**COMP3322 Modern Technologies on World Wide Web**

**Lab 6: RESTful Web Service Using Node.js**

# Introduction

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

# Lab Exercise 1: Create an Express project

Follow the instructions in setup\_nodejs\_runtime.pdf to create an Express project called lab6.

# Lab Exercise 2: Create the Home Page Using Pug

We next modify the Pug templates in the “./views” directory of “lab6”, in order to render the home page of our Express app.

**Step 1**: Open **index.pug** using a text editor. Modify it to contain the following content:

|  |
| --- |
| extends layout  block content  h1= title  p Welcome to lab 6.  // Wrapper  #wrapper  // CONTACT INFO  #contactInfo  h2 Contact Info  p  strong Name:  | <span id='contactInfoName'></span>  br  strong Telephone:  | <span id='contactInfoTel'></span>  br  strong Email:  | <span id='contactInfoEmail'></span>  br  // /CONTACT INFO  // CONTACT LIST  h2 Contact List  #contactList  table  thead  th Name  th Delete?  tbody  // /CONTACT LIST  // ADD CONTACT  h2 Add Contact  #addContact  fieldset  input#inputContactName(type='text', placeholder='Name')  br  input#inputContactTel(type='text', placeholder='Telephone Number')  br  input#inputContactEmail(type='text', placeholder='Email')  br  button#btnAddContact Add Contact  // /ADD CONTACT  // /WRAPPER |

Please refer to https://pugjs.org/api/getting-started.html for explanations of the code in the template. Make sure your code is indented exactly as the given code above.

**Step 2**: Open **layout.jade** 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='http://ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js')  script(src='/javascripts/externalJS.js') |

The first line of code in **index.jade** indicates that **index.jade** extends **layout.jade**. By modifying **layout.jade** as above, the web page rendered will link to a **style.css** file for styling (there is a default style.css file under ./public/stylesheets produced when the project was created) and an **externalJS.js** file for client-side JavaScripts, which we will create under the directory ./public/javascripts in Lab Exercise 4. 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 there are two underscores “\_” before dirname in the code). In this way, the render 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 the following:



# Lab Exercise 3: Install MongoDB

We will need a MongoDB database to store students’ information. We first need to install the database.

**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**: Go to <https://www.mongodb.org/> and download the latest version of MongoDB (you should download the 32bit version if you are working on a lab computer, and a 64-bit version if you work on your own computer which has a 64bit processor). Install MongoDB to a specific directory.

**Step 3**: **Launch the 2nd terminal**, 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: (assuming that the “lab6” project directory is created in the home directory)

|  |
| --- |
| ./bin/mongod --dbpath ~/lab6/data |

After starting the database server successfully, you should see some prompt in the terminal like “NETWORK [initialization] waiting for connection on port 27017”. This means that the database server is up running now. **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 4**: **Open 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.

# When you finish executing the above commands, you should see “WriteResult({ "nInserted" : 1 })” on the terminal. You can insert more records into the database collection to facilitate testing of your program.

**Step 5**: Now switch to the “lab6” project folder, open package.json file and replace the content of package.json with the following content:

|  |
| --- |
| {  "name": "lab6",  "version": "0.0.0",  "private": true,  "scripts": {  "start": "node ./bin/www"  },  "dependencies": {  "body-parser": "~1.15.2",  "cookie-parser": "~1.4.3",  "debug": "~2.2.0",  "express": "~4.14.0",  "jade": "~1.11.0",  "morgan": "~1.7.0",  "pug": "^2.0.0-beta6",  "serve-favicon": "~2.3.0",  "mongodb": "^2.2.11",  "monk": "^3.1.3"  }  } |

Then execute **npm install** command.

After this, we add 2 more Node.js packages to “lab6” project, which are **mongodb** and **monk.** These 2 packages are used to interact with the MongoDB.

# Lab Exercise 4: List Contacts

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

**Step 1**: Open **app.js** and add the following lines ***below*** “var bodyParser = require('body-parser');”. By doing so, we establish a connection with the database “lab6” that we created in Lab Exercise 3.

|  |
| --- |
| // Database  var mongo = require('mongodb');  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('/', routes);”.

|  |
| --- |
| // Make our db accessible to our router  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 ./routesand 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(e,docs){  res.json(docs);  });  });  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 get the ‘contactList’ collection, encode everything inside 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:

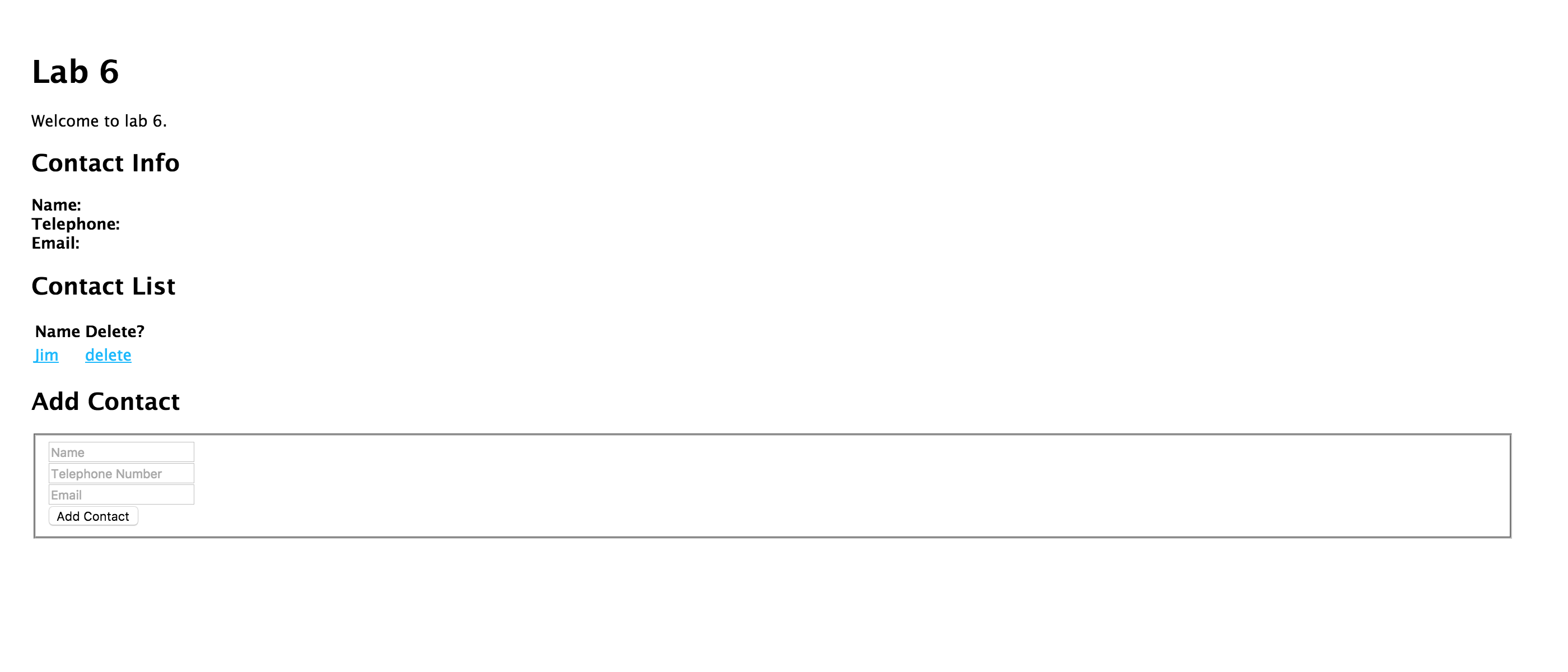
[{"\_id":"582e741ac1f51204644fb50e","name":"Jim","email":"jim@gmail.com","tel":"1234567"}]

We can see that a "\_id" attribute was added by the database server into each contact record 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 retrieved in the contact record.

**Step 4**: Now we add client-side code for displaying the user list. Recall that in Step 2 of Lab Exercise 2, we link the rendered web 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 = [];  // 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 record(s) into the contact list table. You should see that the contact record that we inserted into the database earlier is now displayed on the web page:



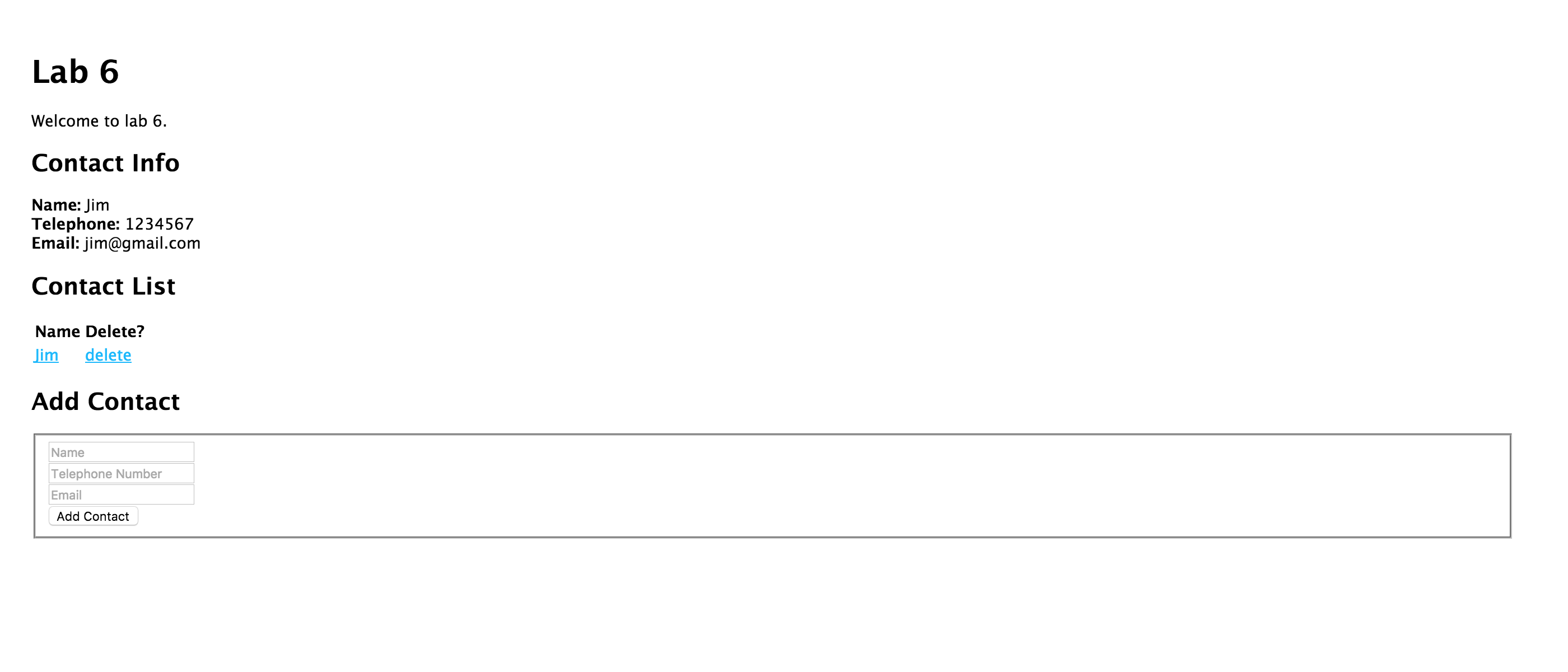
# Lab Exercise 5: Show Detailed Contact Information

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 4, 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 id value  var arrayPosition = contactListData.map(function(arrayItem) { return arrayItem.name; }).indexOf(thisContactName);  // Get our contact Object  var thisContactObject = contactListData[arrayPosition];  //Populate Info Box  $('#contactInfoName').text(thisContactObject.name);  $('#contactInfoTel').text(thisContactObject.tel);  $('#contactInfoEmail').text(thisContactObject.email);  };  // 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 Student List. You should see that the detailed information of the student is displayed in the Student Info part:



# Lab Exercise 6: 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> by adding the request body directly to the contactList collection in the database.

|  |
| --- |
| /\*  \* 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 }  );  });  }); |

**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 Contact”button is clicked, the **addContact**function will be called. **addContact**will first check if all fields in the “**#addContact**” division have been filled: if not, it prompts 'Please fill in all fields'; if so, an AJAX HTTP POST request will be sent to “http://localhost:3000/users/addContact”, carrying a JSON message containing the input information about the new contact inside its body. Upon receiving a success HTTP response, the client clears all the fields “**#addContact**” division, and updates the contact list by calling **populateContactList ()***.*

|  |
| --- |
| // Add contact  function addContact(event) {  event.preventDefault();  // Super basic validation - increase errorCount variable if any fields are blank  var errorCount = 0;  $('#addContact input').each(function(index, val) {  if($(this).val() === '') { errorCount++; }  });  // Check and make sure errorCount's still at zero  if(errorCount === 0) {  // If it is, compile all contact information into one object  var newContact = {  'name': $('#addContact fieldset input#inputContactName').val(),  'tel': $('#addContact fieldset input#inputContactTel').val(),  'email': $('#addContact 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  $('#addContact 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 Contact button click  $('#btnAddContact').on('click', addContact); |

**Step 3**: Restart your Express app and browse <http://localhost:3000> again. Add information of a new contact as in the top figure. After clicking the “Add Contact” button, you should see a page as shown in the bottom figure

# ../../../Desktop/Screen%20Shot%202016-11-18%20at%2011.54.08%20A

# ../../../Desktop/Screen%20Shot%202016-11-18%20at%2011.56.12%20A

# Lab Exercise 7: 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” link in the Contact List is clicked.

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

|  |
| --- |
| /\*  \* DELETE to deleteContact.  \*/  router.delete(?, function(req, res) {  ?  }); |

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

1. In the code we added in Step 4 of Lab Exercise 4, 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 a “**delete”** link:

[http://localhost:3000/users/deleteContact/xx](http://localhost:3000/users/deletestudent/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,** andretrieve 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 student confirmed  if (confirmation === true) {  // If they did, do our delete  $.ajax({  type: ?,  url: ?  }).done(function( response ) {  ?  });  }  else {  // If they said no to the confirm, do nothing  return false;  }  }; |

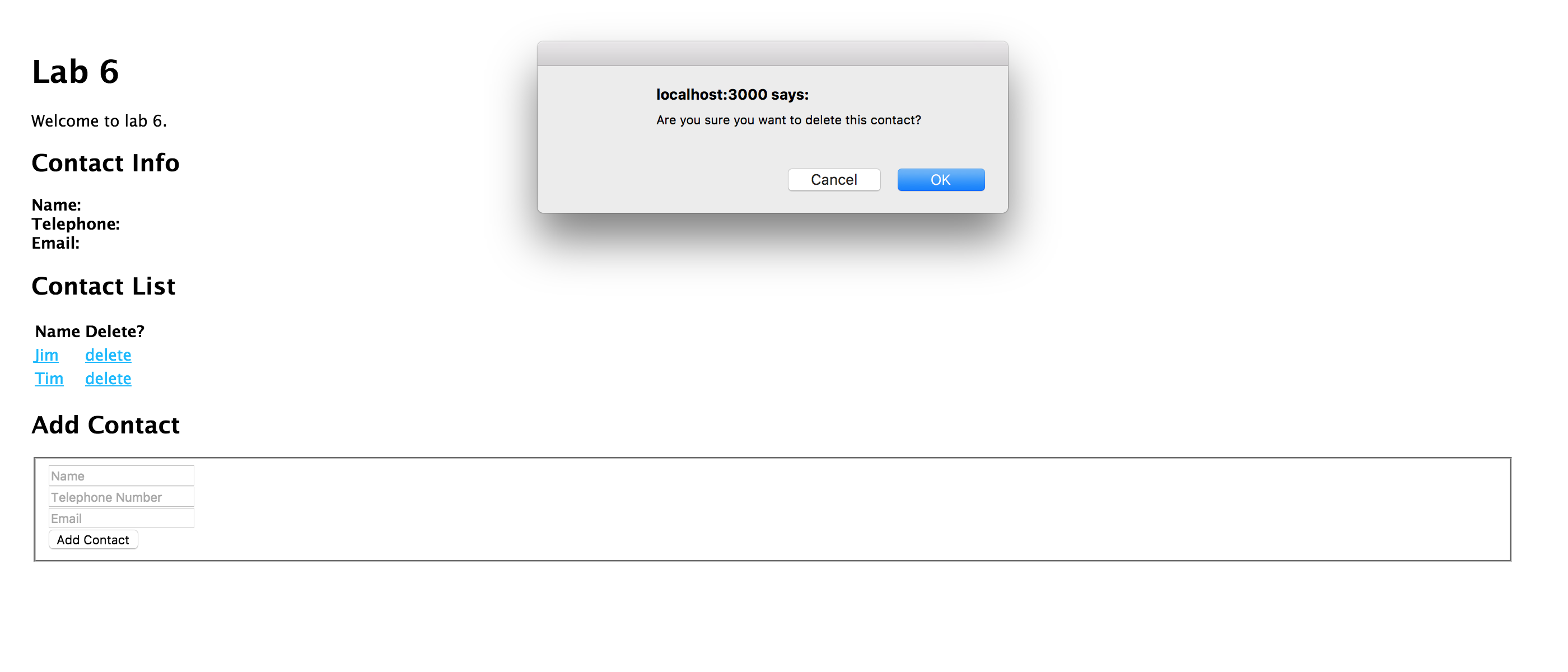
Replace “?” with correct code to finish the client-side code for sending an AJAX HTTP DELETE request and handling the response. You should follow these hints:

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.

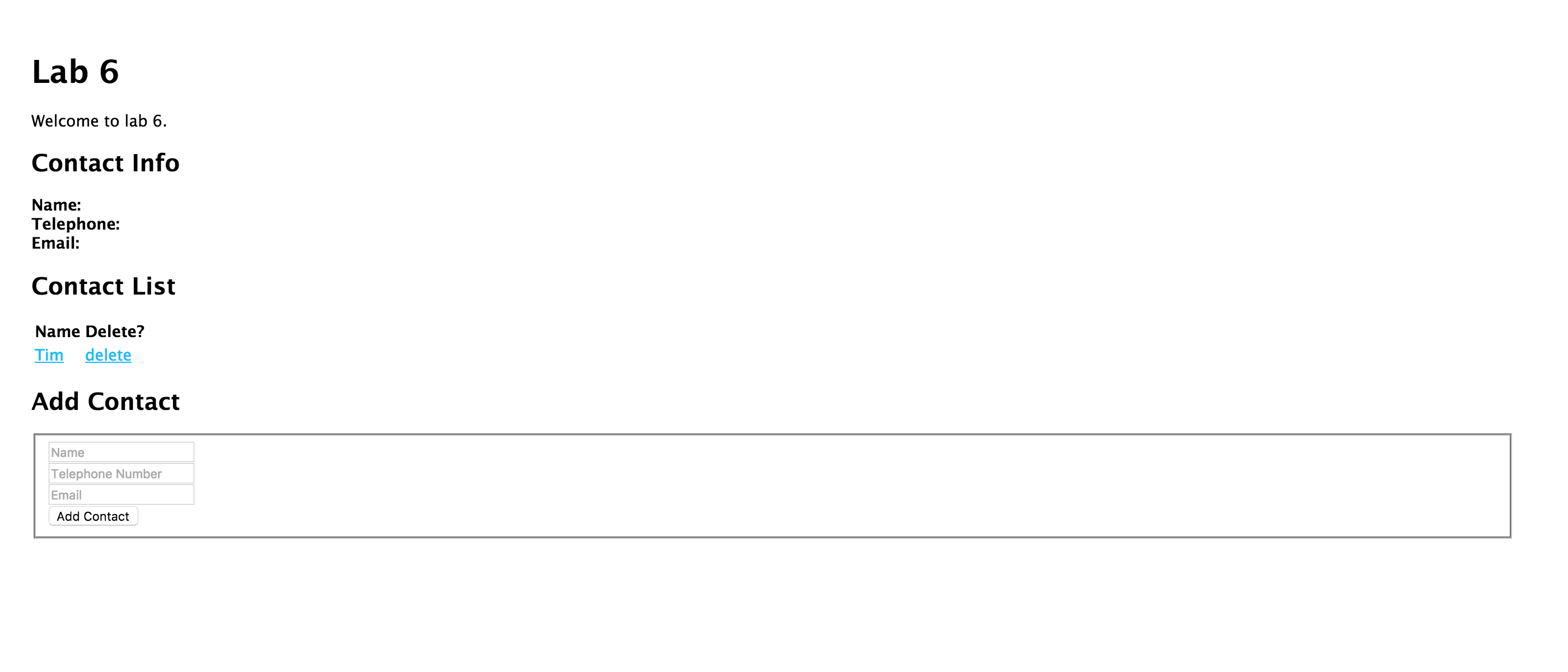
2. You should fill in correct type and url of the HTTP DELETE request in the **$.ajax** method call.

3. Upon successful deletion, you should refresh the “Contact List” displayed on the web page; otherwise, display the error message carried in the response.

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



After clicking “delete” in the row of “Jim”



After confirming deletion by clicking “OK”

# Submission:

Create a .zip file named **lab6.zip** which should contain **app.js**, **package.json**, **the public folder**, **the routes folder** and **the views folder (including all files in these folders)**. Please upload lab7.zip to i.cs.hku.hk web server before **23:59 Wednesday Dec. 2**, such that it can be accessed at the following URL:

**http://i.cs.hku.hk/~[your csid]/lab6/lab6.zip**