

COMP3322A Modern Technologies on World Wide Web

Lab 7: RESTful Web Service Using the Mean Stack

Introduction

In this lab exercise, we will re-implement the client-side of the RESTful Web service in lab6 using AngularJS. Similar to lab 6, the web service allows retrieving, adding, updating and deleting contacts from a MongoDB database. The client-side AngularJS application provides the web page for displaying contact list, adding, updating and deleting the contacts. In this way, the web service is implemented with the MEAN (MongoDB, Express.js, AngularJS and Node.js) stack.

Set Up the Express Project

We will reuse the express server app implemented in Lab 6. Therefore, make a copy of your “lab6” project folder (which you created when working on Lab 6) and name it “lab7”. We will reuse the modules you have installed for the express server app (in “./node_modules” folder), and the MongoDB database you created in Lab 6 (the database file are stored under ./data folder).

Open **app.js** and comment out the following lines of code, since we do not need the pug view engine in this project. You can also remove the ./views folder.

```
app.set('views',path.join(_dirname,'views'));  
app.set('views engine', 'pug');
```

Instead, we will implement the client-side code in **index.html** and **externalJS.js** which will be placed under ./public and ./public/javascripts, respectively, and accessed as static files. Therefore, comment out the following lines of code in **app.js** (those for generating the homepage upon receiving requests for “/”) as well. You can also remove **index.js** from the ./routes folder.

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

All other lines of code in **app.js** can remain, including those for loading MongoDB related modules. Note that since we are reusing the database created in Lab6, the database in the code should still be 'localhost:27017/lab6'.

In addition, we will use the same router ./routes/users.js which you implemented in Lab 6 as the back-end in this web service application.

Launch the MongoDB Database

We will follow the same steps as done in Lab 6 to launch the database server. Launch a terminal and switch to the directory where MongoDB is installed. Start MongoDB server using the “data” directory of “lab7” project as the database location, as follows: (replace “YourPath” by the actual path on your computer that leads to “lab7” directory)

```
./bin/mongod --dbpath YourPath/lab7/data
```

After starting the database server successfully, you should see some prompt in the terminal like "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.

Launch another terminal and switch to the directory where mongodb is installed. Then you can use the following commands to add contacts into the database, if you need them for testing purpose. (If there are still contacts in your "lab6" database as you inserted last time, you do not have to do this step.)

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

Lab Exercise: Create the AngularJS web client-side

Task 1: Download **lab7-materials.zip** from Moodle, unzip it and you will see two files: **index.html** and **externalJS.js**. Copy **index.html** to the **./public/** folder and **externalJS.js** to **./public/javascripts/** folder. The given **index.html** contains HTML content of the page. We will implement AngularJS code in both files.

Like in Lab 6, you can always check out the web page as follows. Launch a terminal, switch to your "lab7" directory, and 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**). Then check out the web page at <http://localhost:3000/index.html> on your browser. You should see a page like the following:

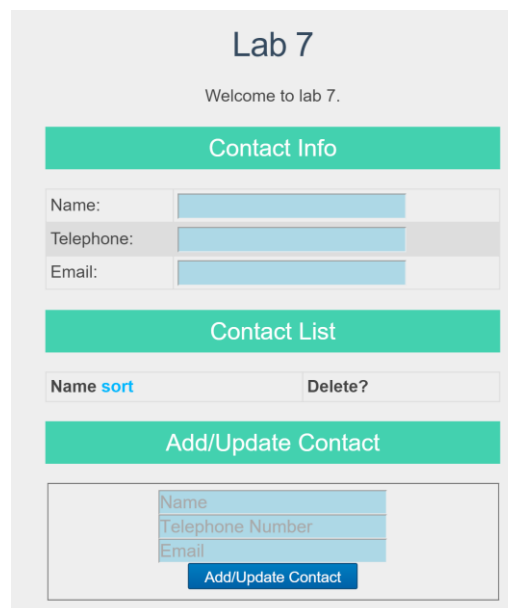


Fig. 1

Task 2: Open **index.html** and add the following code into the **<head>** element, which loads the AngularJS framework.

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
```

Then add the following attributes into the **<body>** tag. Here directives **ng-app** and **ng-controller** define the AngularJS application and controller, respectively.

```
ng-app="contactList" ng-controller="contactListController"
```

Open **externalJS.js**. You will see there are codes to create an AngularJS module (i.e., AngularJS application “contactList”) and add the controller “contactListController” into the application.

Task 3: We add codes for displaying the contact list in externalJS.js. Add the following code into the constructor of the “contactListController” controller (where we mark “to be completed” in externalJS.js).

```
$scope.contacts = null;

$scope.getContacts = function(){
    $http.get("/users/contactList").then(function(response){
        $scope.contacts = response.data;
    }, function(response){
        alert("Error getting contacts.");
    });
};
```

The function `getContacts()` sends an AJAX HTTP GET request for <http://localhost:3000/users/contactList>. Recall that on the server side, the end point is handled by the following middleware in `./routes/users.js` (which we implemented during Lab 6):

```
/*
 * 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});
    });
});
```

After receiving server response, `getContacts()` stores the received JSON object in variable `contacts` (in the `$scope` object), which can be used in the AngularJS code in **index.html**.

Add the following directive as the last attribute in the `<body>` tag in **index.html**. `ng-init` executes some code when the AngularJS application is initialized (http://www.w3schools.com/angular/ng_ng-init.asp).

```
ng-init="getContacts()"
```

Then add the following code into `<tbody> </tbody>` element in **index.html**.

```
<tr ng-repeat="contact in contacts">
    <td> <a href="" ng-click="showContact(contact)">{{contact.name}}</a></td>
    <td> <a href="" ng-click="deleteContact(contact._id)">delete</a></td>
</tr>
```

In this way, `getContacts()` is executed when the application is initialized, and the name of

contacts received from the server are displayed underneath “Contact List” on the page. Note here we add several event handlers for “ng-click” event on the `<button>` elements, which we will implement in following tasks.

Browse the web page at <http://localhost:3000/index.html>. You should see that the contact record that you inserted into the database is displayed on the page:

The screenshot shows a web application titled "Lab 7" with a subtitle "Welcome to lab 7.". The interface is divided into three main sections, each with a green header bar:

- Contact Info:** Contains three input fields labeled "Name:", "Telephone:", and "Email:".
- Contact List:** Contains a table with two columns: "Name" and "Delete?". The table has one row with the name "Jim" and a "delete" button.
- Add/Update Contact:** Contains three input fields labeled "Name", "Telephone Number", and "Email", followed by an "Add/Update Contact" button.

Fig. 2

Task 4: We next add 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. In `externalJS.js`, add the following code in the constructor of the “`contactListController`” controller.

```
$scope.selected_contact = {_id:"", name:"", tel:"", email:""};

$scope.showContact = function(contact){
  $scope.selected_contact = contact;
};
```

In `index.html`, add the following three expressions in `<input>` elements following “Name:”, “Telephone:”, “Email:” (underneath “Contact Info”), respectively.

```
ng-model="selected_contact.name"
ng-model="selected_contact.tel"
ng-model="selected_contact.email"
```

In the above codes, we create a `selected_contact` object, and bind its values with the content to be displayed under “Contact Info” on the page. When a contact name in the “Contact List” is clicked, the contact’s information will be assigned to the `selected_contact` object and displayed on the page.

Browse <http://localhost:3000/index.html> 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:

Lab 7

Welcome to lab 7.

Contact Info

Name:	<input type="text" value="Jim"/>
Telephone:	<input type="text" value="1234567"/>
Email:	<input type="text" value="jim@gmail.com"/>

Contact List

Name sort	Delete?
Jim	delete

Add/Update Contact

Name
Telephone Number
Email

Fig. 3

Task 5: We next add code for adding a new contact record into the database. In **externalJS.js**, add the following code in the constructor of the “contactListController” controller.

```
$scope.addOrUpdateContact = function(contact){
    // Super basic validation - increase errorCount variable if any fields are blank
    var errorCount = 0;
    if(contact.name == "" || contact.tel == "" || contact.email == ""){
        alert('Please fill in all fields');
        return false;
    }
    var existContactIndex = -1;
    if ($scope.contacts != null){
        for (var i = 0; i < $scope.contacts.length; i++){
            if($scope.contacts[i].name == contact.name){
                existContactIndex = i;
                break;
            }
        }
    }
    if(existContactIndex >= 0){
        var existingContact = {
            '_id': $scope.contacts[existContactIndex]._id,
            'name': contact.name,
            'email': contact.email,
            'tel': contact.tel
        };
        $scope.updateContact(existingContact);
    } else{
        var newContact = {
            'name': contact.name,
            'email': contact.email,
            'tel': contact.tel
        }
        $http.?(?).then(function(response){
            if(response.data.msg==""){
                $scope.getContacts();
            }
        });
    }
}
```

```

        $scope.new_contact = { _id:"", name:"", tel:"", email:"" };
    }
    else{
        alert("Error adding contact.");
    }
}, function(response){
    alert("Error adding contact.");
});
}
};

```

Replace “?” with the correct code for sending an AJAX HTTP POST request for <http://localhost:3000/users/addContact>. Recall that the end point is handled by the following middleware in ./routes/users.js.

```

/*
 * 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 }
        );
    });
});

```

Add the following three directives in the text `<input>` elements below “addOrUpdateContact” division in **index.html** for Name, Telephone Number and Email, respectively. The `ng-model` directives bind values of the input elements with the variables in the `new_contact` object.

```

ng-model="new_contact.name"
ng-model="new_contact.tel"
ng-model="new_contact.email"

```

Add the following directive in the “Add/Update Contact” `<button>` element, to register `addContact ()` as the event handler for “ng-click” on the button.

```

ng-click="addOrUpdateContact(new_contact)"

```

The above codes in **externalJS.js** and **index.html** achieve the following functionalities: after the new contact’s information is typed and the “Add/Update Contact” button is clicked, the `addOrUpdateContact` function will be invoked. `addOrUpdateContact` first checks if all input fields 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 input fields, and updates the contact list by calling `getContacts()`.

Restart your Express app and browse <http://localhost:3000/index.html> again. Add information of a new contact. After clicking the “Add/Update Contact” button, you should see a page as in the figure below.

Lab 7

Welcome to lab 7.

Contact Info

Name:

Telephone:

Email:

Contact List

Name sort	Delete?
Jim	delete
Bob	delete

Add/Update Contact

Name
Telephone Number
Email

Add/Update Contact

Fig. 4

Task 6: Now we implement the code for updating an existing contact in the database. Recall that update is handled by the following middleware in `./routes/users.js`:

```

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

  var filter = { "_id": contactToUpdate };
  collection.update(filter, { $set: { "name": req.body.name, "email": req.body.email, "tel":
req.body.tel } }, function (err, result) {
    res.send(
      (err === null) ? { msg: " " } : { msg: err }
    );
  })
});

```

In `externalJS.js`, add the following code:

```

$scope.updateContact = function(contact){
  var id = contact._id;
  var url = '?';
  $http.?(?).then(function(response){
    if (response.data.msg === ""){
      $scope.getContacts();
      // Clear the form inputs
      ?
      // Update detailed info
      ?
    } else{
      alert(response.data.msg);
    }
  })
}

```

```

    }, function(response){
        alert("Error updating contact");
    });
};

```

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()`.

Restart your Express app, browse <http://localhost:3000/index.html> again, and test the update function as follows:

Lab 7

Welcome to lab 7.

Contact Info

Name: Bob

Telephone: 7654321

Email: bob@gmail.com

Contact List

Name sort	Delete?
Jim	delete
Bob	delete

Add/Update Contact

Bob

1234567

bob19@gmail.com

[Add/Update Contact](#)

Fig. 5

Lab 7

Welcome to lab 7.

Contact Info

Name: Bob

Telephone: 1234567

Email: bob19@gmail.com

Contact List

Name sort	Delete?
Jim	delete
Bob	delete

Add/Update Contact

Name

Telephone Number

Email

[Add/Update Contact](#)

Fig. 6

Task 7: Now we implement the code for deleting a contact from the database, when a respective “delete” button in the Contact List is clicked. Recall that delete is handled by the following middleware in `./routes/users.js`:


```

/*
 * DELETE to deleteContact.
 */
router.delete('/deleteContact/:id', function(req, res) {
  var db = req.db;
  var contactID = req.params.id;
  var collection = db.get('contactList');
  collection.remove({'_id':contactID}, function(err, result){
    res.send((err === null)?{msg:''}:{msg:err});
  });
});

```

In **externalJS.js**, add the following code:

```

$scope.deleteContact = function(id){
  // 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) {
    var url = ?;

    $http.?(?).then(function(response){
      ?
    }, function(response){
      alert("Error deleting contacts.");
    });
  } else {
    // If they said no to the confirm, do nothing
    return false;
  }
};

```

Replace “?” with correct code to complete the client-side code for sending an AJAX HTTP DELETE request and handling the response. Your implementation should achieve the following: upon successful deletion, you should refresh the “Contact List” displayed on the web page; otherwise, display the error message. If the detailed contact information of the deleted contact is being displayed under “Contact Info”, remove the contact info there.

Restart your Express app, browse <http://localhost:3000/index.html> again, and test the delete function as follows:

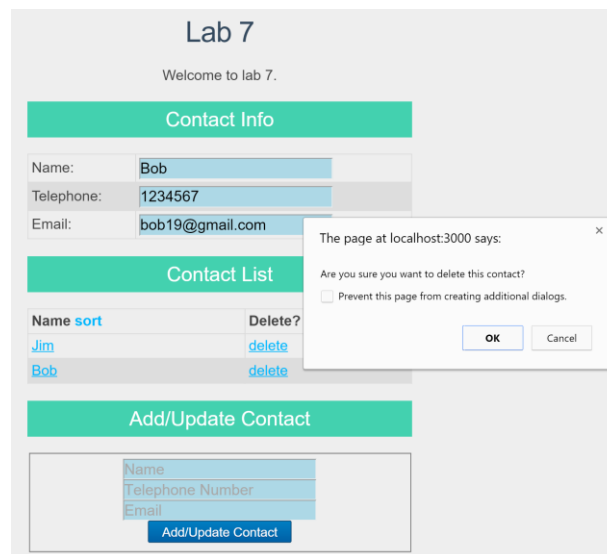


Fig. 7

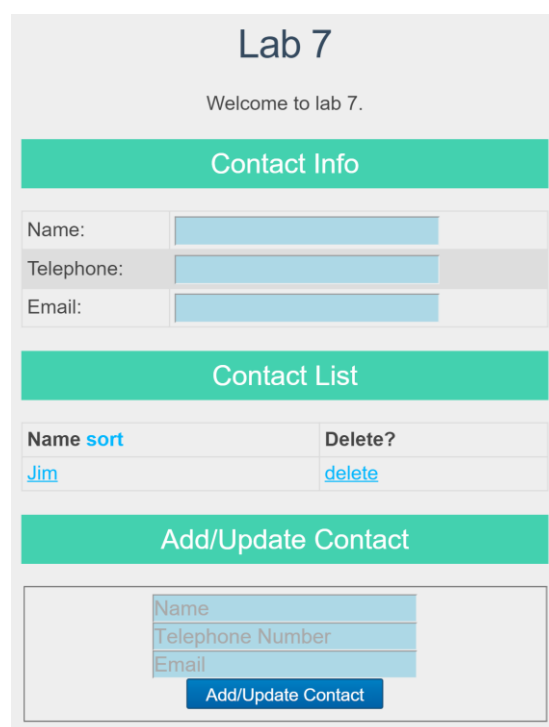


Fig. 8

Task 8: Now we implement the code for sorting the contacts in the contact list, when “sort” in the Contact List is clicked. 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.

In **index.html**, add the following directive in the “sort” `<a>` element, to register `sortList()` as the event handler for “ng-click” on the link.

```
ng-click="sortList()"
```

Add and implement the following function in **externalJS.js**, to achieve the above sort functionalities. **Hints:** learn from how we implemented “function sortList()” in Lab 6; and here with AngularJS, we do not need to reconstruct the HTML content and the code is simpler.

```
// for sorting order
$scope.order = 1;

$scope.sortList = function() {
  //to complete
}
```

Restart your Express app, browse <http://localhost:3000/index.html> again, and test the delete function as follows:

The screenshot shows the 'Lab 7' application interface. At the top, it says 'Welcome to lab 7.' Below this is a 'Contact Info' section with three input fields for 'Name:', 'Telephone:', and 'Email:'. Underneath is a 'Contact List' section with a table. The table has two columns: 'Name' and 'Delete?'. The 'Name' column has a link 'sort' next to it. The table contains three rows: 'Jim', 'Bob', and 'Alice', each with a 'delete' link in the 'Delete?' column. Below the table is an 'Add/Update Contact' section with three input fields for 'Name', 'Telephone Number', and 'Email', and an 'Add/Update Contact' button.

Fig. 9 Before clicking “sort”

The screenshot shows the 'Lab 7' application interface after clicking 'sort'. The 'Contact List' table now shows the contacts sorted alphabetically: 'Alice', 'Bob', and 'Jim'. The 'Name' column still has the 'sort' link, and each row has a 'delete' link in the 'Delete?' column. The 'Add/Update Contact' section remains the same.

Fig. 10 After clicking “sort” for the first time

The screenshot shows the 'Lab 7' application interface after clicking 'sort' again. The 'Contact List' table now shows the contacts sorted alphabetically: 'Jim', 'Bob', and 'Alice'. The 'Name' column still has the 'sort' link, and each row has a 'delete' link in the 'Delete?' column. The 'Add/Update Contact' section remains the same.

Fig. 11 After clicking “sort” again

Submission:

You should submit the following files and folders only:

- (1) app.js
- (2) index.html

(3) externalJS.js

Please compress the above files in a .zip file and submit it on Moodle before **23:59 Wednesday Nov. 27, 2019**:

- (1) Login Moodle.
- (2) Find "Labs" in the left column and click "Lab 7".
- (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.