });

    // Inject the whole content string into our existing HTML table
    $('#contactList table tbody').html(tableContent);
  });
  };
```

Step 5: Now browse the home page at http://localhost:3000/. The request is handled by the middleware in router **index.js**, which renders the web page using **index.pug** and **layout.pug**. The rendered page links to **externalJS.js**. The jQuery code in **externalJS.js** is executed when the page has been loaded by the browser ($(document).ready()), which adds retrieved document(s) into the contact list. You should see that the contact record that we inserted into the database earlier is now displayed on the web page:

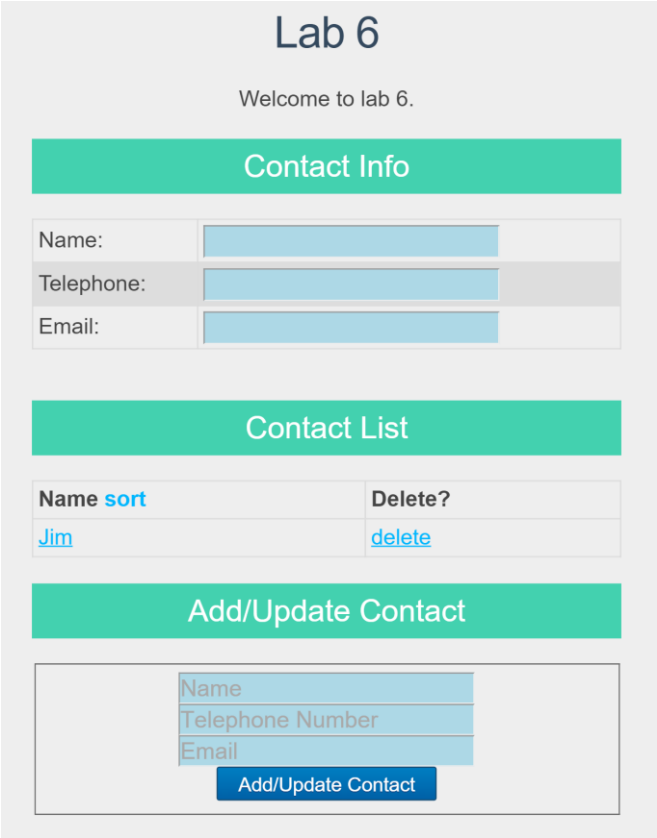

Fig. 2

Lab Exercise 3: Show Detailed Contact Information

We next implement the client-side code for displaying a contact's detailed information in the "Contact Info" part of the page, when a contact name in the contact list is clicked.

Step 1: In the code we added into **externalJS.js** in Step 4 of Lab Exercise 2, we saved contact record(s) retrieved from the database into an array ***contactListData*.** We now retrieve the respective contact's record from array **contactListData** and display the detailed information. Open **externalJS.js** and add the following content at the end of the file.

```
// Show Contact Info
function showContactInfo(event) {

   // Prevent Link from Firing
   event.preventDefault();

   // Retrieve contact name from link rel attribute
   var thisContactName = $(this).attr('rel');

   // Get Index of object based on name
   var arrayPosition = contactListData.map(function(arrayItem) { return
arrayItem.name; }).indexOf(thisContactName);

   // Get our contact Object
   var thisContactObject = contactListData[arrayPosition];

   //Populate Info Box
   $('#contactInfoName').val(thisContactObject.name);
   $('#contactInfoTel').val(thisContactObject.tel);
   $('#contactInfoEmail').val(thisContactObject.email);

   //record id in "rel" attribute of the contactInfoName field
            $('#contactInfoName').attr({'rel': thisContactObject._id});

            //enable the contact info
            $("#contactInfoName").prop("disabled", false);
            $("#contactInfoTel").prop("disabled", false);
            $("#contactInfoEmail").prop("disabled", false);
};

 // handle contact name link click
$('#contactList table tbody').on('click', 'td a.linkShowContact', showContactInfo);
```

Step 2: Browse http://localhost:3000/ in your browser: refresh the page and click **Jim** in the Contact List. You should see that the detailed information of the contact is displayed in the Contact Info part:

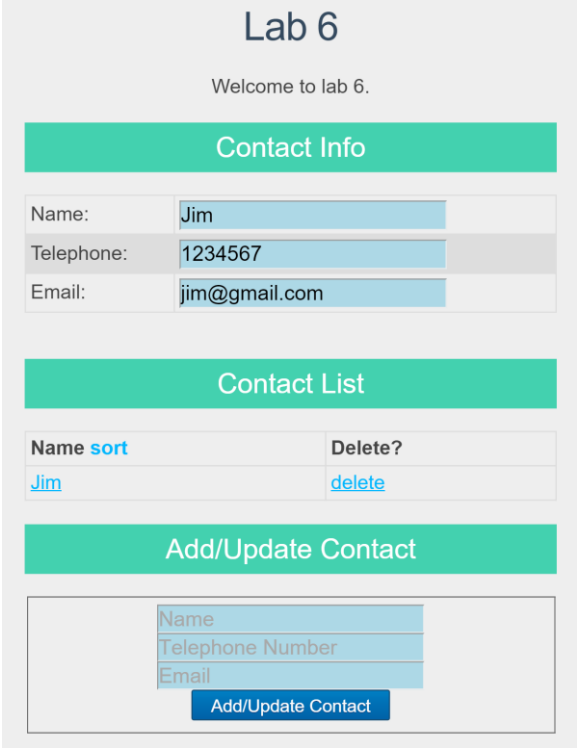

Fig. 3

Lab Exercise 4: Add a New Contact

We next implement the server-side and client-side code for adding a new contact record into the database.

Step 1: Open **users.js** in the ./routes directory and add the following middleware into this file, which handles HTTP POST requests sent for http://localhost:3000/users/addContact .

```
/*
 * POST to addContact.
 */
router.post('/addContact', function(req, res) {
    var db = req.db;
    var collection = db.get('contactList');
    collection.insert(req.body, function(err, result){
        res.send(
            (err === null) ? { msg: '' } : { msg: err }
        );
    });
});
```

Make sure names of the contacts you add into the database are different, as we are going to make use of the name to distinguish a new or existing contact in the "Add/Update Contact" part.

Step 2: Open **externalJS.js** in the ./public/javascripts directory and add the following code at the end of the file. What the code achieves is as follows: when the "Add/Update Contact" button is clicked, the **addOrUpdateContact** function will be invoked. **addOrUpdateContact** first checks if all fields in the "**#addOrUpdateContact**" division have been filled: if not, it prompts 'Please fill in all fields' and return; otherwise, it further checks if the entered name exists among names of all contacts displayed on the page. If the name does not exist, it sends an AJAX HTTP POST request to http://localhost:3000/users/addContact, carrying a JSON string

containing the input information of the new contact inside its body. Upon receiving a success HTTP response, the client clears all the fields in the "**#addOrUpdateContact**" division, and updates the contact list by calling **populateContactList ()**.

```
// Add or update contact
function addOrUpdateContact(event) {
  event.preventDefault();

  // Super basic validation - increase errorCount variable if any fields are blank
  var errorCount = 0;
  $('#addOrUpdateContact input').each(function(index, val) {
    if($(this).val() === '') { errorCount++; }
  });

  // Check and make sure errorCount's still at zero
  if(errorCount === 0) {
    // check if the input contact exists already
    var name = $('#addOrUpdateContact fieldset input#inputContactName').val();
    var existContactIndex = -1;
    for (var i = 0; i < contactListData.length; i++){
      if(contactListData[i].name == name){
        existContactIndex = i;
        break;
      }
    }

    if(existContactIndex >= 0){
      //the contact exists
      var existingContact = {
        '_id': contactListData[existContactIndex]._id,
        'name': $('#addOrUpdateContact fieldset input#inputContactName').val(),
        'tel': $('#addOrUpdateContact fieldset input#inputContactTel').val(),
        'email': $('#addOrUpdateContact fieldset input#inputContactEmail').val()
      }
      updateContact(existingContact);
    } else{
      // the contact is new
      var newContact = {
        'name': $('#addOrUpdateContact fieldset input#inputContactName').val(),
        'tel': $('#addOrUpdateContact fieldset input#inputContactTel').val(),
        'email': $('#addOrUpdateContact fieldset input#inputContactEmail').val()
      }
      // Use AJAX to post the object to our addContact service
      $.ajax({
        type: 'POST',
        data: newContact,
        url: '/users/addContact',
        dataType: 'JSON'
      }).done(function( response ) {
        // Check for successful (blank) response
        if (response.msg === '') {
          // Clear the form inputs
          $('#addOrUpdateContact fieldset input').val('');
          // Update the table
          populateContactList();
        }
```

```
            else {
                // If something goes wrong, alert the error message that our service returned
                alert('Error: ' + response.msg);
            }
        });
    }
  }
  else {
    // If errorCount is more than 0, error out
    alert('Please fill in all fields');
    return false;
  }
};

// Add/Update Contact button click
$('#btnAddUpdateContact').on('click', addOrUpdateContact);
```

Step 3: Restart your Express app and browse http://localhost:3000 again. Add information of a new contact as Fig. 4 below. After clicking the "Add/Update Contact" button, you should see a page as shown in Fig. 5.

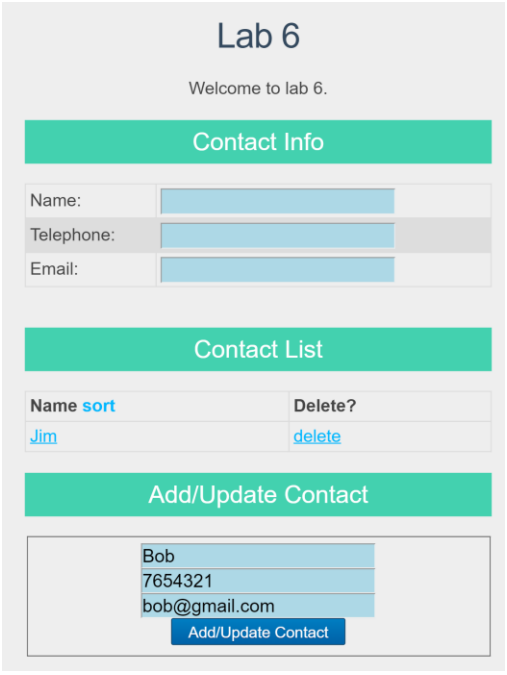

Fig. 4

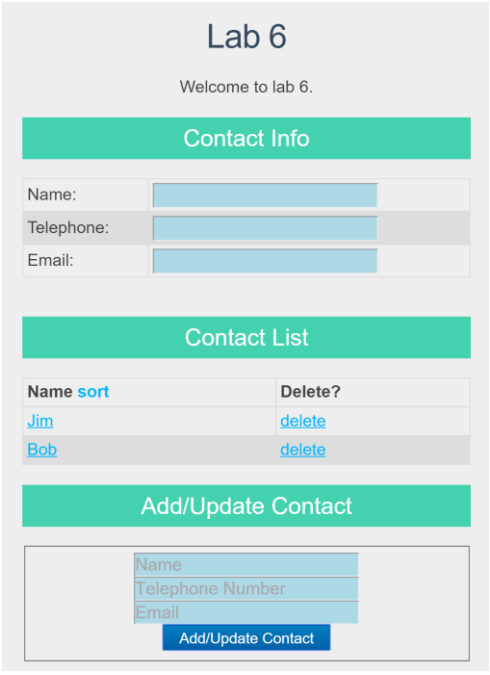

Fig. 5

Lab Exercise 5: Update a Contact

In this part, we implement the server-side and client-side code for updating an existing contact record in the database.

Step 1: Open **users.js** in the ./routes directory and add the following middleware:

```
/*
* PUT to updateContact
```

```
*/
router.put('/updateContact/:id', function (req, res) {
 var db = req.db;
 var collection = db.get('contactList');
 var contactToUpdate = req.params.id;

//TO DO: update the contact record in contactList collection, according to
contactToUpdate and data included in the body of the HTTP request

});
```

Implement the code in the above middleware, for updating an existing contact record in the contactList collection, whose "_id" is carried in the URL of the PUT request, and new contact information carried in request body. **Hint**: use **collection.update()**. Upon successful update, the server should send a response message back to the client with an empty body; otherwise, it sends the error message back to the client.

Step 2: Open **externalJS.js** in the ./public/javascripts/ directory and add the following code at the end of the file to update the contact record when an existing contact record is updated.

```
// Update Contact
function updateContact(existingContact) {
   var id = existingContact._id;

   $.ajax({
      type: ?,
      url: ?,
      data: ?,
      dataType: 'JSON'
   }).done(function (response) {
      if (response.msg === '') {
         // Clear the form inputs
         ?

        // Update the table
        populateContactList();

         //Update detailed info if the updated contact's detailed info is displayed
         ?
      }
      else {
          ?
      }
   });
};
```

Replace "?" with correct code to finish the client-side code for sending an AJAX HTTP PUT request and handling the response. Upon successful update, you should clear the form input on the web page; and if the detailed contact information of the updated contact is being displayed under "Contact Info", update the contact info there. If an error message is carried

in the response, prompt the error message using alert().

Step 3: Restart your Express app, browse http://localhost:3000 again, and test the update function as follows:

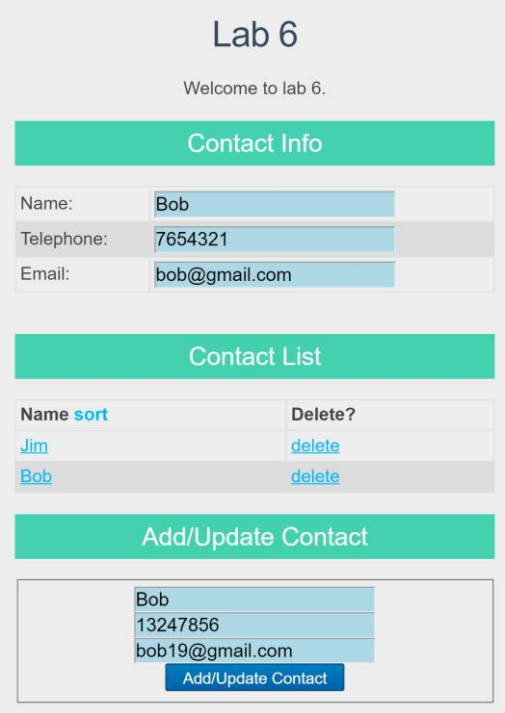

Fig. 6 Input new information of an existing contact

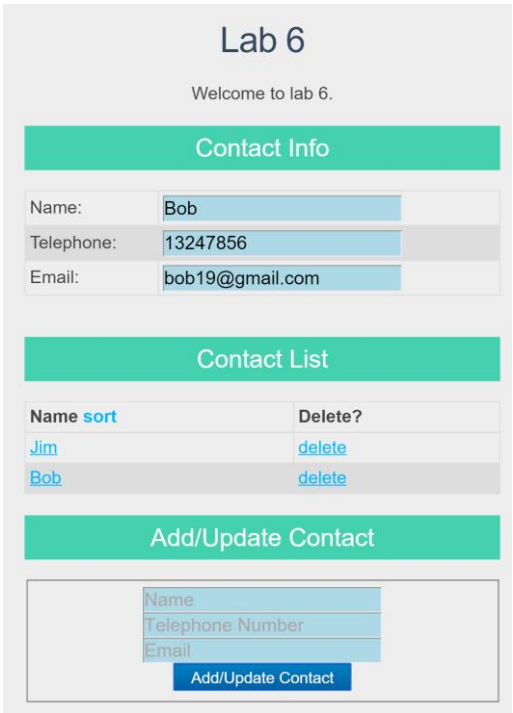

Fig. 7 After clicking "Add/Update Contact" on the Fig. 6 view

Lab Exercise 6: Delete a Contact

In this part, we implement the server-side and client-side code for deleting a contact record from the database, when a respective "delete" button in the contact list is clicked.

Step 1: Open **users.js** in the ./routes directory and add the following middleware:

```
/*
 * DELETE to delete a contact.
 */
router.delete(?, function(req, res) {
   ?
});
```

You should replace "?" with correct code for handling a delete request, by following the hints below:

1. Among the code we added in Step 4 of Lab Exercise 2, the **"_id"** attribute of a contact record is saved to the **"rel"** attribute of a **"<a>"** element of class **"linkDeleteContact"**, i.e., the "**delete**" link shown in the screenshots. The client will send an AJAX HTTP DELETE request to the following URL once you click the "**delete"** button:

http://localhost:3000/users/deleteContact/xx
(replace xx by the value of "_id" attribute of a contact record to be deleted).

2. The middleware should handle HTTP DELETE requests for path **/deleteContact/:id,** and retrieve the **"_id"** attribute carried in a DELETE request through **req.params.id**.

3. Use **remove()** method on a MongoDB collection for deleting the respective contact record from the collection in the database. Upon successful deletion, the server should send an empty response message back to the client; otherwise, it sends the error message back to the client.

Step 2: Open **externalJS.js** in the ./public/javascripts/ directory and add the following code at the end of the file.

```
// Delete contact link click
$('#contactList table tbody').on('click', 'td a.linkDeleteContact', deleteContact);

// Delete Contact
function deleteContact(event) {
  event.preventDefault();

  // Pop up a confirmation dialog
  var confirmation = confirm('Are you sure you want to delete this contact?');

  // Check and make sure the contact confirmed
  if (confirmation === true) {
    // If confirmed, do our delete
    var id = $(this).attr('rel');
    $.ajax({
      type: ?,
      url: ?
    }).done(function( response ) {
      ?
    });
  }
  else {
    // If saying no to the confirm, do nothing
    return false;
  }
};
```

Replace "?" with correct code to finish the client-side code for sending an AJAX HTTP DELETE request and handling the response. You should follow these hints:

1. **$('#contactList table tbody').on('click', 'td a.linkDeleteContact', deleteContact);** captures the click event on the **delete** link. This event will be processed by the **deleteContact** function.

2. You should fill in correct type and url of the HTTP DELETE request in the **$.ajax** method call.

3. Upon successful deletion, you should refresh the "Contact List" display on the web page. If the detailed information of the contact deleted is displayed in the "Contact Info" part, empty the detailed information there. If the deletion failed on the server side, display the error message carried in the response using alert().

Step 3: Restart your Express app, browse http://localhost:3000 again, and test the delete function as follows:

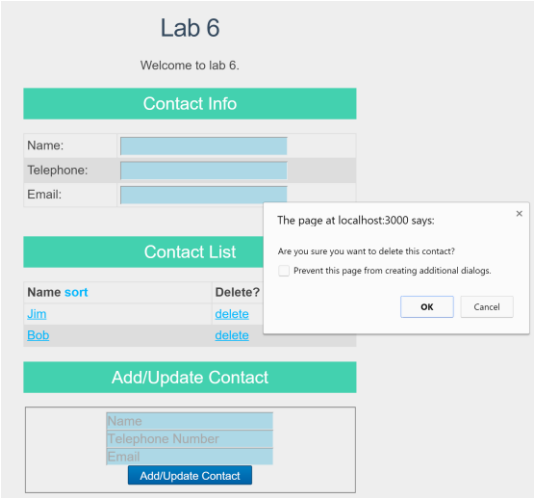

Fig. 8 After clicking "delete" in the row of "Jim" (when Jim's detailed info is not displayed in "Contact Info")

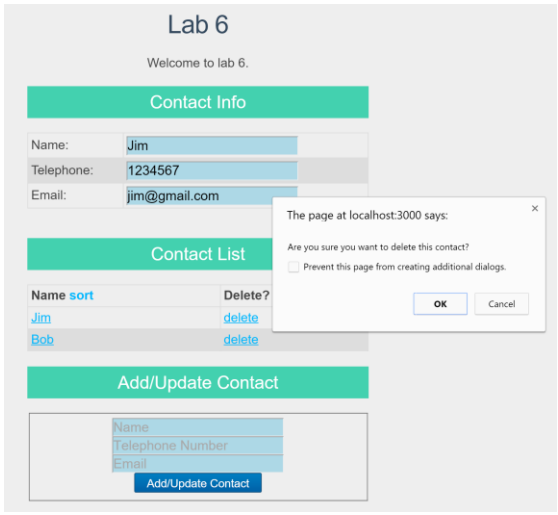

Fig.9 After clicking "delete" in the row of "Jim" (when Jim's detailed info is displayed in "Contact Info")

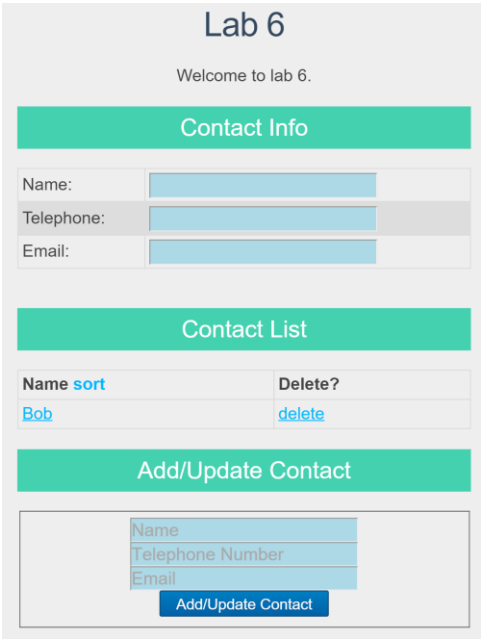

Fig. 10 After confirming deletion by clicking "OK" in both case (1) and case (2) above

Lab Exercise 7: Sort the Contacts

In this part, we implement the client-side code for sorting contacts displayed in the contact list, when the "sort" in the contact list is clicked. Note that the "sort" is a link with click event handler "sortList()", as we have "sort" in **index.pug**. When the "sort" link is clicked for the first time, the contacts should be sorted according to alphabetic order of their names; when "sort" is clicked again, the contacts should be listed in reverse alphabetical order of their names; and the list shuffles between alphabetic order and reverse alphabetical order when "sort" is clicked again and again.

Add and implement the following function in **externalJS.js** :

```
function sortList() {
  //to complete
}
```

Hints: you can directly sort the previously retrieved contacts array, using the Array sort method https://www.w3schools.com/jsref/jsref_sort.asp. You need a global variable to

remember whether you should sort the list into alphabetical order or reserve alphabetical order each time.

Restart your Express app, browse http://localhost:3000 again, and test the sort function as follows:

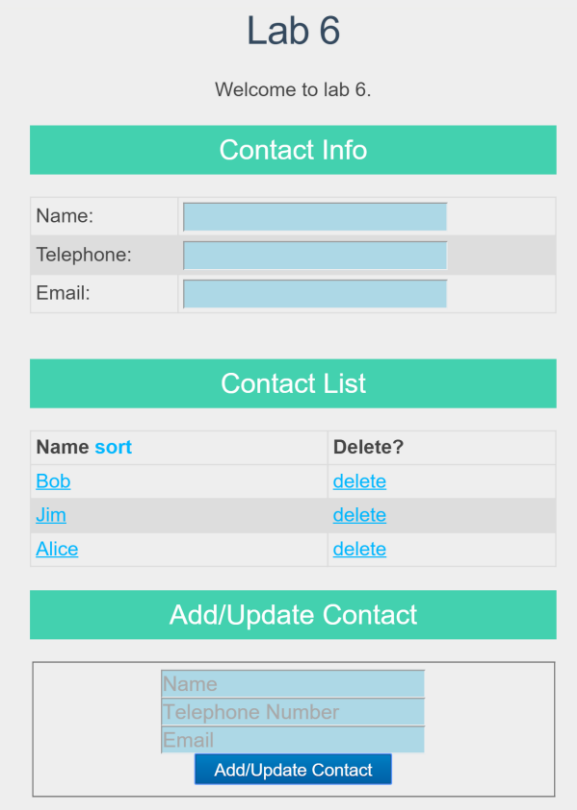

Fig. 11 Before clicking "sort"

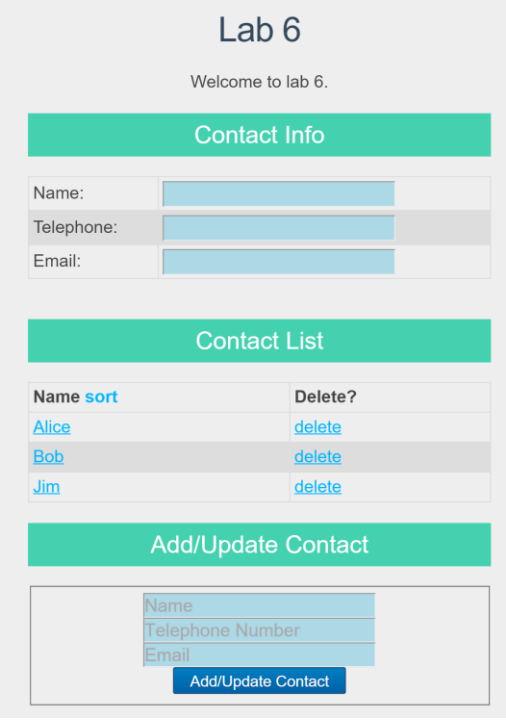

Fig. 12 after clicking "sort" for the first time

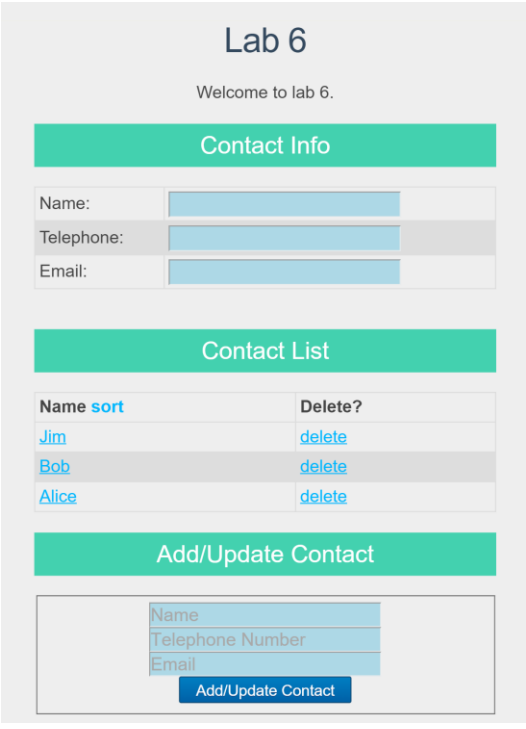

Fig. 13 after clicking "sort" again

Submission:

You should submit the following files and folders only:

(1) app.js
(2) the “public” folder
(3) the “routes” folder
(4) the “views” folder

Please compress the above files/folders in a .zip file and submit it on Moodle before 23:59 Wednesday Nov. 13, 2019:

(1) Login Moodle.
(2) Find “Labs” in the left column and click “Lab 6”.
(3) Click “Add submission”, browse your .zip file and save it. Done.
(4) You will receive an automatic confirmation email, if the submission was successful.
(5) You can “Edit submission” to your already submitted file, but ONLY before the deadline.